



Nota:

Este tutor está basado en una traducción-adaptación del tutorial de Sun que puedes encontrar: [aquí](#)

Tu Primera 'Taza' de Java

- [La Primera 'Taza' de Java](#)
 - [En Windows](#)
 - [En UNIX](#)
 - [En Mac OS](#)

¿Por donde Empezar?

- [¿Por dónde Empezar?](#)
- [El compilador Javac](#)
- [El visualizador de Applets](#)
- [El intérprete Java](#)
- [El decompilador Javap](#)

Conceptos Básicos

- [Conceptos Básicos de Programación Orientada a Objetos](#)

Características del Lenguaje

- [Variables y Tipos de Datos](#)
- [Operadores](#)
- [Expresiones](#)
- [Control de Flujo](#)
- [Arrays y Cadenas](#)

Objetos, Clases e Interfaces

- [Crear Objetos](#)
- [Utilizar Objetos](#)
- [Eliminar Objetos no utilizados](#)
- [Declarar Clases](#)
- [El cuerpo de la Clase](#)
- [Declarar Variables Miembro](#)
- [Declarar Métodos](#)
- [Pasar Argumentos a un Método](#)
- [El cuerpo de un Método](#)
- [Miembros de la Clases y del Ejemplar](#)
- [Control de Acceso a Miembros](#)
- [Constructores](#)
- [Construir un Metodo Finalize](#)
- [Subclases y SuperClases](#)
- [Crear una Subclase](#)
- [Sobreescribir Métodos](#)
- [Clases y Métodos Finales](#)
- [Clases y Métodos Abstractos](#)
- [La clase Object](#)
- [¿Qué es un Interface?](#)
- [Crear un Interface](#)
- [Utilizar un Interface](#)
- [Usar un Interface como un Tipo](#)
- [Crear Paquetes de Clases](#)
- [Utilizar Paquetes](#)
- [Los Paquetes Internos del JDK](#)
- [Cambios en el JDK 1.1.x](#)

La clase String

- [String y StringBuffer](#)
- [¿Por qué dos clases String?](#)
- [Crear String y StringBuffer](#)

- [Métodos Accesores](#)
- [Modificar un StringBuffer](#)
- [Convertir Objetos a Stings](#)
- [Los Strings y el Compilador Javac](#)
- [Notas sobre el JDK 1.1](#)

Atributos del Programa

- [Atributos del Programa](#)
- [Seleccionar y Utilizar Propiedades](#)
- [Argumentos de la Línea de Comandos](#)
 - [Convenciones](#)
 - [Analizar Argumentos](#)
- [Notas sobre el JDK 1.1](#)

Recursos del Sistema

- [Recursos del Sistema](#)
- [Utilizar la Clase System](#)
- [Los Streams de I/O Estándar](#)
- [Propiedades del Sistema](#)
- [Recolección de Basura](#)
- [Otros Métodos de la Clase System](#)
- [Notas sobre el JDK 1.1](#)

Manejo de Errores

- [Manejo de Errores utilizando Excepciones](#)
- [¿Qué es una Excepción?](#)
- [Primer encuentro con las Excepciones](#)
- [Declarar o Expecificar](#)
- [Tratar con Excepciones](#)
 - [El ejemplo ListOfNumbers](#)
 - [Capturar y Manejar Excepciones](#)
 - [El bloque try](#)
 - [Los bloques catch](#)

- [El bloque finally](#)
- [Juntándolo todo](#)
- [Especificar Excepciones](#)
- [La sentencia throw](#)
- [La clase Throwable](#)
- [Crear Clases de Excepciones](#)
- [Excepciones en Tiempo de Ejecución](#)
- [Notas sobre el JDK 1.1](#)

Threads de Control

- [Threads de Control](#)
- [¿Qué es un Thread?](#)
- [Sencillo Thread de Ejemplo](#)
- [Atributos de un Thread](#)
 - [El cuerpo de un Thread](#)
 - [El applet del Reloj](#)
 - [Estados de un Thread](#)
 - [Prioridad de un Thread](#)
 - [Threads Servidores](#)
 - [Grupos de Threads](#)
 - [La clase ThreadGroup](#)
- [Programas Multi-Thread](#)
- [Sincronización de Threads](#)
 - [Monitores Java](#)
 - [Los monitores Java son reentrantes](#)
 - [Los métodos wait\(\) y notify\(\)](#)
- [Notas sobre el JDK 1.1](#)

Canales de I/O

- [Streams de I/O](#)
- [Primer encuentro con la I/O en Java](#)
- [Introducción a los Streams](#)
- Utilizar Streams de I/O

- [Implementar Tuberías](#)
- [I/O de Ficheros](#)
- [I/O sobre Memoria](#)
- [Concatener Ficheros](#)
- [Streams Filtrados](#)
 - [DataInputStream y DataOutputStream](#)
 - [Escribir Streams Filtrados](#)
- [Ficheros de Acceso Aleatorio](#)
 - [Utilizar Ficheros de Acceso Aleatorio](#)
 - [Filtros para Ficheros de Acceso Aleatorio](#)
- [Notas sobre el JDK 1.1](#)

Los Applets

- [Introducción a los Applets](#)
 - [Anatomía de un Applet](#)
 - [Importar Clases y Paquetes](#)
 - [Definir una subclase de Applet](#)
 - [Implementar métodos en un Applet](#)
 - [Ejecutar un Applet](#)
- [Descripción de un Applet](#)
 - [El ciclo de vida de un Applet](#)
 - [Métodos para Millestones](#)
 - [Métodos para Dibujar y manejar Eventos](#)
 - [Usar Componentes UI](#)
 - [Threads en un Applet](#)
 - [Ejemplos](#)
 - [Qué puede y qué no puede hacer un Applet](#)
 - [Añadir un Applet a una página HTML](#)
 - [Sumario](#)
- [Crear un Interface de Usuario](#)
 - [Crear un GUI](#)
 - [Ejecutar Sonidos](#)
 - [Usar Parámetros en en un Applet](#)

- [Parámetros a Soportar](#)
- [Escribir código para soportar Parámetros](#)
- [Obtener información sobre Parámetros](#)
- [Leer las Propiedades del Sistema](#)
- [Mostrar cadenas de Estado](#)
- [Diagnóstico en la Salida Estándar](#)
- [Comunicarse con otros programas](#)
 - [Enviar Mensajes a otros Applets](#)
 - [Comunicación con el Navegador](#)
 - [Aplicaciones en el lado del Servidor](#)
- Capacidades y Restricciones en un Applet
 - [Restricciones de Seguridad](#)
 - [Capacidades de un Applet](#)
- Finalizar un Applet
 - [Antes de Liberar un Applet](#)
 - [Finalización perfecta de un Applet](#)
- [Problemas Comunes con los Applets \(y sus soluciones\)](#)
- [Notas sobre el JDK 1.1](#)

Interface Gráfico de Usuario

- [Introducción al UI de Java](#)
 - [Componentes de la clase AWT](#)
 - [Otras Clases AWT](#)
 - [La Anatomía de un programa basado en GUI](#)
 - [Las clases del Programa Ejemplo](#)
 - [La Herencia de Componentes](#)
 - [Dibujo](#)
 - [Manejo de Eventos](#)
- [Utilizar Componentes AWT](#)
 - [Reglas Generales](#)
 - [Utilizar Button](#)
 - [Utilizar Canvas](#)
 - [Utilizar Checkbox](#)

- [Utilizar Choice](#)
- [Utilizar Dialog](#)
- [Utilizar Frame](#)
- [Utilizar Label](#)
- [Utilizar List](#)
- [Utilizar Menu](#)
- [Utilizar Panel](#)
- [Utilizar Scrollbar](#)
- [Utilizar Campos y Areas de Texto](#)
- [Detalles de la Arquitectura de Componentes](#)
- [Problemas con los Componentes](#)
- [Distribuir Componentes](#)
 - [Utilizar Manejadores de Distribución](#)
 - [Reglas Generales](#)
 - [BorderLayout](#)
 - [CardLayout](#)
 - [FlowLayout](#)
 - [GridLayout](#)
 - [GridBagLayout](#)
 - [Especificar Restricciones](#)
 - [El Applet de Ejemplo](#)
 - [Crear un Controlador Personalizado](#)
 - [Posicionamiento Absoluto](#)
 - [Problemas con los controladores](#)
- [Introducción a los Gráficos del AWT](#)
 - [Dibujar Formas Sencillas](#)
 - [Dibujar Texto](#)
 - [Utilizar Imagenes](#)
 - [Cargar Imagenes](#)
 - [Mostrar Imagenes](#)
 - [Manipular Imagenes](#)
 - [Utilizar un Filtro](#)
 - [Escribir un Filtro](#)

- [Realizar Animaciones](#)
 - [Crear un Bucle](#)
 - [Animar Gráficos](#)
 - [Eliminar el Parpadeo](#)
 - [Sobreescribir el método update\(\)](#)
 - [Doble Buffer](#)
 - [Mover una Imagen](#)
 - [Mostrar Secuencias de Imágenes](#)
 - [Aumentar el rendimiento de una Animación](#)
- [Problemas comunes con los Gráficos](#)

Gráficos 2D

- [Introducción al API 2D de Java](#)
 - [Dibujado Java 2D](#)
 - [Sistema de Coordenadas](#)
 - [Formas](#)
 - [Texto](#)
 - [Imágenes](#)
 - [Imprimir](#)
- [Mostrar Gráficos con Graphics2D](#)
 - [Rellenar y patronar gráficos primitivos](#)
 - [Transformar formas texto e imágenes](#)
 - [Recortar la región de dibujo](#)
 - [Componer Gráficos](#)
 - [Controlar la Calidad del dibujo](#)
 - [Construir formas complejas con geométricos primitivos](#)
 - [Soportar Interacción del Usuario](#)
- [Trabajar con Texto y Fuentes](#)
 - [Crear y Derivar Fuentes](#)
 - [Dibujar múltiples líneas de texto](#)
- [Manipular y Mostrar Imágenes](#)
 - [Modo Inmediato con BufferedImage](#)
 - [Filtrar un BufferedImage](#)

- [Usar un BufferedImage para doble buffer](#)
- [Imprimir](#)
 - [Imprimir en Java](#)
 - [Imprimir el contenido de un componente](#)
 - [Mostrar el diálogo Page Setup](#)
 - [Imprimir una colección de páginas](#)
- [Resolver problemas comunes con los gráficos en 2D](#)

Trabajo en Red

- [Trabajo en la Red](#)
- Introducción al Trabajo en Red
 - [Trabajo en Red Básico](#)
 - [Lo que ya podrías conocer sobre el trabajo en Red](#)
- Trabajar con URLs
 - [¿Qué es una URL?](#)
 - [Crear una URL](#)
 - [Analizar una URL](#)
 - [Leer desde una URL](#)
 - [Conectar con una URL](#)
 - [Leer y Escribir utilizando una URL](#)
- [Todo sobre los Sockets](#)
 - [¿Qué es un Socket?](#)
 - [Leer y Escribir utilizando un Socket](#)
 - [Escribir el lado del servidor de un Socket](#)
- Todo sobre los Datagramas
 - [¿Qué es un Datagrama?](#)
 - [Escribir Datagramas Cliente y Servidor](#)
- [Controlador de Seguridad](#)
 - [Introducción a los Controladores de Seguridad](#)
 - [Escribir un Controlador](#)
 - [Instalar un Controlador](#)
 - [Decidir los Métodos a sobrescribir del SecurityManager](#)
- [Notas sobre el JDK 1.1](#)

Los Beans

- [JavaBeans](#)
- Introducción a los Beans
 - [Conceptos básicos](#)
 - [El Kit de Desarrollo de beans](#)
- [Utilizar BeanBox](#)
 - [Arrancar y utilizar BeanBox](#)
 - [Menús de BeanBox](#)
 - [Utilizar BeanBox para generar Applets](#)
- [Escribir un Bean sencillo](#)
- [Propiedades](#)
 - [Propiedades sencillas](#)
 - [Propiedades Compartidas](#)
 - [Propiedades Restringidas](#)
 - [Propiedades Indexadas](#)
- [Manipular Eventos en BeanBox](#)
- [El Interface BeanInfo](#)
- [Personalizar Beans](#)
- [Persistencia de un Bean](#)
- [Nuevas Caracterísitcas](#)

Servlets

- [Introducción a los Servlets](#)
 - [Arquitectura del paquete Servlet](#)
 - [Un Servlet Sencillo](#)
 - [Ejemplos](#)
- [Interacción con los Clientes](#)
 - [Peticiones y Respuestas](#)
 - [Manejar Peticiones GET y POST](#)
 - [Problemas con los Threads](#)
 - [Proporcionar Información de un Servlet](#)
- [El Ciclo de Vida de un Servlet](#)
 - [Inicializar un Servlet](#)

- [Destruir un Servlet](#)
- [Guardar el estado del Cliente](#)
 - [Trayectoria de Sesión](#)
 - [Utilizar Cookies](#)
- [La utilidad ServletRunner](#)
 - [Propiedades de un Servlet](#)
 - [Arrancar Servletrunner](#)
- [Ejecutar Servlets](#)
 - [Desde un Navegador](#)
 - [Desde una Página HTML](#)
 - [Desde otro Servlet](#)

Internacionalización

- [Mercados Globales](#)
 - [Internacionalización](#)
 - [Localización](#)
 - [Datos Dependientes de la Cultura](#)
- [Un Ejemplo Rápido](#)
 - [Antes de la Internacionalización](#)
 - [Después de la Internacionalización](#)
 - [Ejecutar el programa](#)
 - [Internacionalizar el ejemplo](#)
 - [Crear el fichero de propiedades](#)
 - [Definir la Localidad](#)
 - [Crear el ResourceBundle](#)
 - [Buscar el texto en el ResourceBundle](#)
- [Seleccionar la Localidad](#)
 - [Crear una Localidad](#)
 - [Identificar las Localidades disponibles](#)
 - [La Localidad por defecto](#)
 - [El ámbito de la Localidad](#)
- [Aislar Objetos Específicos en un ResourceBundle](#)
 - [La clase ResourceBundle](#)

- [Preparar un ResourceBundle](#)
 - [Ficheros de Propiedades](#)
 - [Utilizar un ListResourceBundle](#)
- [Formatear Números y Moneda](#)
 - [Usar Formatos Predefinidos](#)
 - [Formatear con Patrones](#)
- [Formatear Fechas y Horas](#)
 - [Usar Formatos Predefinidos](#)
 - [Formatear con Patrones](#)
 - [Cambiar simbolos en el formato de Fechas](#)
- [Formatear Mensajes](#)
 - [Tratar con mensajes concatenados](#)
 - [Manejar Plurales](#)
- [Trabajar con Excepciones](#)
 - [Manejar mensajes de las Excepciones](#)
 - [Crear subclases de Exception independientes de la Localidad](#)
- [Comparar Strings](#)
 - [Realizar comparaciones independientes de la Localidad](#)
 - [Reglas personales de comparación](#)
 - [Aumentar el rendimiento de la comparación](#)
- [Detectar Límites de Texto](#)
 - [La clase BreakIterator](#)
 - [Límite de Caracter](#)
 - [Límite de Palabra](#)
 - [Límite de Sentencia](#)
 - [Límite de Línea](#)
- [Convertir texto no Unicode](#)
 - [Bytes Codificados y Strings](#)
 - [Streams de Bytes y de Caracter](#)
- [Un lista de chequeo para Internacionalizar una programa existente](#)

Ficheros JAR

- [Formato de Fichero JAR](#)
- [Usar ficheros JAR: básico](#)
 - [Crear un fichero JAR](#)
 - [Ver el contenido de un fichero JAR](#)
 - [Extraer el contenido de un fichero JAR](#)
 - [Modificar un fichero de Manifiesto](#)
 - [Ejecutar software contenido en un fichero JAR](#)
 - [Entender el Manifiesto](#)
- [Firmar y Verificar ficheros JAR](#)
 - [Firmar un Fichero JAR](#)
 - [Verificar un Fichero JAR Firmado](#)
 - [Entender la Firma y la Verificación](#)

Métodos Nativos

- [El JNI de Java](#)
- [Paso a Paso](#)
 - [Paso 1: Escribir el código Java](#)
 - [Paso 2: Compilar el código Java](#)
 - [Paso 3: Crear el fichero .H](#)
 - [Paso 4: Escribir el Método Nativo](#)
 - [Paso 5: Crear una Librería Compartida](#)
 - [Paso 6: Ejecutar el Programa](#)
- [Implementar Métodos Nativos](#)
 - [Declarar Métodos Nativos](#)
 - [Los tipos Java en Métodos Nativos](#)
 - [Acceder a Strings Java en Métodos Nativos](#)
 - [Trabajar con Arrays Java en Métodos Nativos](#)
 - [Llamar a Métodos Java](#)
 - [Acceder a campos Java](#)
 - [Manejar errores Java desde Métodos Nativos](#)
 - [Referencias Locales y Globales](#)
 - [Threads y Métodos Nativos](#)

- [Invocar a la Máquina Virtual Java](#)
- [Programación JNI en C++](#)

Acceso a Bases de Datos: JDBC

- [Acceso a Bases de Datos](#)
- JDBC Básico
 - [Empezar](#)
 - [Seleccionar una base de datos](#)
 - [Establecer una Conexión](#)
 - [Seleccionar una Tabla](#)
 - [Recuperar Valores desde una Hoja de Resultados](#)
 - [Actualizar Tablas](#)
 - [Utilizar Sentencias Preparadas](#)
 - [Utilizar Uniones](#)
 - [Utilizar Transacciones](#)
 - [Procedimientos Almacenados](#)
 - [Utilizar Sentencias SQL](#)
 - [Crear Aplicaciones JDBC Completas](#)
 - [Ejecutar la Aplicación de Ejemplo](#)
 - [Crear un Applet desde una Aplicación](#)
- [El API del JDBC 2.0](#)
 - [Inicialización para utilizar JDBC 2.0](#)
 - [Mover el Cursor sobre una hoja de Resultados](#)
 - [Hacer Actualizaciones en una hoja de Resultados](#)
 - [Actualizar una Hoja de Resultados Programáticamente](#)
 - [Insertar y borrar filas Programáticamente](#)
 - [Insertar una fila](#)
 - [Borrar una fila](#)
 - [Hacer Actualizaciones por Lotes](#)
 - [Usar tipos de datos SQL3](#)
 - [Características de Extensión Estándar](#)

Invocación Remota de Métodos: RMI

- [RMI](#)
 - [Introducción al RMI](#)
 - [Escribir un Servidor de RMI](#)
 - [Diseñar un Interface Remoto](#)
 - [Implementar un Interface Remoto](#)
 - [Crear un Programa Cliente](#)
 - [Compilar y Ejecutar el Ejemplo](#)
 - [Compilar el Programa de Ejemplo](#)
 - [Ejecutar el Programa de Ejemplo](#)

Cambios en el JDK 1.1.x

- [Cambios en el JDK 1.1.x](#)
- [¿Qué hay de nuevo?](#)
 - [Internacionalización](#)
 - [Seguridad y los Applets firmados](#)
 - [Ampliación del AWT](#)
 - [JavaBeans](#)
 - [Ficheros JAR](#)
 - [Ampliación de la Red](#)
 - [Ampliación de la I/O](#)
 - [El paquete Math](#)
 - [RMI](#)
 - [Serializar Objetos](#)
 - [Reflexión](#)
 - [Bases de Datos](#)
 - [Clases Internas](#)
 - [Interface Nativo](#)
 - [Aumento de Rendimiento](#)
 - [Miscelánea](#)
 - [Notas sobre JDK 1.1](#)
- [Cambios en el GUI](#)
 - [Nuevo modelo de Eventos](#)

- [Introducción](#)
- [Adaptadores y clases Internas](#)
- [Eventos Estandar](#)
- [Eventos generados por componentes del AWT](#)
- [Escribir un oyente de Action](#)
- [Escribir un oyente de Adjustment](#)
- [Escribir un oyente de Component](#)
- [Escribir un oyente de Container](#)
- [Escribir un oyente de Focus](#)
- [Escribir un oyente de Item](#)
- [Escribir un oyente de Key](#)
- [Escribir un oyente de Ratón](#)
- [Escribir un oyente de movimiento de Ratón](#)
- [Escribir un oyente de Texto](#)
- [Escribir un oyente de Ventanas](#)
- [Utilizar la versión "Swing" del JFC](#)
 - [Introducción al Swing](#)
 - [Empezar con Swing](#)
 - [Ejecutar un Applet de Swing](#)
 - [Detalles de Componentes](#)
 - [Reglas Generales](#)
 - [La clase Button](#)
 - [La clase Checkbox](#)
 - [La clase Label](#)
 - [La clase RadioButton](#)
 - [La clase TabbedPane](#)
 - [La clase Table](#)
 - [La clase ToolTip](#)
 - [La clase Tree](#)

Tu Primera 'Taza' de Java

Las dos páginas siguientes proporcionan instrucciones detalladas para compilar y ejecutar tu primer programa Java. Elige la sección a la que pertenezca tu sistema operativo

[Tu primera 'Taza' en Win32](#)

Estas instrucciones son para usuarios de plataformas Win32, que incluye a Windows 95, Windows 98 y Windows NT.

[Tu primera 'Taza' en UNIX](#)

Estas instrucciones son para usuarios de plataformas basadas en UNIX, incluyendo Linux y Solaris.

[Tu primera 'Taza' en MAC](#)

Estas instrucciones son para usuarios de plataformas basadas en MAC.

Tu Primera Taza de Java en Win32



Instrucciones Detalladas para Tu Primer Programa

Las siguientes instrucciones te ayudarán a escribir tu primer programa Java. Estas instrucciones son para usuarios de plataformas Win32, que incluye Windows 95, Windows 98 y Windows NT.

1. [Checklist](#)
 2. [Crear tu Primera Aplicación](#)
 - a. [Crear un Fichero Fuente Java](#)
 - b. [Compilar el Fichero Fuente](#)
 - c. [Ejecutar el Programa](#)
 3. [Crear Tu Primer Applet](#)
 4. [Dónde ir desde Aquí](#)
-

1. Checklist

Para escribir tu primer programa, necesitarás:

1. La Edición Estándar de la Plataforma Java 2™. Puedes [descargarla ahora](#) y consultar las [instrucciones de instalación](#).
2. Un Editor de texto. En este ejemplo, usaremos el NotePad de Windows. Para encontrar NotePad desde le menú **Inicio** selecciona **Programas** > **Accesorios** > **NotePad**. Si usas un editor diferente no te será difícil adaptar estas instrucciones.

Estas dos cosas son todo lo que necesitas para programar en Java.

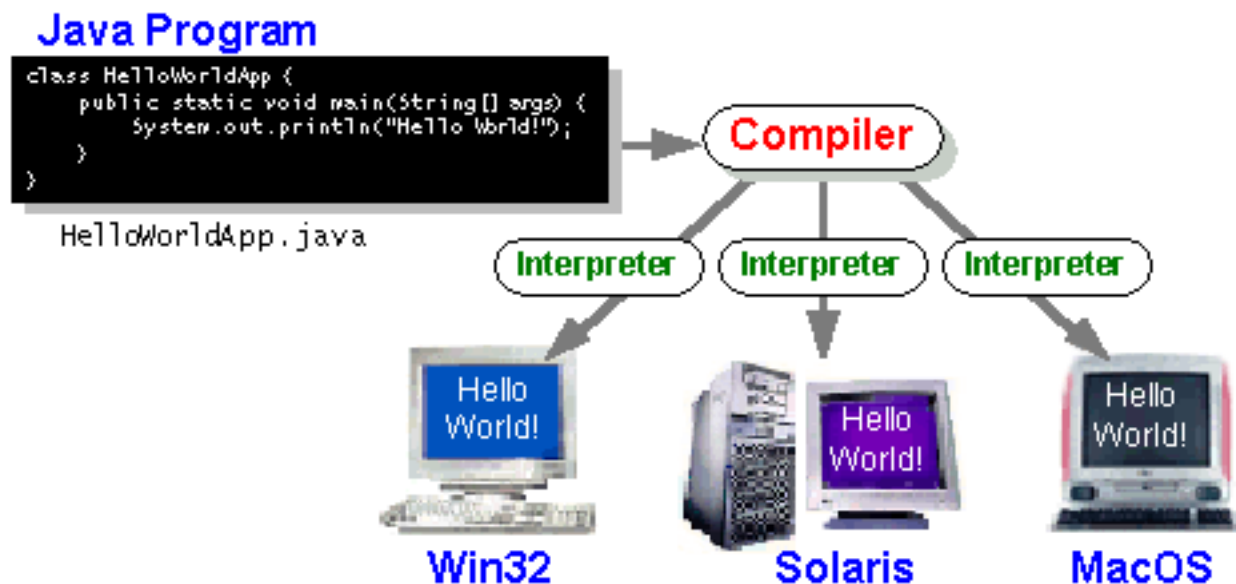
2. Crear Tu Primera Aplicación

Tu primer programa, HelloWorldApp, simplemente mostrará el saludo "Hello world!". Para crear este programa, deberás:

- Crear un **fichero fuente** Java. Un fichero fuente contiene texto, escrito en el lenguaje de programación Java, que tu y otros programadores pueden entender. Se puede usar cualquier editor de texto para crear y editar ficheros fuente.
- Compilar el fichero fuente en un **fichero de bytecodes**. El compilador de Java, javac, toma nuestro fichero fuente y lo traduce en instrucciones que la Máquina Virtual Java (Java VM) puede entender. El compilador pone estas instrucciones en un fichero de bytecodes.
- Ejecutar le programa contenido en el fichero de bytecodes. La máquina virtual Java está implementada por un intérprete Java, java. Este intérprete toma nuestro fichero de bytecodes y lleva a cabo las instrucciones traduciéndolas a instrucciones que nuestro ordenador puede entender.

¿Por qué están de moda los Bytecodes

Habrás oído que con el lenguaje de programación Java, puedes "escribir una vez, ejecutar en cualquier parte". Esto significa que cuando se compila un programa, no se generan instrucciones para una plataforma específica. En su lugar, se generan bytecodes Java, que son instrucciones para la Máquina Virtual Java (Java VM). Si tu plataforma- sea Windows, UNIX, MacOS o un navegador de internet-- tiene la Java VM, podrá entender los bytecodes.



[Subir](#)

a. Crear un Fichero Fuente Java.

Tienes dos opciones:

- Puedes grabar el fichero [HelloWorldApp.java](#) en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).

- O, puedes seguir estas (largas) instrucciones:


1. Arranca NotePad. En un nuevo documento, teclea el siguiente código:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Se cuidadoso cuando lo teclees



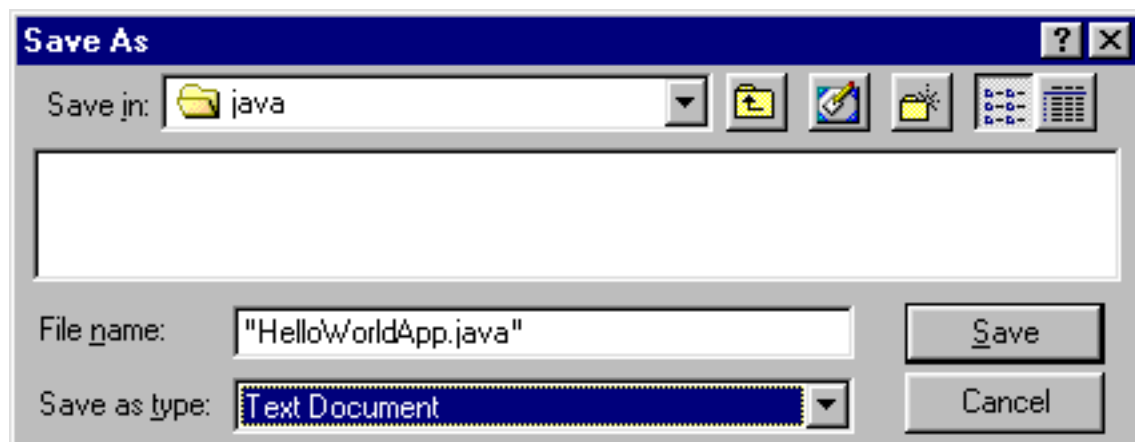
Teclea todo el código, comandos y nombres de ficheros exactamente como los ves. El compilador y el intérprete Java son sensibles a las mayúsculas.

HelloWorldApp  helloworldapp

2. Graba este código en un fichero. Desde la barra de menú, selecciona **File > Save As**. En la caja de diálogo **Save As**:

- Usa el menú desplegable **Save in**, para especificar el directorio (o carpeta) donde grabarás el fichero. En este ejemplo, es el directorio java en la unidad C.
- En la caja de texto **File name**, teclea, "HelloWorldApp.java", incluyendo las comillas.
- En el menú desplegable **Save as type**, elige **Text Document**.

Cuando hayas terminado la caja de diálogo se debería parecer a esto:

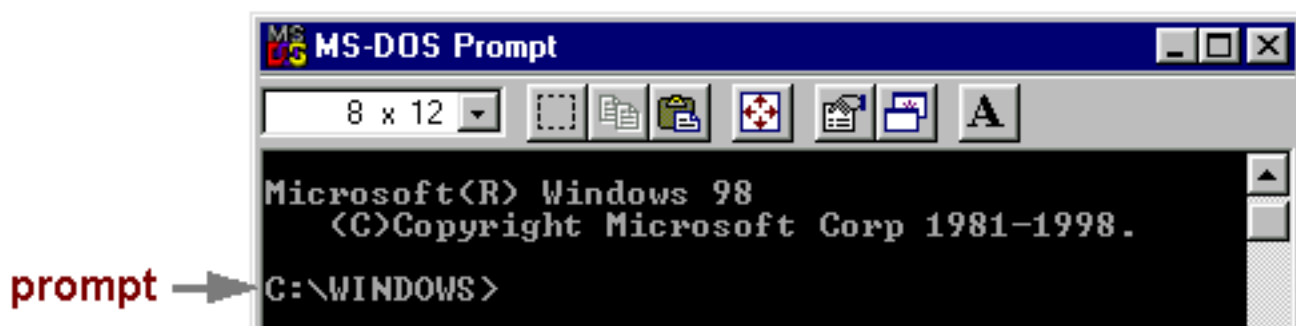


Ahora pulsa sobre **Save**, y sal de NotePad.

 [Subir](#)

b. Compilar el Fichero Fuente

Desde el menú **Start**, selecciona la aplicación **MS-DOS Prompt** (Windows 95/98) o **Command Prompt** (Windows NT). Cuando se lance, se debería parecer a esto:



El prompt muestra tu directorio actual. Cuando salimos a una ventana del DOS en Windows 95/98, el directorio actual normalmente es WINDOWS en nuestra unidad C (como se ve arriba) o WINNT para Windows NT. Para compilar el fichero fuente, cambiamos al directorio en el que se encuentra el fichero. Por ejemplo, si nuestro directorio de código fuente es java en la unidad C, deberíamos teclear el siguiente comando y pulsar **Enter**:

```
cd c:\java
```

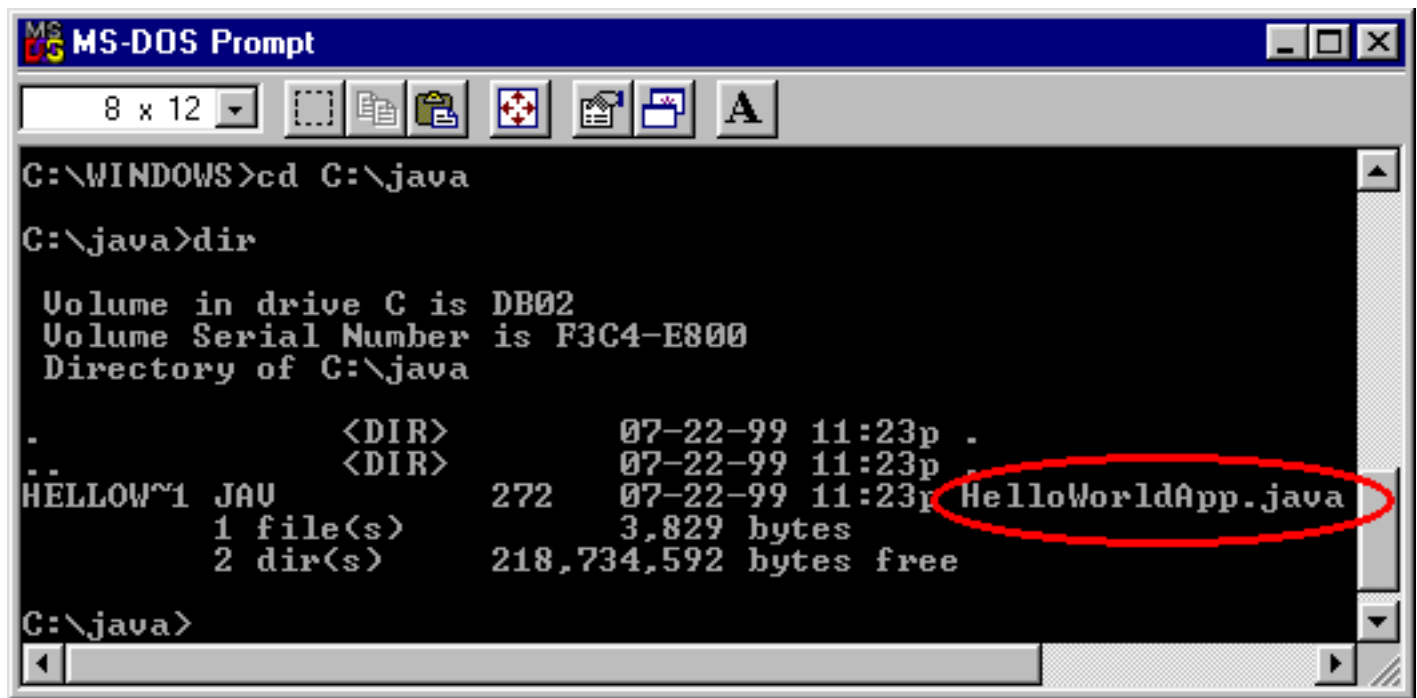
Ahora el prompt debería cambiar a C:\java>.

Nota: Para cambiar a un directorio en una unidad diferente, deberemos teclear un comando extra.

```
C:\WINDOWS>cd d:\java
C:\WINDOWS>d:
D:\java>
```

Como se ve aquí, para cambiar al directorio java en la unidad D, debemos reentrar en la unidad d:

Si tecleas dir en la línea de comandos, deberías ver tu fichero.



```
MS-DOS Prompt
8 x 12
C:\WINDOWS>cd C:\java
C:\java>dir

Volume in drive C is DB02
Volume Serial Number is F3C4-E800
Directory of C:\java

.                <DIR>                07-22-99 11:23p .
..               <DIR>                07-22-99 11:23p ..
HELLOW~1 JAV    272      07-22-99 11:23p HelloWorldApp.java
1 file(s)        3,829 bytes
2 dir(s)        218,734,592 bytes free

C:\java>
```

Ahora puedes compilar. En la línea de comandos, teclea el siguiente comando y pulsa **Enter**:

```
javac HelloWorldApp.java
```

Si el prompt reaparece sin mensajes de error, felicidades. Tu programa se ha compilado con éxito.

Explicación de Error

Bad command or file name (Windows 95/98)

El nombre especificado no es reconocido como un comando interno o externo, operable program or batch file (Windows NT)

Si recibes este error, Windows no puede encontrar el compilador Java, javac.

Aquí hay una forma de decirle a Windows dónde encontrar javac.

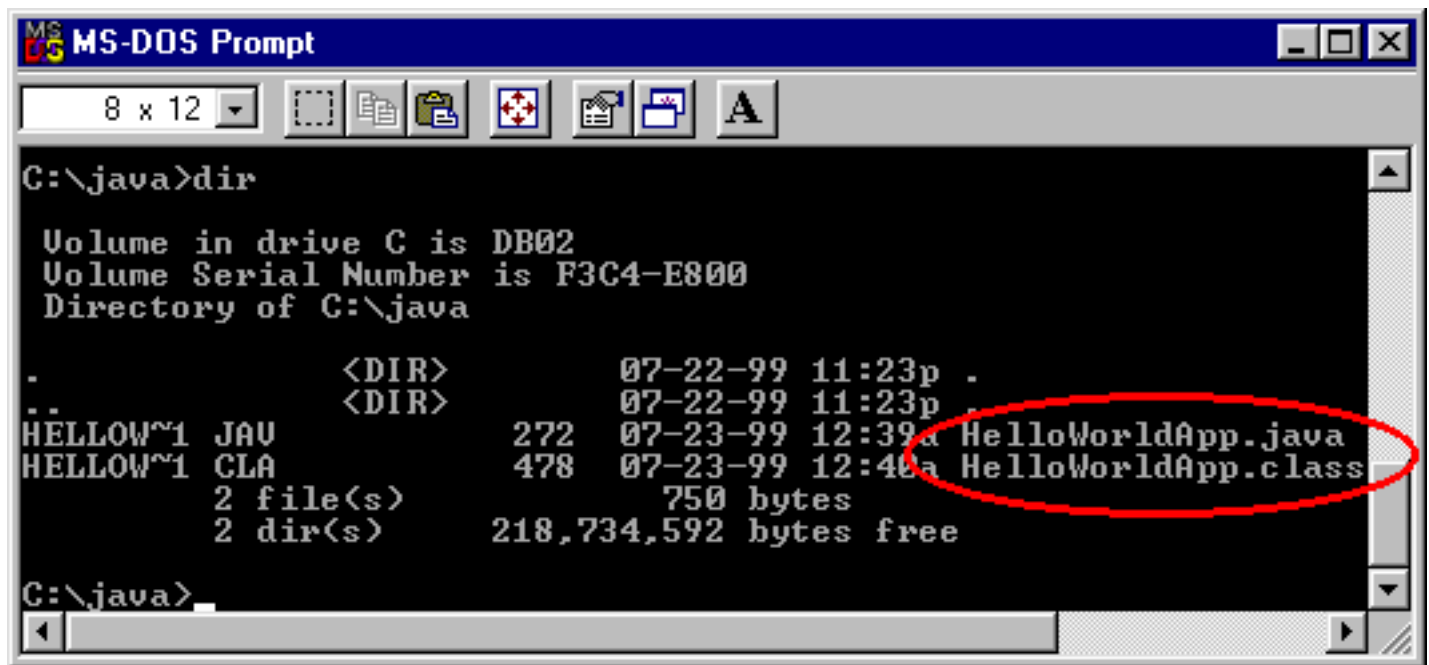
Supongamos que has instalado el SDK de Java 2 en C:\jdk1.2.2. En el prompt deberías teclear el siguiente comando y pulsar Enter:

```
C:\jdk1.2.2\bin>javac HelloWorldApp.java
```

Nota: Si eliges esta opción, cada vez que compiles o ejecutes un programam tendrás que preceder a tus comandos javac y java con C:\jdk1.2.2\bin\.

Para evitar esto consulta la sección [Update the PATH variable](#) en las instrucciones de instalación.

El compilador ha generado un fichero de bytecodes Java, HelloWorldApp.class. En el prompt, teclea dir para ver el nuevo fichero que se ha generado:



```
MS-DOS Prompt
8 x 12
C:\java>dir

Volume in drive C is DB02
Volume Serial Number is F3C4-E800
Directory of C:\java

.                <DIR>                07-22-99  11:23p  .
..               <DIR>                07-22-99  11:23p  ..
HELLOW~1 JAV      272    07-23-99  12:39a  HelloWorldApp.java
HELLOW~1 CLA      478    07-23-99  12:40a  HelloWorldApp.class
2 file(s)                750 bytes
2 dir(s)                218,734,592 bytes free

C:\java>
```

Ahora que tienen un fichero .class, puedes ejecutar tu programa.

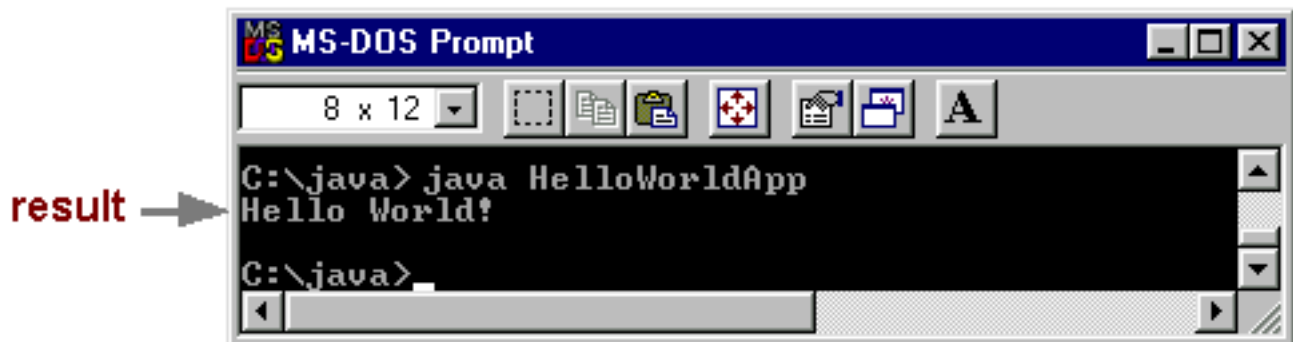
[↑subir](#)

c.Ejecutar el Programa

En el mismo directorio teclea en el prompt:

```
java HelloWorldApp
```

Ahora deberías ver:



```
MS-DOS Prompt
8 x 12
C:\java>java HelloWorldApp
Hello World!

C:\java>
```

result →

Felicidades!, tu programa funciona.

Explicación de Error

Exception in thread "main" java.lang.NoClassDefFoundError:
HelloWorldApp

Si recibes este error, java no puede encontrar tu fichero de bytecodes, HelloWorldApp.class.

Uno de los lugares donde java intenta buscar el fichero de bytecodes es el directorio actual. Por eso, si tu fichero de bytecodes está en C:\java, deberías cambiar a ese directorio como directorio actual.

Si todavía tienes problemas, podrías tener que cambiar tu variables CLASSPATH. Para ver si es necesario, intenta seleccionar el casspath con el siguiente comando:

```
set CLASSPATH=
```

Ahora introduce de nuevo java HelloWorldApp. Si el programa funciona, tendrás que cambiar tu variable CLASSPATH. Para más información, consulta la sección [Check the CLASSPATH Variable](#) en las instrucciones de instalación.

[↑subir](#)

3. Crear tu Primer Applet

HelloWorldApp es un ejemplo de una aplicación Java, un programa solitario. Ahora crearás un applet Java, llamado HelloWorld, que también muestra el salido "Hello world!". Sin embargo, al contrario que HelloWorldApp, el applet se ejecuta sobre un navegador compatible con Java, como HotJava, Netscape Navigator, o Microsoft Internet Explorer.

Para crear este applet, debes realizar los mismos pasos básicos que antes: crear un fichero fuente Java; compilarlo y ejecutar el programa.

a. Crear un Fichero Fuente Java.

De nuevo, tienes dos opciones:

- Puedes grabar los ficheros [HelloWorld.java](#) y [Hello.html](#) en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).
- O, puedes seguir estas instrucciones:

1. Arranca NotePad y teclea el siguiente código en un nuevo documento:


```
import java.applet.*;
import java.awt.*;

/**
 * The HelloWorld class implements an applet that
 * simply displays "Hello World!".
 */
public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Graba este código en un fichero llamado HelloWorld.java.

2. También necesitas un fichero HTML que acompañe a tu applet. Teclea el siguiente código en nuevo documento del NotePad:

```
<HTML>
<HEAD>
<TITLE>A Simple Program</TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Graba este código en un fichero llamado Hello.html.

b. Compilar el Código Fuente.

En el prompt, teclea el siguiente comando y pulsa **Return**:

```
javac HelloWorld.java
```

El compilar debería generar el fichero de bytecodes Java, HelloWorld.class.

c. Ejecutar el Programa.

Aunque puedes ver tu applet usando un navegador, podrías encontrar más sencillo probarlos usando la aplicación appletviewer que viene con la plataforma Java™. Para ver el applet HelloWorld usando el appletviewer, teclea esto en la línea de comandos:

```
appletviewer Hello.html
```



Ahora deberías ver:

Felicidades! tu applet funciona.

[↑subir](#)

4. ¿A dónde ir desde aquí?

Para continuar con la introducción al lenguaje Java, puedes visitar estas secciones:

[Por dónde Empezar](#)

[Conceptos Básicos](#)

[↑Subir](#)

[Ozito](#)

Primera Taza de Java en UNIX



Instrucciones Detalladas para Tu Primer Programa

Las siguientes instrucciones te ayudarán a escribir tu primer programa Java. Estas instrucciones son para usuarios de plataformas basadas en UNIX, incluyendo Linux y Solaris

1. [Checklist](#)

2. [Crear tu Primera Aplicación](#)

- a. [Crear un Fichero Fuente Java](#)
- b. [Compilar el Fichero Fuente](#)
- c. [Ejecutar el Programa](#)

3. [Crear Tu Primer Applet](#)

4. [Dónde ir desde Aquí](#)

1. Checklist

Para escribir tu primer programa, necesitarás:

1. La Edición Estándar de la Plataforma Java 2™. Puedes [descargarla ahora](#) y consultar las [instrucciones de instalación](#).
2. Un Editor de texto. En este ejemplo, usaremos el Pico, un editor disponible en muchas plataformas basadas en UNIX. Si usas un editor diferente, como VI o Emacs, no te será difícil adaptar estas instrucciones.

Estas dos cosas son todo lo que necesitas para programar en Java.

2. Crear Tu Primera Aplicación

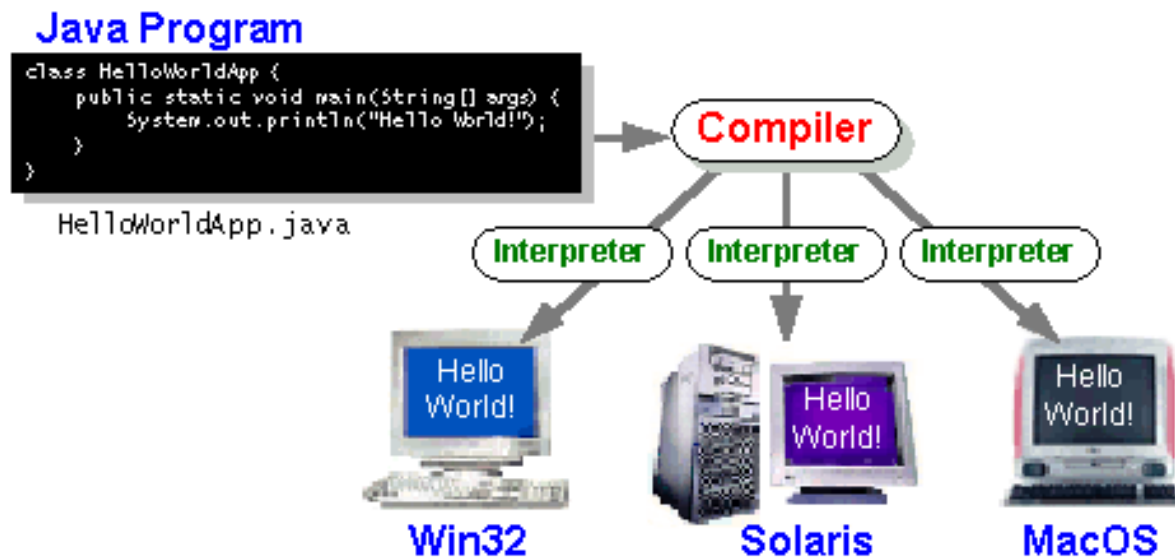
Tu primer programa, HelloWorldApp, simplemente mostrará el saludo "Hello world!". Para crear este programa, deberás:

- Crear un **fichero fuente** Java. Un fichero fuente contiene texto, escrito en el lenguaje de programación Java, que tu y otros programadores pueden entender. Se puede usar cualquier editor de texto para crear y editar ficheros fuente.
- Compilar el fichero fuente en un **fichero de bytecodes**. El compilador de Java, javac, toma nuestro fichero fuente y lo traduce en instrucciones que la Máquina Virtual Java (Java VM) puede entender. El compilador pone estas instrucciones en un fichero de bytecodes.

¿Por qué están de moda los Bytecodes

Habrás oído que con el lenguaje de programación Java, puedes "escribir una vez, ejecutar en cualquier parte". Esto significa que cuando se compila un programa, no se generan instrucciones para una plataforma específica. En su lugar, se generan bytecodes Java, que son instrucciones para la Máquina Virtual Java (Java VM). Si tu plataforma- sea Windows, UNIX, MacOS o un navegador de internet-- tiene la Java VM, podrá entender los bytecodes.

- Ejecutar le programa contenido en el fichero de bytecodes. La máquina virtual Java está implementada por un intérprete Java, java. Este intérprete toma nuestro fichero de bytecodes y lleva a cabo las instrucciones traduciéndolas a instrucciones que nuestro ordenador puede entender.



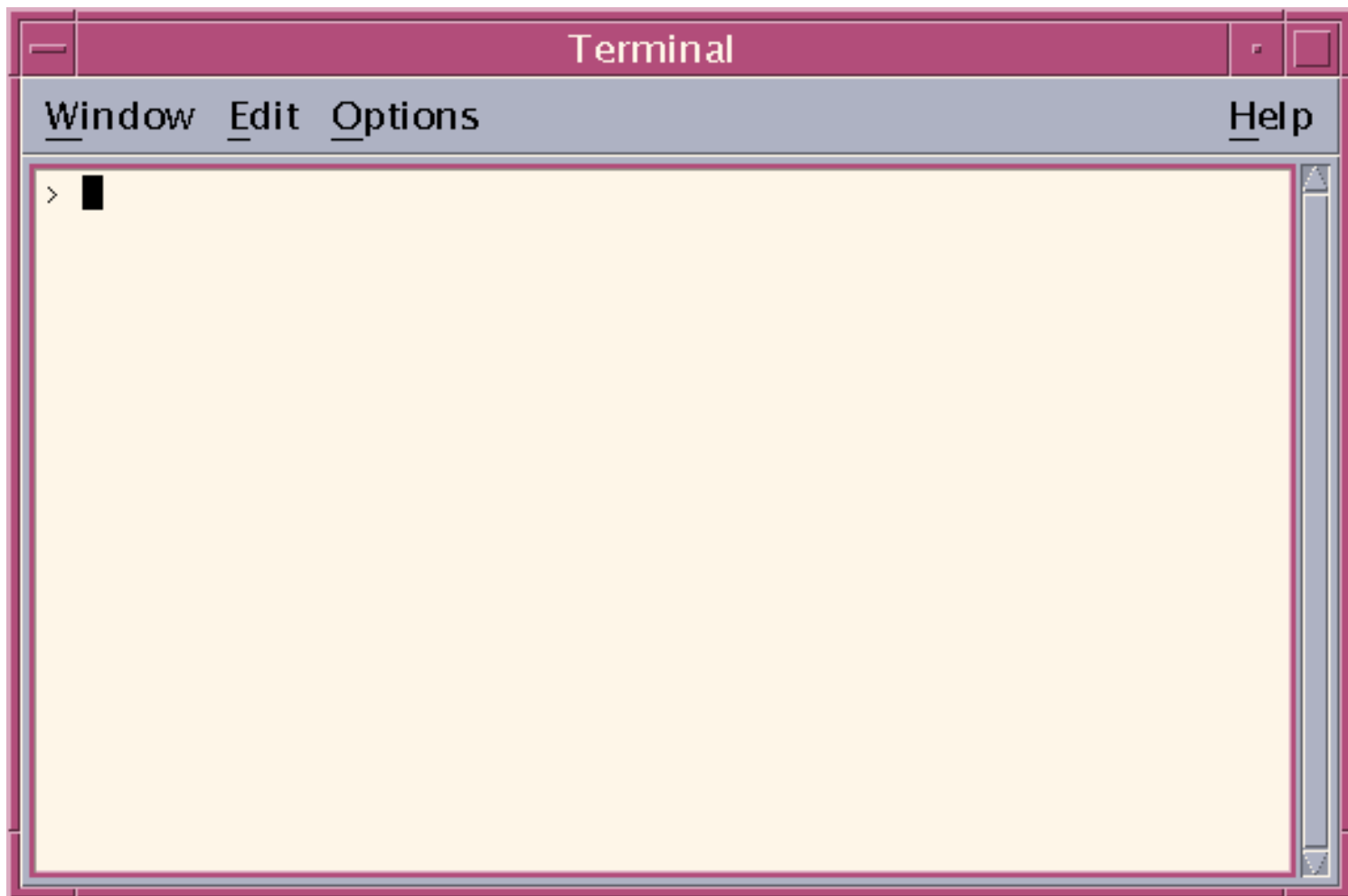
[↑ Subir](#)

a. Crear un Fichero Fuente Java.

Tienes dos opciones:

- Puedes grabar el fichero HelloWorldApp.java en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).
- O, puedes seguir estás (largas) instrucciones:

1. Trae una ventana shell (algunas veces llamada una ventana de terminal). Cuando veas la ventana se parecerá a esto



Cuando se muestra por primera vez el prompt, tu directorio actual será normalmente tu directorio 'home'. Puedes cambiar tu directorio actual a tu directorio home en cualquier momento tecleando `cd` y pulsando **Return**.

Los ficheros Java que crees deberían estar en un directorio separado. Puedes crear un directorio usando el comando `mkdir`. Por ejemplo, para crear el directorio `java` en tu directorio home, primero debes cambiar tu directorio actual a tu directorio home entrando el siguiente comando:

```
cd
```

Luego introducirás el siguiente comando:

```
mkdir java
```

Para cambiar tu directorio actual a este nuevo directorio, deberías teclear:

```
cd java
```

Ahora ya puedes empezar a crear tu fichero fuente.

2. Arranca el editor Pico tecleando pico en el prompt y pulsando **Return**. Si el sistema responde con el mensaje pico: command not found, es que Pico no está disponible. Consulta a tu administrador de sistemas para más información o usa otro editor.

Cuando se arranca Pico, se muestra un nuevo buffer en blanco. Este es el área en que el teclearás tu código

Pico? VI? Emacs?

Pico es probablemente el más sencillo de los tres editores. Si tienes curiosidad sobre el uso de los otros editores puedes visitar las siguientes páginas [VI](#) y [Emacs](#) .


3. Teclea el siguiente código dentro del nuevo buffer:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Se cuidadoso cuando lo teclees



Teclea todo el código, comandos y nombres de ficheros exactamente como los ves. El compilador y el intérprete Java son sensibles a las mayúsculas.

HelloWorldApp  helloworldapp

4. Graba el código pulsando **Ctrl-O**. En la parte inferior verás el prompt File Name to write:. Introduce HelloWorldApp.java, precedido por el directorio en el que deseas grabar el fichero. Por ejemplo, si lo deseas grabar en el directorio /home/rortigas/java, deberías teclear /home/rortigas/java/HelloWorldApp.java y pulsar **Return**.

Puedes teclear **Ctrl-X** para salir de Pico. [↑ subir](#)

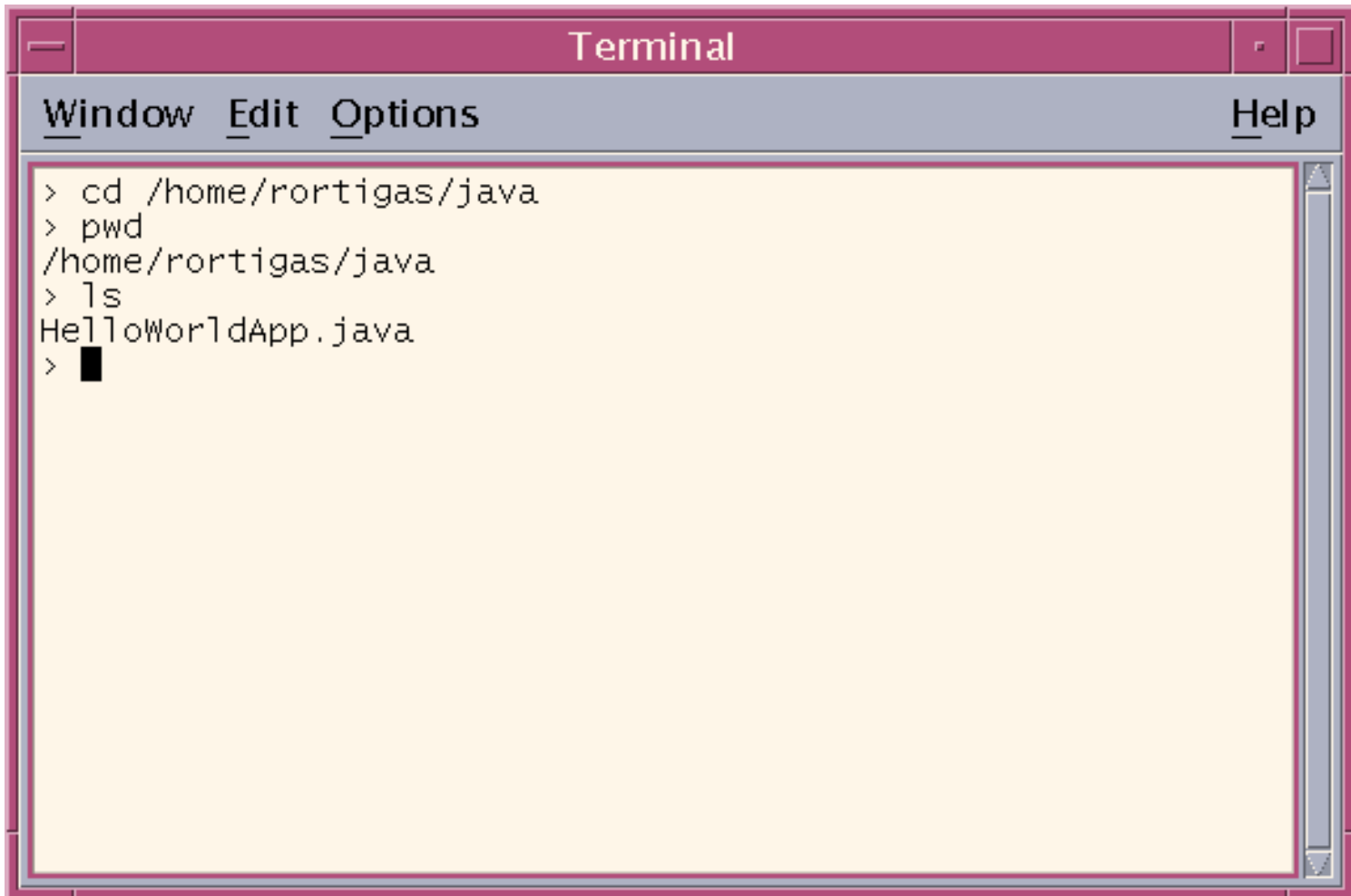
b. Compilar el fichero Fuente.

Trae otra ventana del shell. Para compilar el fichero fuente, cambiar tu directorio actual al directorio en el que estaba el fichero fuente. Por ejemplo, si tu directorio fuente es /home/rortigas/java, deberías teclear el siguiente comando en el prompt y pulsar **Return**:

```
cd /home/rortigas/java
```

Si introduces `pwd` en el prompt deberías ver el directorio actual, que en este ejemplo ha sido cambiado a `/home/rortigas/java`.

Si introduces `ls` en el prompt deberías ver tu fichero.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "Window", "Edit", "Options", and "Help". The terminal content shows a series of commands and their outputs: a prompt followed by "cd /home/rortigas/java", another prompt followed by "pwd" and the output "/home/rortigas/java", a third prompt followed by "ls" and the output "HelloWorldApp.java", and finally a fourth prompt with a cursor. The terminal has a yellow background and a scroll bar on the right.

```
> cd /home/rortigas/java
> pwd
/home/rortigas/java
> ls
HelloWorldApp.java
> █
```

Ahora puedes compilarlo. En el prompt, teclea el siguiente comando y pulsa **Return**:

```
javac HelloWorldApp.java
```

Si tu prompt reaparece sin mensajes de error, Felicidades! tu programa se ha compilado con éxito.

Explicación de Error

javac: Command not found

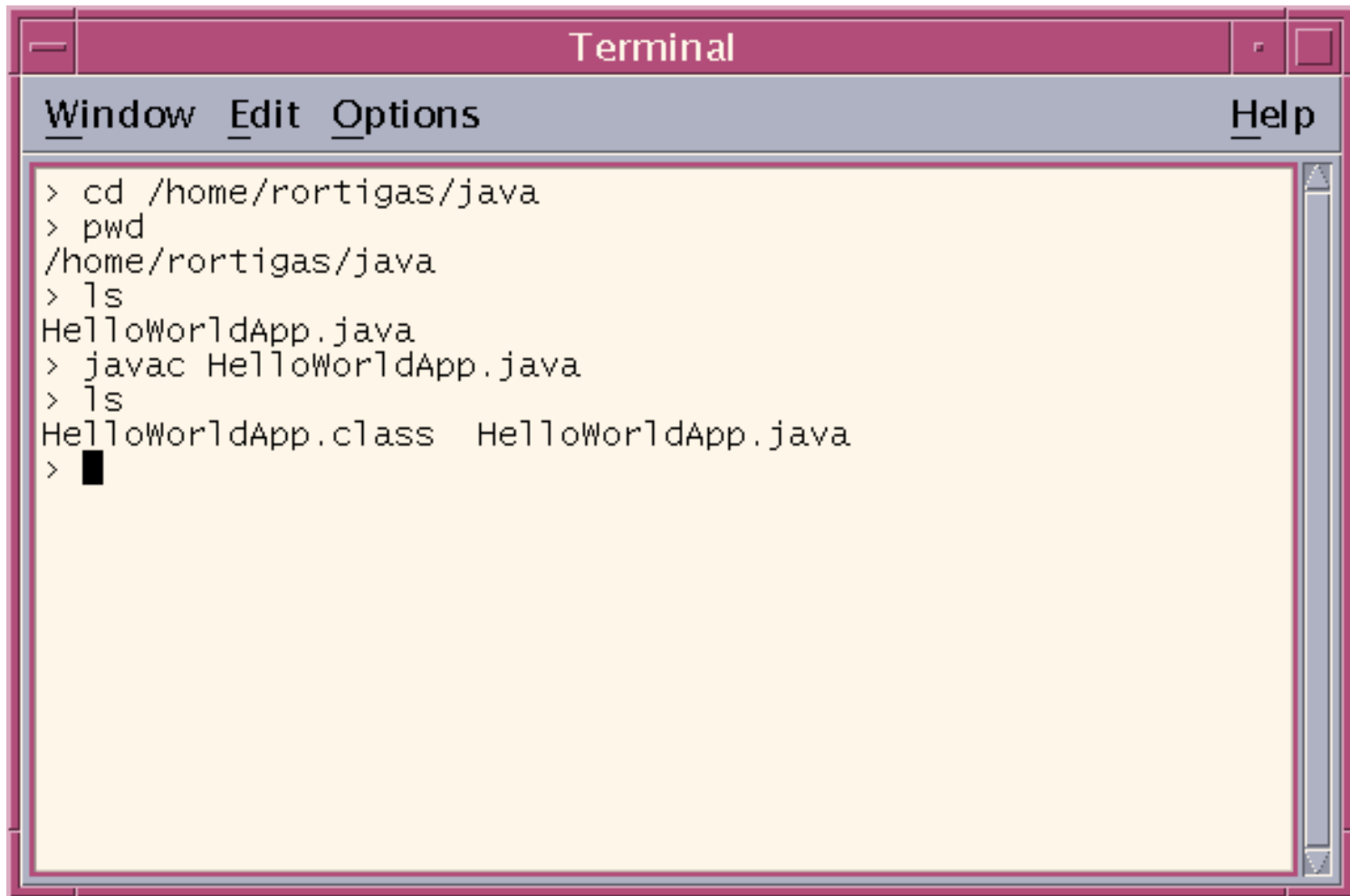
Si recibes este error, UNIX no puede encontrar el compilador Java, javac.

Aquí hay una forma de decirle a UNIX dónde encontrar javac. Supongamos que has instalado el SDK de Java 2 en /usr/local/jdk1.2.2. En el prompt deberías teclear el siguiente comando y pulsar Enter:

```
/usr/local/jdk1.2.2\bin\javac HelloWorldApp.java
```

Nota: Si eliges esta opción, cada vez que compiles o ejecutes un programam tendrás que preceder a tus comandos javac y java con /usr/loacl/jdk1.2.2\bin\. Para evitar esto consulta la sección [Update the PATH variable](#) en las instrucciones de instalación.

El compilador ha generado un fichero de bytecodes de Java, HelloWorldApp.class. En el prompt, teclea ls para ver el nuevo fichero generado



```
Terminal
Window Edit Options Help
> cd /home/rortigas/java
> pwd
/home/rortigas/java
> ls
HelloWorldApp.java
> javac HelloWorldApp.java
> ls
HelloWorldApp.class HelloWorldApp.java
> █
```

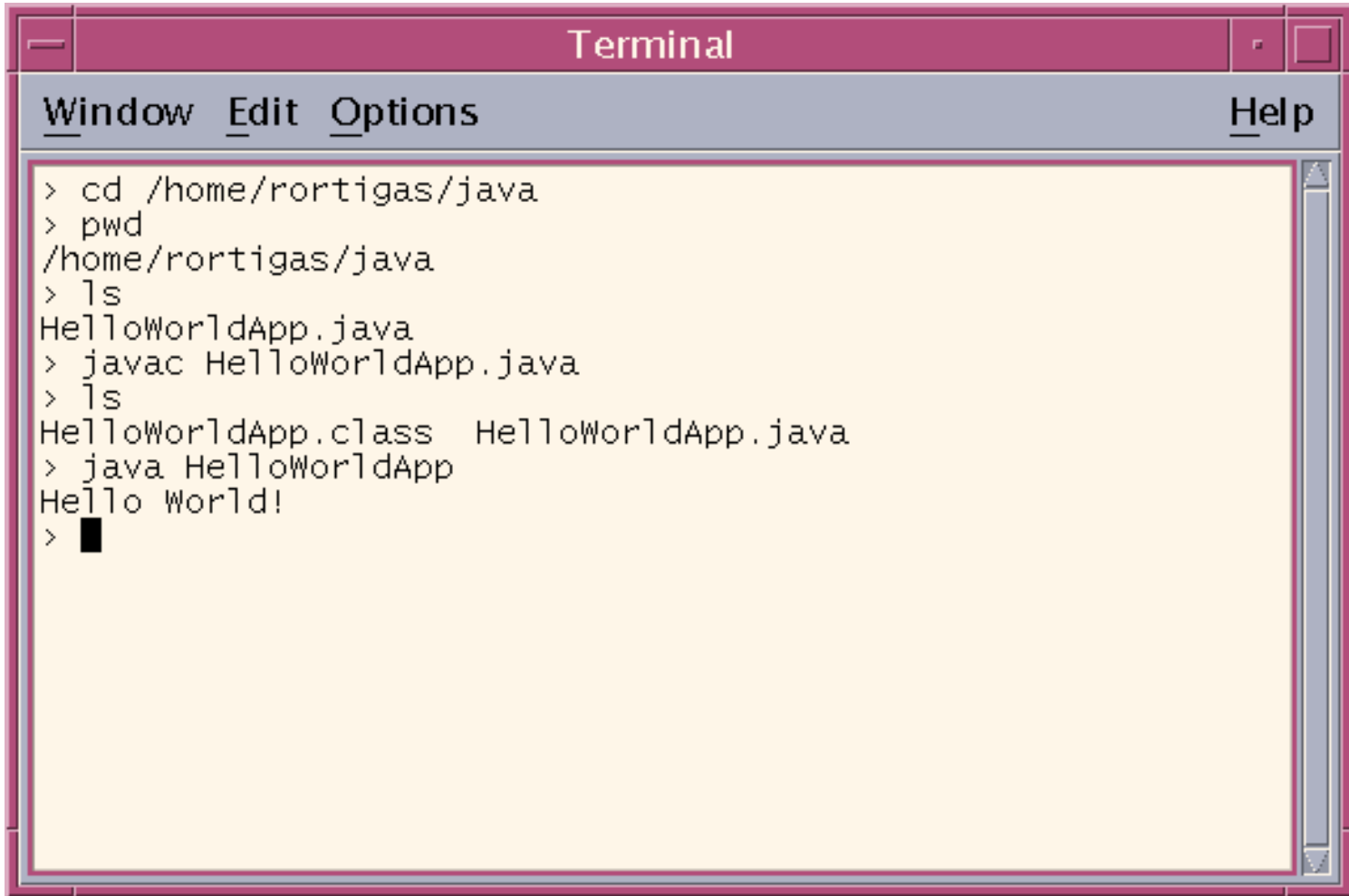
Ahora que tenemos un fichero .class, podemos ejecutar el programa.

c.Ejecutar el Programa.

En el mismo directorio introducir en el prompt:

```
java HelloWorldApp
```

Ahora deberías ver:

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "Window", "Edit", "Options", and "Help". The terminal content shows a series of commands and their outputs:

```
> cd /home/rortigas/java
> pwd
/home/rortigas/java
> ls
HelloWorldApp.java
> javac HelloWorldApp.java
> ls
HelloWorldApp.class HelloWorldApp.java
> java HelloWorldApp
Hello World!
> █
```

Felicidades! tu programa funciona!.

Explicación de Error

Exception in thread "main" java.lang.NoClassDefFoundError:
HelloWorldApp

Si recibes este error, java no puede encontrar tu fichero de bytecodes,
HelloWorldApp.class.

Uno de los lugares donde java intenta buscar el fichero de bytecodes es el
directorio actual. Por eso, si tu fichero de bytecodes está en
/home/rortigas/java, deberías cambiar a ese directorio como directorio
actual.

```
cd /home/rortigas/java
```

Si todavía tienes problemas, podrías tener que cambiar tu variable
CLASSPATH. Para ver si es necesario, intenta seleccionar el casspath con el

siguiente comando:

```
unset CLASSPATH=
```

Ahora introduce de nuevo java HelloWorldApp. Si el programa funciona, tendrás que cambiar tu variable CLASSPATH. Para más información, consulta la sección [Check the CLASSPATH Variable](#) en las instrucciones de instalación.

[↑ Subir](#)

3. Crear tu primer Applet

HelloWorldApp es un ejemplo de una aplicación Java, un programa solitario. Ahora crearás tu primer Applet Ajava llamado HelloWorld, que también muestra el saludo "Hello world!". Sin embargo, al contrario que HelloWorldApp, el applet se ejecuta sobre un navegador compatible con Java, como HotJava, Netscape Navigator, o Microsoft Internet Explorer.

Para crear este applet, debes realizar los mismos pasos básicos que antes: crear un fichero fuente Java; compilarlo y ejecutar el programa.

a. Crear un Fichero Fuente Java.

De nuevo, tienes dos opciones:

- Puedes grabar los ficheros [HelloWorld.java](#) y [Hello.html](#) en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).
- O, puedes seguir estas instrucciones:

1. Arrancar Pico, y teclear el siguiente código dentro del buffer:

```
import java.applet.*;
import java.awt.*;

/**
 * The HelloWorld class implements an applet that
 * simply displays "Hello World!".
 */
public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Graba este código en un fichero llamado HelloWorld.java. Teclea **Ctrl-X**

para salir de Pico.

2. También necesitas un fichero HTML que acompañe al applet. Arranca Pico de nuevo y tecela el siguiente código en un nuevo buffer:

```
<HTML>
<HEAD>
<TITLE>A Simple Program</TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Graba este código en un fichero llamado HelloWorld.html. Teclea **Ctrl-X** para salir de Pico.

b. Compilar el Fichero Fuente.

En el prompt, teclea el siguiente comando y pulsa **Return**:

```
javac HelloWorld.java
```

El compilar debería generar un fichero de bytecodes Java, HelloWorld.class.

c. Ejecutar el Programa.

Aunque puedes ver tu applet usando un navegador, podrías encontrar más sencillo probarlos usando la aplicación appletviewer que viene con la plataforma Java™. Para ver el applet HelloWorld usando el appletviewer, teclea esto en la línea de comandos:

```
appletviewer Hello.html
```

Ahora deberías ver:



Felicidades! tu applet Funciona!

[↑subir](#)

4. ¿A dónde ir desde aquí?

Para continuar con la introducción al lenguaje Java, puedes visitar estas secciones:

[Por dónde Empezar](#)

[Conceptos Básicos](#)

[↑Subir](#)

[Ozito](#)

Tu Primera Taza de Java en MacOS



Instrucciones Detalladas para Tu Primer Programa

Las siguientes instrucciones te ayudarán a escribir tu primer programa Java. Estas instrucciones son para usuarios de la plataforma MacOS.

1. [Checklist](#)

2. [Crear tu Primera Aplicación](#)

- [Crear un Fichero Fuente Java](#)
- [Compilar el Fichero Fuente](#)
- [Ejecutar el Programa](#)

3. [Crear Tu Primer Applet](#)

4. [Dónde ir desde Aquí](#)

1. Checklist

1. Un Entorno de Desarrollo para la Plataforma Java. Puedes [descargar](#) el MRJ SDK (Macintosh Runtime Environment for Java Software Development Kit) desde la website de Apple.
2. Un entorno de ejecución para la misma versión de la plataforma Java. Puedes [descargar](#) el MRJ (Macintosh Runtime Enviroment for Java) desde la website de Apple.
3. Un Editor de texto. En este ejemplo, usaremos SimpleText, el sencillo editor incluido con las plataformas Mac OS. Si usas un editor diferente no te será difícil adaptar estas instrucciones.

Estas tres cosas son todo lo que necesitas para programar en Java.

2. Crear Tu Primera Aplicación

Tu primer programa, HelloWorldApp, simplemente mostrará el saludo "Hello world!". Para crear este programa, deberás:

- Crear un **fichero fuente** Java. Un fichero fuente contiene texto, escrito en el lenguaje de programación Java, que tu y otros programadores pueden entender. Se puede usar cualquier editor de texto para crear y editar ficheros fuente.
- Compilar el fichero fuente en un **fichero de bytecodes**. El compilador de Java, javac, toma nuestro fichero fuente y lo traduce en instrucciones que la Máquina Virtual Java (Java VM) puede entender. El compilador pone estas

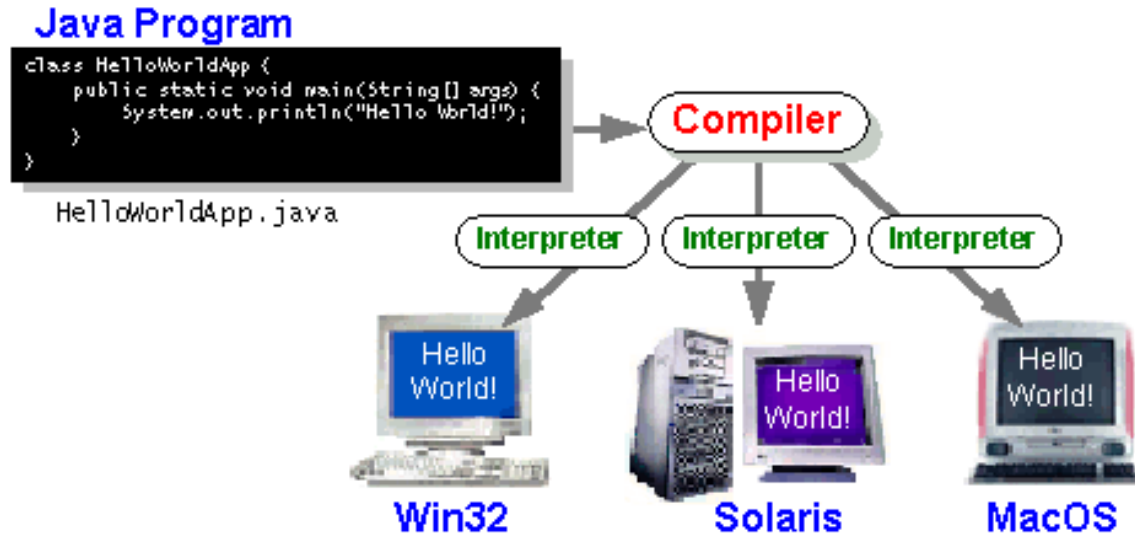
¿Por qué están de moda los Bytecodes

Habrás oído que con el lenguaje de programación Java, puedes "escribir una vez, ejecutar en cualquier parte". Esto significa que cuando se compila un programa, no se generan instrucciones para una plataforma específica. En su lugar, se generan bytecodes Java, que son instrucciones para la Máquina Virtual Java (Java VM). Si tu plataforma- sea Windows, UNIX, MacOS o un navegador de internet-- tiene la Java VM, podrá

instrucciones en un fichero de bytecodes.

entender los bytecodes.

- Ejecutar le programa contenido en el fichero de bytecodes. La máquina virtual Java está implementada por un intérprete Java, java. Este intérprete toma nuestro fichero de bytecodes y lleva a cabo las instrucciones traduciéndolas a instrucciones que nuestro ordenador puede entender.



[Subir](#)

a. Crear un Fichero Fuente Java.

Tienes dos opciones:

- Puedes grabar el fichero [HelloWorldApp.java](#) en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).
- O, puedes seguir estas (largas) instrucciones:

1. Arranca SimpleText. En un nuevo documento, teclea el siguiente código:

```
/**  
 * The HelloWorldApp class implements an application that  
 * simply displays "Hello World!" to the standard output.  
 */  
class HelloWorldApp {  
    public static void main(String[] args) {  
        // Display "Hello World!"  
        System.out.println("Hello World!");  
    }  
}
```

Se cuidadoso cuando lo teclees



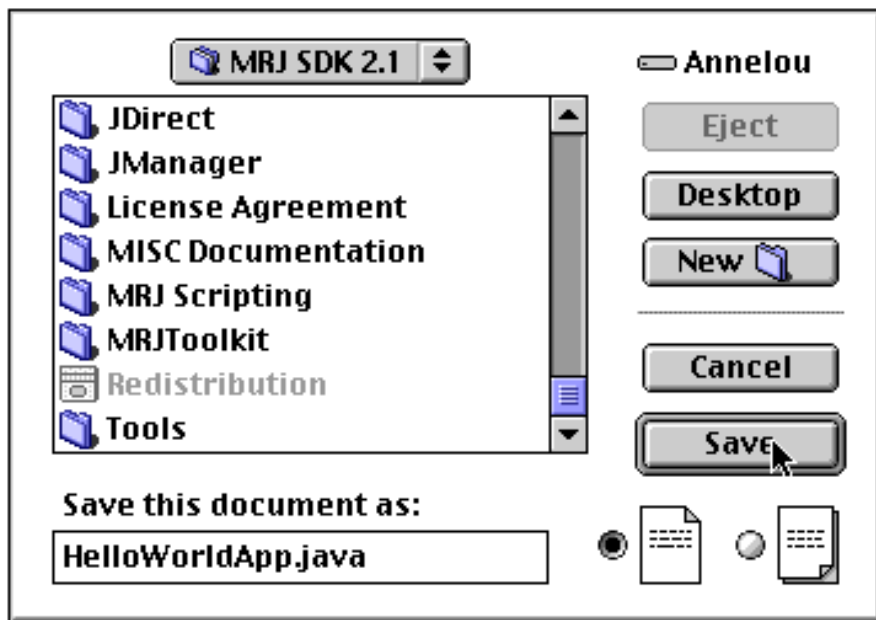
Teclea todo el código, comandos y nombres de ficheros exactamente como los ves. El compilador y el intérprete Java son sensibles a las mayúsculas.

HelloWorldApp ~~≠~~ helloworldapp

2. Graba este código en un fichero. Desde la barra de menú, selecciona **File > Save As**. En la caja de diálogo **Save As**:

- Especificar la carpeta donde grabarás el fichero. En este ejemplo, el es la carpeta MRJ SDK 2.1.
- En la caja de texto **Save This Document as:**, teclea, "HelloWorldApp.java"

Cuando hayas terminado la caja de diálogo se debería parecer a esto:

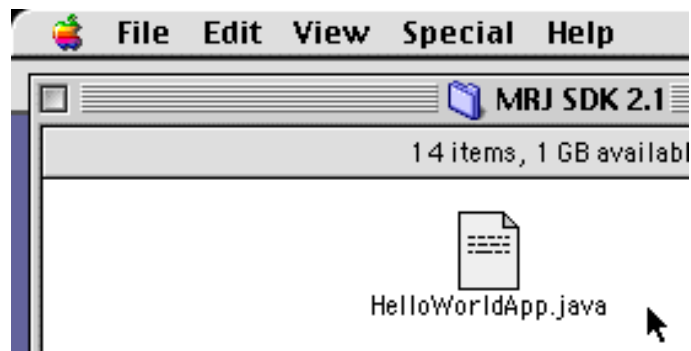


Ahora pulsa sobre **Save**, y sal de SimpleText.

[↑ Subir](#)

b. Compilar el Fichero Fuente

Ve a la carpeta MRJ SDK 2.1 y allí deberías ver algo como esto:

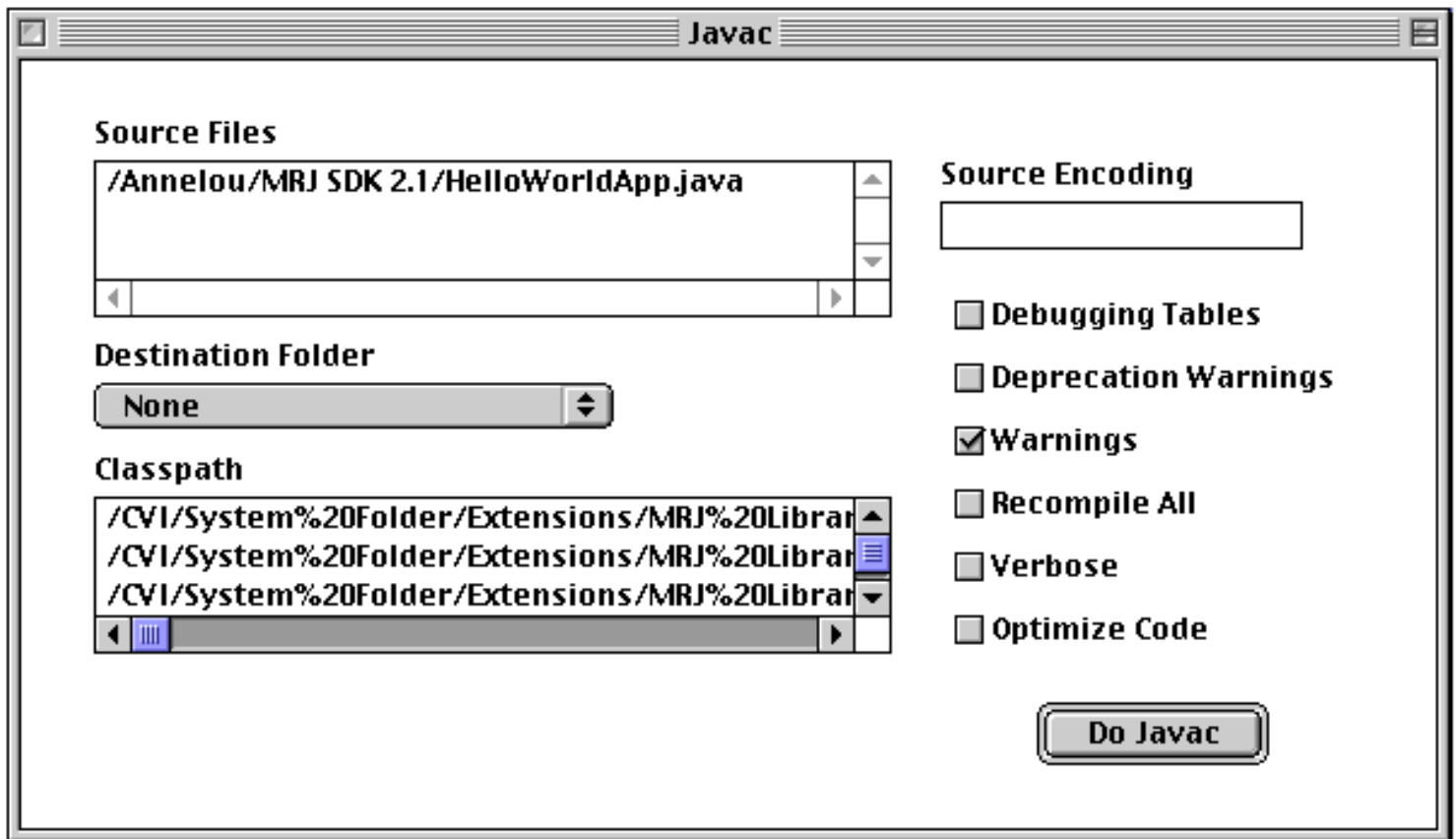


Habr  una carpeta llamada Tools. En esta carpeta hay una carpeta llamada MRJTools.

Abrela. Ver s un programa llamado javac.



Ahora arrastra nuestro HelloWorldApp.java sobre esta aplicaci n Javac. Se abrir  javac y deber s ver:



En la caja **Source Files** nos ofrece el path absoluto que acabamos de crear. Por ahora no tenemos que hacer nada m s exceto pulsar el bot n **Do Javac**.

Si no aparecen mensajes de error, felicidades. Tu programa se ha compilado con  xito.

Explicación de Error

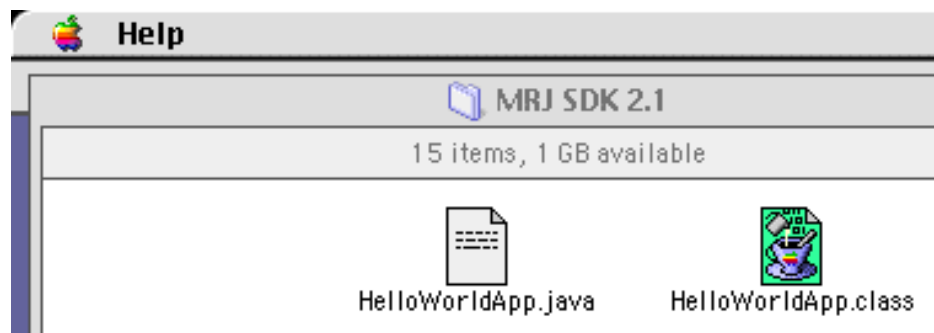
Si arrastras nuestro fichero .java sobre el programa javac y el fichero se muda sobre la aplicación javac

Cuando se intenta esto y lo único que sucede es que nuestro fichero .java se copia o se mueve encima de nuestra aplicación javac, tenemos reconstruir nuestro escritorio.

Para hacer esto tenemos que volver a arrancar el ordenador y pulsar y mantener las teclas "Apple" - y "Alt" hasta que obtengamos una ventana preguntándonos si queremos reconstruir nuestro escritorio.

Respondemos que sí. Cuando el ordenador haya finalizado debemos poder arrastrar nuestro fichero .java sobre la aplicación javac.

El compilador ha generado un fichero de bytecodes Java, HelloWorldApp.class. Mira en la misma carpeta en la que grabaste el fichero .java y verás el fichero .class:



Ahora que tenemos un fichero .class, puedes ejecutar tu programa.

[↑subir](#)

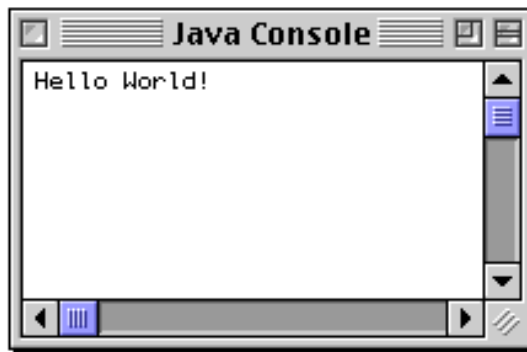
c. Ejecutar el Programa

En la carpeta MRJ SDK 2.1 hay otra carpeta llamada JBindery. Abrela y debería hacer una aplicación llamada JBindery



Arrastra el fichero HelloWorldApp.class sobre el icono JBindery.

Si obtienes esto:



Felicidades!, tu programa funciona.

[↑subir](#)

3. Crear tu Primer Applet

HelloWorldApp es un ejemplo de una aplicación Java, un programa solitario. Ahora crearás un applet Java, llamado HelloWorld, que también muestra el salido "Hello world!". Sin embargo, al contrario que HelloWorldApp, el applet se ejecuta sobre un navegador compatible con Java, como HotJava, Netscape Navigator, o Microsoft Internet Explorer.

Para crear este applet, debes realizar los mismos pasos básicos que antes: crear un fichero fuente Java; compilarlo y ejecutar el programa.

a. Crear un Fichero Fuente Java.

De nuevo, tienes dos opciones:

- Puedes grabar los ficheros [HelloWorld.java](#) y [Hello.html](#) en tu ordenador y así evitarte todo el tecleo. Luego puedes ir directo al [paso b](#).
- O, puedes seguir estas instrucciones:

1. Arranca SimpleText y teclea el siguiente código en un nuevo documento:

```
import java.applet.*;
import java.awt.*;

/**
 * The HelloWorld class implements an applet that
 * simply displays "Hello World!".
 */
public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        // Display "Hello World!"
        g.drawString("Hello world!", 50, 25);
    }
}
```

Graba este código en un fichero llamado HelloWorld.java.

2. También necesitas un fichero HTML que acompañe a tu applet. Teclea el siguiente código en nuevo documento del SimpleText:

```
<HTML>
<HEAD>
<TITLE>A Simple Program</TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Graba este código en un fichero llamado Hello.html.

b. Compilar el Código Fuente.

Compila el fichero fuente HelloWorld.java usando javac.

El compilador debería generar el fichero de bytecodes Java, HelloWorld.class.

c. Ejecutar el Programa.

Aunque puedes ver tu applet usando un navegador, podrías encontrar más sencillo probarlos usando la aplicación appletviewer que viene con la plataforma Java™. Para ver el applet HelloWorld usando el applet runner, abre la carpeta **Apple applet viewer** en la carpeta MRJ SDK 2.1. Debería haber una aplicación llamada Apple Applet Runner.




Apple Applet Runner

Arrastra nuestro fichero Hello.html sobre esta aplicación.

Ahora deberías ver:



Felicidades! Tu Applet funciona.

 [subir](#)

4. ¿A dónde ir desde aquí?

Para continuar con la introducción al lenguaje Java, puedes visitar estas secciones:

[Por dónde Empezar](#)

[Conceptos Básicos](#)

 [Subir](#)

Ozito

¿Por Dónde Empezar?

Bien, si estás interesado en este potente lenguaje y no sabes nada de él te recomiendo que te des una vuelta por la site de sus creadores:

<http://java.sun.com>

En esta otra dirección podrás bajarte la última versión del JDK:

<http://java.sun.com/products/index.html>

Dentro del JDK encontrarás varias aplicaciones que te ayudarán en el trabajo con Java, por ejemplo:

- [javac](#) El compilador Java por excelencia, un compilador de línea de comandos, que te permitirá crear tus programas y applets en Java.
- [appletviewer](#) Un visualizador de Applets para no tener que cargarlos en un navegador.
- [java](#) El intérprete que te permitirá ejecutar tus aplicaciones creadas en Java.
- [javap](#) Un descompilador que te permite ver el contenido de las clases compiladas.

Pero si no te interesa el trabajo puro y duro con Java, puedes bajarte cualquiera de los entornos integrados de desarrollo de Java, como el Visual J++ de Microsoft en <http://www.microsoft.com/visualj/> o el Wrokshop de Sun en <http://www.sun.com/developer-products/java/>.

Ozito

javac - El compilador de Java

Síntaxis de utilización

```
javac [opciones] fichero.java ...
```

```
javac_g [ opciones] fichero.java ...
```

Descripción

El comando javac compila el código fuente Java y lo convierte en Bytecodes. Después se puede utilizar el intérprete Java - java - o el navegador en caso de los applets para interpretar esos Bytecodes Java.

El código fuente de Java debe estar contenido en ficheros con extensión '.java'. Para cada clase definida en el fichero fuente pasado a javac, el compilador almacena los bytecodes resultantes en un fichero llamado **nombredeclase.class**. El compilador sitúa estos ficheros en el mismo directorio en el que estaba el fichero fuente (a menos que se especifique la opción -d).

Cuando defines tus propias clases necesitarás especificar su localización. Utiliza CLASSPATH para hacer esto. CLASSPATH consiste en una lista de directorios separados por puntos y comas que especifican el path. Si el fichero fuente pasado a javac hace referencia a otra clase que no está definida en otros ficheros pasados a javac, el compilador busca las clases referenciadas utilizando CLASSPATH. Por ejemplo:

```
.;C:/users/java/clases
```

Observa que el sistema siempre añade la localización de las clases del sistema al final del CLASSPATH a menos que se utilice la opción -classpath para especificar un path.

javac_g es una versión no optimizada de javac especialmente preparada para utilizar con depuradores como jdb.

Opciones

-classpath path

Especifica el path que javac utilizará para buscar las clases. Sobreescribe el path por defecto generado por la variable de entorno CLASSPATH. Los directorios están separados por puntos y comas. Por ejemplo:

```
.;C:/users/java/clases;C:\tools\java\clases
```

-d directorio

Especifica el directorio raíz para el árbol de clases. Hacer esto:

```
javac -d <mi_dir> miprograma.java
```

hace que los ficheros '.class' del fichero fuente 'miprograma.java' sean guardados en el directorio 'mi_dir'.

-g

Habilita la generación de tablas de depurado. Estas tablas contienen información sobre los números de líneas y las variables locales - información utilizada por las herramientas de depurado de Java. Por defecto, sólo genera números de líneas a menos que se active la optimización (-O).

-nowarn

Desactiva los avisos. Si se utiliza el compilador no imprime ningún aviso.

-O

Optimiza el código compilado introduciendo en línea los métodos finales y privados. Observa que esto puede hacer que el tamaño de tus clases crezca demasiado.

-verbose

Hace que el compilador y el enlazador impriman los mensajes sobre los ficheros fuentes que están siendo compilados y que ficheros .class están siendo cargados.

appletviewer - El visualizador de Applets

Este comando te permite ejecutar applets fuera del contexto de los navegadores de la WWW.

Síntaxis

```
appletviewer [ opciones ] urls ...
```

Descripción

El comando appletviewer conecta los documentos o recursos designados por urls y muestra cada applet referenciado en su propia ventana.

Nota: si los documentos referenciados con las urls no contienen ningún applet con la etiqueta APPLET, appletviewer no hace nada.

Opciones

-debug

Arranca el visualizador dentro del depurador de Java, el jdb, permitiéndote depurar los applets en el documento

java - El intérprete Java

Síntaxis de Utilización

```
java [ opciones ] nombredelclase <argumentos>
java_g [ opciones ] nombredelclase <argumentos>
```

Descripción

El comando java ejecuta los bytecodes Java creados por el compilador [javac](#).

El argumento nombredelclase es el nombre de la clase que se va a ejecutar. Debe estar totalmente cualificado e incluir su nombre de paquete en el nombre, por ejemplo:

```
java java.lang.String
```

Observa que todos los argumentos que aparecen después de nombredelclase en la línea de comandos son pasados al método main() de la clase.

java espera que los bytecodes de la clase se encuentren en un fichero llamado nombredelclase.class que ha sido generado al compilar el fichero fuente correspondiente con javac. Todos los ficheros de bytecodes java tienen la extensión .class que añade automáticamente el compilador cuando la clase es compilada. La clase debe contener un método main() definido de la siguiente forma:

```
class Unaclass {
    public static void main( String argv []) {
        ...
    }
}
```

java ejecuta el método main y luego sale a menos que éste cree uno o más threads. Si el método main() crea algún thread, java no sale hasta que haya terminado el último thread.

Normalmente, compilar tus ficheros fuente con javac y luego ejecutas los programas con java. Sin embargo, se puede utilizar java para compilar y ejecutar los programas cuando se utiliza la opción -cs. Cuando se carga una clase se compara su fecha de última modificación con la fecha del fichero fuente. Si el fuente ha sido modificado, se recompila y se carga el nuevo fichero de bytecodes. java repite este procedimiento hasta que todas las clases se han compilado y cargado correctamente.

El intérprete puede determinar si una es legítima a través de un mecanismo de verificación. Esta verificación asegura que los bytecodes que están siendo interpretados no violan las restricciones del lenguaje.

java_g es una versión no optimizada de java pensada para ser utilizada con depuradores como el jdb.

Opciones

-debug

Permite que el depurador de Java jdb se añada a esta sesión java. Cuando se especifica esta opción en la línea de comandos, java muestra una password que se debe utilizar cuando empieza la sesión de depurado.

-cs, -checksource

Cuando se carga una clase compilada, esta opción hace que la fecha de modificación de los bytecodes sea comparada con la del fichero fuente. Si el fichero fuente ha sido modificado recientemente, es recompilado y se cargan los nuevos bytecodes.

-classpath path

Especifica el path que utilizará java para buscar las clases. Sobreescribe el valor por defecto de la variable de entorno CLASSPATH. Los directorios están separados por comas.

-mx x

Selecciona el máximo tamaño de memoria reservada para la pila del recolector de basura a x. El valor por defecto es 16 megabytes de memoria. x debe ser mayor de 1000 bytes

Por defecto, x se mide en bytes. Pero puedes especificarlo en kb o Mb añadiéndole la letra 'k' para kilobytes o 'm' para megabytes.

-ms x

Selecciona el tamaño inicial de la memoria reservada para la pila del recolector de basura a x. El valor por defecto es 16 megabytes de memoria. x debe ser mayor de 1000 bytes

Por defecto, x se mide en bytes. Pero puedes especificarlo en kb o Mb añadiéndole la letra 'k' para kilobytes o 'm' para megabytes.

-noasyncgc

Desactiva la recolección de basura asíncrona. Cuando se activa la recolección de basura no tiene lugar a menos que la llame explícitamente o el programa sale de la memoria. Normalmente, la recolección de basura se ejecuta en un thread asíncrono en paralelo con otros threads.

-ss x

Cada thread Java tiene dos pilas: una para el código Java y otra para el código C. La opción `-ss` selecciona el tamaño máximo de la pila que puede ser utilizada por el código C en un thread a `x`. Cada uno de los threads ejecutados dentro del programa que se paso a **java** tiene `x` como su tamaño de pila C. El valor por defecto es 128 kilobytes de memoria. `x` debe ser mayor de 1000 bytes

Por defecto, `x` se mide en bytes. Pero puedes especificarlo en kb o Mb añadiéndole la letra 'k' para kilobytes o 'm' para megabytes.

`-oss x`

Cada thread Java tiene dos pilas: una para el código Java y otra para el código C. La opción `-oss` selecciona el tamaño máximo de la pila que puede ser utilizada por el código Java en un thread a `x`. Cada uno de los threads ejecutados dentro del programa que se paso a **java** tiene `x` como su tamaño de pila Java. El valor por defecto es 400 kilobytes de memoria. `x` debe ser mayor de 1000 bytes

Por defecto, `x` se mide en bytes. Pero puedes especificarlo en kb o Mb añadiéndole la letra 'k' para kilobytes o 'm' para megabytes.

`-t`

Imprime un rastro de las instrucciones ejecutadas (sólo en `java_g`).

`-v, -verbose`

Hace que java en el canal `stdout` cada vez que se crea un fichero `class`.

`verify`

Ejecuta el verificador en todo el código.

`-verifyremote`

Ejecuta el verificador para todo el código que es carga dentro del sistema a través de `classloader`. `verifyremote` es el valor por defecto para el intérprete.

`-noverify`

Desactiva al verificación.

`-verbosegc`

Hace que el recolector de basura imprima mensajes cada vez que libere memoria.

`-DnombrePropiedad=nuevoValor`

Redefine un valor de una propiedad. `nombrePropiedad` es el nombre de la propiedad cuyo valor se quiere cambiar y `nuevoValor` es el valor a cambiar. Por ejemplo, esta línea de comando:

```
java -Dawt.button.color=green ...
```

selecciona el color de la propiedad `awt.button.color` a verde. `java` acepta cualquier número de opciones `-D` en la línea de comandos.

Ozito

javap - El descompilador de Ficheros de Clases Java

Desensambla los ficheros class.

Síntaxis de Utilización

```
javap [ opciones ] clase ...
```

Descripción

El comando javap desensambla un fichero class. Su salida depende de las opciones utilizadas. Si no se utilizan opciones, javap imprime los campos y los métodos públicos de la clase pasada. javap muestra la salida en stdout. Por ejemplo, la siguiente declaración de clase:

```
class C {
    static int a= 1;
    static int b= 2;
    static {
        System.out.println(a);
    }
    static {
        a++;
        b = 7;
        System.out.println(a);
        System.out.println(b);
    }
    static {
        system.out println(b);
    }
    public static void main(String args[]) {
        C c = new C();
    }
}
```

Cuando el resultado de la clase C, sea pasado a javap sin utilizar opciones, resultará la siguiente salida:

```
Compiled from C:\users\clases\C.java
private class C extends java.lang.Object {
    static int a;
    static int b;
    public static void main(java.lang.String[]);
    public C();
    static void();
}
```

```
}
```

Opciones

- l
Imprime una tabla de línea y variables locales.
- p
Imprime también los miembros y método privados y protegidos de la clase, además de los miembros públicos.
- c

Imprime el código desensamblado, es decir, las instrucciones que comprenden los bytecodes Java, para cada uno de los métodos de la clase. Por ejemplo, si se pasa la clase C a javap utilizando -c el resultado sería la siguiente salida:

```
Compiled from C:\users\clases\C.java
private class C extends java\lang\Object {
    static int a;
    static int b;
    public static void main(java\lang\String[]);
    public C();
    static void();
}
```

```
Method void main(java\lang\String [])
    0 new #4
    3 invokenonvirtual # 9 () V>
    6 return
```

```
Method C()
    0 aload_0 0
    1 invokenonvirtual #10 ()V>
    4 return
```

```
Method void()
    0 iconst_1
    1 putstatic #7
    4 getstatic #6
    7 getstatic #7
    10 invokevirtual #8
    13 getstatic #7
    16 iconst_1
    17 iadd
    18 putstatic #7
    21 bipush 7
    23 putstatic #5
```

```
26 getstatic #6
29 getstatic #7
32 invokevirtual #8
35 getstatic #6
38 getstatic #5
41 invokevirtual #8
44 iconst_2
45 putstatic #5
48 getstatic #6
51 getstatic #5
54 invokevirtual #8
57 return
}
```

-classpath path

Especifica el path que javap utilizará para buscar las clases. Sobreescribe el valor de la variable de entorno CLASSPATH. Los directorios deben estar separados por puntos y comas.

Conceptos Básicos de Programación Orientada a Objetos

¿Qué son los objetos?

En informática, un OBJETO es un conjunto de variables y de los métodos relacionados con esas variables.

Un poco más sencillo: un objeto contiene en sí mismo la información y los métodos o funciones necesarios para manipular esa información.

Lo más importante de los objetos es que permiten tener un control total sobre 'quién' o 'qué' puede acceder a sus miembros, es decir, los objetos pueden tener miembros públicos a los que podrán acceder otros objetos o miembros privados a los que sólo puede acceder él. Estos miembros pueden ser tanto variables como funciones.

El gran beneficio de todo esto es la encapsulación, el código fuente de un objeto puede escribirse y mantenerse de forma independiente a los otros objetos contenidos en la aplicación.

¿Qué son las clases?

Una CLASE es un proyecto, o prototipo, que define las variables y los métodos comunes a un cierto tipo de objetos.

Un poco más sencillo: las clases son las matrices de las que luego se pueden crear múltiples objetos del mismo tipo. La clase define las variables y los métodos comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.

Primero deberemos crear una clase antes de poder crear objetos o ejemplares de esa clase.

¿Qué son los mensajes?

Para poder crear una aplicación necesitarás más de un objeto, y estos objetos no pueden estar aislados unos de otros, pues bien, para comunicarse esos objetos se envían mensajes.

Los mensajes son simples llamadas a las funciones o métodos del objeto con el se quiere comunicar para decirle que haga cualquier cosa.

¿Qué es la herencia?

Qué significa esto la herencia, quién hereda qué; bueno tranquilo, esto sólo significa que puedes crear una clase partiendo de otra que ya exista.

Es decir, puedes crear una clase a través de una clase existente, y esta clase tendrá todas las variables y los métodos de su 'superclase', y además se le podrán añadir otras variables y métodos propios.

Se llama 'Superclase' a la clase de la que desciende una clase, puedes ver más sobre la declaración de clases en la página [Declarar Clases](#).

Variables y Tipos de Datos

Las variables son las partes importantes de un lenguaje de programación: ellas son las entidades (valores, datos) que actúan y sobre las que se actúa.

Una declaración de variable siempre contiene dos componentes, el tipo de la variable y su nombre:

`tipoVariable nombre;`

Tipos de Variables

Todas las variables en el lenguaje Java deben tener un tipo de dato. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorías de datos principales en el lenguaje Java: los tipos primitivos y los tipos referenciados.

Los tipos primitivos contienen un sólo valor e incluyen los tipos como los enteros, coma flotante, los caracteres, etc... La tabla siguiente muestra todos los tipos primitivos soportados por el lenguaje Java, su formato, su tamaño y una breve descripción de cada uno:

Tipo	Tamaño/Formato	Descripción
(Números enteros)		
byte	8-bit complemento a 2	Entero de un Byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
(Números reales)		
float	32-bit IEEE 754	Coma flotante de precisión simple
double	64-bit IEEE 754	Coma flotante de precisión doble
(otros tipos)		
char	16-bit Character	Un sólo carácter
boolean	true o false	Un valor booleano (verdadero o falso)

Los tipos referenciados se llaman así porque el valor de una variable de referencia es una referencia (un puntero) hacia el valor real. En Java tenemos los arrays, las clases y los interfaces como tipos de datos referenciados.

Nombres de Variables

Un programa se refiere al valor de una variable por su nombre. Por convención, en Java, los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra

mayúscula).

Un nombre de variable Java:

1. debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. Unicode es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos. Unicode permite la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.
2. no puede ser el mismo que una palabra clave o el nombre de un valor booleano (true or false)
3. no deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.

La regla número 3 implica que podría existir el mismo nombre en otra variable que aparezca en un ámbito diferente.

Por convención, los nombres de variables empiezan por un letra minúscula. Si una variable está compuesta de más de una palabra, como 'nombreDato' las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.

Operadores

Los operadores realizan algunas funciones en uno o dos operandos. Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo. La notación de prefijo significa que el operador aparece antes de su operando:

operador operando

La notación de sufijo significa que el operador aparece después de su operando:

operando operador

Todos los operadores binarios de Java tienen la misma notación, es decir aparecen entre los dos operandos:

op1 operator op2

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas. El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si sumas dos enteros, obtendrás un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales. lógicos y de desplazamiento y de asignación.

Operadores Aritméticos

El lenguaje Java soporta varios operadores aritméticos - incluyendo + (suma), - (resta), * (multiplicación), / (división), y % (módulo)-- en todos los números enteros y de coma flotante. Por ejemplo, puedes utilizar este código Java para sumar dos números:

sumaEsto + aEsto

O este código para calcular el resto de una división:

divideEsto % porEsto

Esta tabla resume todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
----------	-----	-------------

+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplifica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas.

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos, ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

Operadores Relacionales y Condicionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.

Esta tabla resume los operadores relacionales de Java:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Frecuentemente los operadores relacionales se utilizan con otro juego de operadores, los operadores condicionales, para construir expresiones de decisión más complejas. Uno de estos operadores es && que realiza la operación Y booleana. Por ejemplo puedes utilizar dos operadores relacionales diferentes junto con && para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para

determinar si un índice de un array está entre dos límites- esto es, para determinar si el índice es mayor que 0 o menor que NUM_ENTRIES (que se ha definido previamente como un valor constante):

```
0 < index && index < NUM_ENTRIES
```

Observa que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

```
((count > NUM_ENTRIES) && (System.in.read() != -1))
```

Si count es menor que NUM_ENTRIES, la parte izquierda del operando de && evalúa a false. El operador && sólo devuelve true si los dos operandos son verdaderos. Por eso, en esta situación se puede determinar el valor de && sin evaluar el operador de la derecha. En un caso como este, Java no evalúa el operando de la derecha. Así no se llamará a System.in.read() y no se leerá un carácter de la entrada standard.

Aquí tienes tres operadores condicionales:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
	op1 op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son booleanos. Similarmente, | es un sinonimo de || si ambos operandos son booleanos.

Operadores de Desplazamiento

Los operadores de desplazamiento permiten realizar una manipulación de los bits de los datos. Esta tabla resume los operadores lógicos y de desplazamiento disponibles en el lenguaje Java:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1 (sin signo)
&	op1 & op2	bitwise and
	op1 op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~ op	bitwise complemento

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador. Por ejemplo:

13 >> 1;

desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el 6 decimal. Observe que el bit situado más a la derecha desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.

Los otros operadores realizan las funciones lógicas para cada uno de los pares de bits de cada operando. La función "y" activa el bit resultante si los dos operandos son 1.

op1	op2	resultado
0	0	0
0	1	0
1	0	0
1	1	1

Supon que quieres evaluar los valores 12 "and" 13:

12 & 13

El resultado de esta operación es 12. ¿Por qué? Bien, la representación binaria de 12 es 1100 y la de 13 es 1101. La función "and" activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si colocas en línea los dos operandos y realizas la función "and", puedes ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 porque al menos uno de los dos operandos es 0:

```
      1101
&     1100
-----
      1100
```

El operador | realiza la operación O inclusiva y el operador ^ realiza la operación O exclusiva. O inclusiva significa que si uno de los dos operandos es 1 el resultado es 1.

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	1

O exclusiva significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0:

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador complemento invierte el valor de cada uno de los bits del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

Operadores de Asignación

Puedes utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo. Específicamente, supón que quieres añadir un número a una variable y asignar el resultado dentro de la misma variable, como esto:

```
i = i + 2;
```

Puedes ordenar esta sentencia utilizando el operador +=.

```
i += 2;
```

Las dos líneas de código anteriores son equivalentes.

Esta tabla lista los operadores de asignación y sus equivalentes:

Operador	Uso	Equivale a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
=	op1 = op2	op1 = op1 op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Expresiones

Las expresiones realizan el trabajo de un programa Java. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa Java. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor.

Definición: Una expresión es una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

El tipo del dato devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión `count++` devuelve un entero porque `++` devuelve un valor del mismo tipo que su operando y `count` es un entero. Otras expresiones devuelven valores booleanos, cadenas, etc...

Una expresión de llamada a un método devuelve el valor del método; así el tipo de dato de una expresión de llamada a un método es el mismo tipo de dato que el valor de retorno del método. El método `System.in.read()` se ha declarado como un entero, por lo tanto, la expresión `System.in.read()` devuelve un entero.

La segunda expresión contenida en la sentencia `System.in.read() != -1` utiliza el operador `!=`. Recuerda que este operador comprueba si los dos operandos son distintos. En esta sentencia los operandos son `System.in.read()` y `-1`. `System.in.read()` es un operando válido para `!=` porque devuelve un entero. Así `System.in.read() != -1` compara dos enteros, el valor devuelto por `System.in.read()` y `-1`. El valor devuelto por `!=` es `true` o `false` dependiendo de la salida de la comparación.

Como has podido ver, Java te permite construir expresiones compuestas y sentencias a partir de varias expresiones pequeñas siempre que los tipos de datos requeridos por una parte de la expresión correspondan con los tipos de datos de la otra. También habrás podido concluir del ejemplo anterior, el orden en que se evalúan los componentes de una expresión compuesta.

Por ejemplo, toma la siguiente expresión compuesta:

`x * y * z`

En este ejemplo particular, no importa el orden en que se evalúe la expresión porque el resultado de la multiplicación es independiente del orden. La salida es siempre la misma sin importar el orden en que se apliquen las multiplicaciones. Sin embargo, esto no es cierto para todas las expresiones. Por ejemplo, esta expresión obtiene un resultado diferente dependiendo de si se realiza primero la suma o la división:

`x + y / 100`

Puedes decirle directamente al compilador de Java cómo quieres que se evalúe una expresión utilizando los paréntesis (y). Por ejemplo, para aclarar la sentencia anterior, se podría escribir: $(x + y) / 100$.

Si no le dices explícitamente al compilador el orden en el que quieres que se realicen las operaciones, él decide basándose en la precedencia asignada a los operadores y otros elementos que se utilizan dentro de una expresión. Los operadores con una precedencia más alta se evalúan primero. Por ejemplo, el operador división tiene una precedencia mayor que el operador suma, por eso, en la expresión anterior $x + y / 100$, el compilador evaluará primero $y / 100$. Así

$x + y / 100$

es equivalente a:

$x + (y / 100)$

Para hacer que tu código sea más fácil de leer y de mantener deberías explicar e indicar con paréntesis los operadores que se deben evaluar primero.

La tabla siguiente muestra la precedencia asignada a los operadores de Java. Los operadores se han listado por orden de precedencia de mayor a menor. Los operadores con mayor precedencia se evalúan antes que los operadores con un precedencia relativamente menor. Los operadores con la misma precedencia se evalúan de izquierda a derecha.

Precedencia de Operadores en Java

operadores sufijo	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creación o tipo	new (type)expr
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
AND lógico	&&
OR lógico	
condicional	? :
asignación	= += -= *= /= %= ^= &= = <<= >>= >>>=

Sentencias de Control de Flujo

Las sentencias de control de flujo determinan el orden en que se ejecutarán las otras sentencias dentro del programa. El lenguaje Java soporta varias sentencias de control de flujo, incluyendo:

Sentencias	palabras clave
toma de decisiones	if-else, switch-case
bucles	for, while, do-while
excepciones	try-catch-finally, throw
miscelaneas	break, continue, label:, return

Nota: Aunque goto es una palabra reservada, actualmente el lenguaje Java no la soporta. Podemos utilizar las [rupturas etiquetadas](#) en su lugar.

La sentencia if-else

La sentencia if-else de java proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio. Por ejemplo, supón que tu programa imprime información de depurado basándose en el valor de una variable booleana llamada DEBUG. Si DEBUG fuera verdadera true, el programa imprimiría la información de depurado, como por ejemplo, el valor de una variable como x. Si DEBUG es false el programa procederá normalmente. Un segmento de código que implemente esto se podría parecer a este:

```
. . .
if (DEBUG)
    System.out.println("DEBUG: x = " + x);
. . .
```

Esta es la versión más sencilla de la sentencia if: la sentencia gobernada por if se ejecuta si alguna condición es verdadera. Generalmente, la forma sencilla de if se puede escribir así:

```
if (expresión)
    sentencia
```

Pero, ¿y si quieres ejecutar un juego diferente de sentencias si la expresión es falsa? Bien, puedes utilizar la sentencia else. Echemos un vistazo a otro ejemplo. Supón que tu programa necesita realizar diferentes acciones dependiendo de que el usuario pulse el botón OK o el botón Cancel en un ventana de alarma. Se podría hacer esto utilizando una sentencia if:

```
. . .
// Respuesta dependiente del botón que haya pulsado el usuario
// OK o Cancel
. . .
if (respuesta == OK) {
    . . .
    // Código para la acción OK
    . . .
} else {
    . . .
    // código para la acción Cancel
    . . .
}
```

Este uso particular de la sentencia else es la forma de capturarlo todo. Existe otra forma de la sentencia else, else if que ejecuta una sentencia basada en otra expresión. Por ejemplo, supón que has escrito un programa que asigna notas basadas en la

puntuación de un examen, un Sobresaliente para una puntuación del 90% o superior, un Notable para el 80% o superior y demás. odrías utilizar una sentencia if con una serie de comparaciones else if y una setencia else para escribir este código:

```
int puntuacion;
String nota;

if (puntuacion >= 90) {
    nota = "Sobresaliente";
} else if (puntuacion >= 80) {
    nota = "Notable";
} else if (puntuacion >= 70) {
    nota = "Bien";
} else if (puntuacion >= 60) {
    nota = "Suficiente";
} else {
    nota = "Insuficiente";
}
```

Una sentencia if puede tener cualquier número de sentencias de acompañamiento else if. Podrías haber observado que algunos valores de puntuacion pueden satisfacer más una de las expresiones que componen la sentencia if. Por ejemplo, una puntuación de 76 podría evaluarse como true para dos expresiones de esta sentencia: puntuacion >= 70 y puntuacion >= 60. Sin embargo, en el momento de ejecución, el sistema procesa una sentencia if compuesta como una sólo; una vez que se ha satisfecho una condición (76 >= 70), se ejecuta la sentencia apropiada (nota = "Bien";), y el control sale fuera de la sentencia if sin evaluar las condiciones restantes.

La sentencia switch

La sentencia switch se utiliza para realizar sentencias condicionalmente basadas en alguna expresión. Por ejemplo, supon que tu programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha. Supon que también quieres mostrar el nombre del mes basándose en su número entero equivalente. Podrias utilizar la sentencia switch de Java para realizar esta tarea:

```
int mes;
. . .
switch (mes) {
case 1:  System.out.println("Enero"); break;
case 2:  System.out.println("Febrero"); break;
case 3:  System.out.println("Marzo"); break;
case 4:  System.out.println("Abril"); break;
case 5:  System.out.println("May0"); break;
case 6:  System.out.println("Junio"); break;
case 7:  System.out.println("Julio"); break;
case 8:  System.out.println("Agosto"); break;
case 9:  System.out.println("Septiembre"); break;
case 10: System.out.println("Octubre"); break;
case 11: System.out.println("Noviembre"); break;
case 12: System.out.println("Diciembre"); break;
}
```

La sentencia switch evalúa su expresión, en este caso el valor de mes, y ejecuta la sentencia case apropiada. Decidir cuando utilizar las sentencias if o switch dependen del juicio personal. Puedes decidir cual utilizar basándose en la buena lectura del código o en otros factores.

Cada sentencia case debe ser única y el valor proporcionado a cada sentencia case debe ser del mismo tipo que el tipo de dato devuelto por la expresión proporcionada a la sentencia switch.

Otro punto de interés en la sentencia switch son las sentencias break después de cada case. La sentencia break hace que el control salga de la sentencia switch y continúe con la siguiente línea. La sentencia break es necesaria porque las sentencias case se siguen ejecutando hacia abajo. Esto es, sin un break explícito, el flujo de control seguiría secuencialmente a través de las sentencias case siguientes. En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia case a otra, por eso se han tenido que poner las sentencias break. Sin embargo, hay ciertos escenarios en los que querrás que el control proceda secuencialmente a través de las sentencias case. Como este código que calcula el número de días de un mes de acuerdo con el rítmico refrán que dice "Treinta tiene Septiembre...".

```
int mes;
int numeroDias;
. . .
switch (mes) {
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
    numeroDias = 31;
    break;
case 4:
case 6:
case 9:
case 11:
    numeroDias = 30;
    break;
case 2:
    if ( ((ano % 4 == 0) && !(ano % 100 == 0)) || ano % 400 == 0 )
        numeroDias = 29;
    else
        numeroDias = 28;
    break;
}
```

Finalmente, puede utilizar la sentencia default al final de la sentencia switch para manejar los valores que no se han manejado explícitamente por una de las sentencias case.

```
int mes;
. . .
switch (mes) {
case 1: System.out.println("Enero"); break;
case 2: System.out.println("Febrero"); break;
case 3: System.out.println("Marzo"); break;
case 4: System.out.println("Abril"); break;
case 5: System.out.println("Mayo"); break;
case 6: System.out.println("Junio"); break;
case 7: System.out.println("Julio"); break;
```

```

case 8: System.out.println("Agosto"); break;
case 9: System.out.println("Septiembre"); break;
case 10: System.out.println("Octubre"); break;
case 11: System.out.println("Noviembre"); break;
case 12: System.out.println("Diciembre"); break;
default: System.out.println("Ee, no es un mes válido!");
        break;
}

```

Sentencias de Bucle

Generalmente hablando, una sentencia while realiza una acción mientras se cumpla una cierta condición. La sintaxis general de la sentencia while es:

```

while (expresión)
    sentencia

```

Esto es, mientras la expresión sea verdadera, ejecutará la sentencia.

sentencia puede ser una sólo sentencia o puede ser un bloque de sentencias. Un bloque de sentencias es un juego de sentencias legales de java contenidas dentro de corchetes('{ 'y '}'). Por ejemplo, supón que además de incrementar contador dentro de un bucle while también quieres imprimir el contador cada vez que se lea un carácter. Podrías escribir esto en su lugar:

```

. . .
while (System.in.read() != -1) {
    contador++;
    System.out.println("Se ha leído un el carácter = " + contador);
}
. . .

```

Por convención el corchete abierto '{' se coloca al final de la misma línea donde se encuentra la sentencia while y el corchete cerrado '}' empieza una nueva línea indentada a la línea en la que se encuentra el while.

Además de while Java tiene otros dos constructores de bucles que puedes utilizar en tus programas: el bucle for y el bucle do-while.

Primero el bucle for. Puedes utilizar este bucle cuando conozcas los límites del bucle (su instrucción de inicialización, su criterio de terminación y su instrucción de incremento). Por ejemplo, el bucle for se utiliza frecuentemente para iterar sobre los elementos de un array, o los caracteres de una cadena.

```

// a es un array de cualquier tipo
. . .
int i;
int length = a.length;
for (i = 0; i < length; i++) {
    . . .
    // hace algo en el elemento i del array a
    . . .
}

```

Si sabes cuando estás escribiendo el programa que quieres empezar en el inicio del array, parar al final y utilizar cada uno de los elementos. Entonces la sentencia for es una buena elección. La forma general del bucle for puede expresarse así:

```

for (inicialización; terminación; incremento)
    sentencias

```

inicialización es la sentencia que inicializa el bucle -- se ejecuta una vez al iniciar el bucle.

terminación es una sentencia que determina cuando se termina el bucle. Esta expresión se evalúa al principio de cada iteración en el bucle. Cuando la expresión se evalúa a false el bucle se termina.

Finalmente, incremento es una expresión que se invoca en cada interacción del bucle. Cualquiera (o todos) de estos componentes pueden ser una sentencia vacía (un punto y coma).

Java proporciona otro bucle, el bucle do-while, que es similar al bucle while que se vio al principio, excepto en que la expresión se evalúa al final del bucle:

```
do {  
    sentencias  
} while (Expresión Booleana);
```

La sentencia do-while se usa muy poco en la construcción de bucles pero tiene sus usos. Por ejemplo, es conveniente utilizar la sentencia do-while cuando el bucle debe ejecutarse al menos una vez. Por ejemplo, para leer información de un fichero, sabemos que al menos debe leer un carácter:

```
int c;  
InputStream in;  
.  
.  
do {  
    c = in.read();  
    .  
    .  
} while (c != -1);
```

Sentencias de Manejo de Excepciones

Cuando ocurre un error dentro de un método Java, el método puede lanzar una excepción para indicar a su llamador que ha ocurrido un error y que el error está utilizando la sentencia throw. El método llamador puede utilizar las sentencias try, catch, y finally para capturar y manejar la excepción. Puedes ver [Manejar Errores Utilizando Excepciones](#) para obtener más información sobre el lanzamiento y manejo de excepciones.

Sentencias de Ruptura

Ya has visto la sentencia break en acción dentro de la sentencia switch anteriormente. Como se observó anteriormente, la sentencia break hace que el control del flujo salte a la sentencia siguiente a la actual.

Hay otra forma de break que hace que el flujo de control salte a una sentencia etiquetada. Se puede etiquetar una sentencia utilizando un identificador legal de Java (la etiqueta) seguido por dos puntos (:) antes de la sentencia:

```
SaltaAqui: algunaSentenciaJava
```

Para saltar a la sentencia etiquetada utilice esta forma de la sentencia break.

```
break SaltaAqui;
```

Las rupturas etiquetadas son una alternativa a la sentencia goto que no está soportada por el lenguaje Java.

Se puede utilizar la sentencia continue dentro de un bucle para saltar de la sentencia actual hacia el principio del bucle o a una sentencia etiquetada. Considera esta implementación del método indexOf() de la clase String que utiliza la forma de

continue que continúa en una sentencia etiquetada:

```
public int indexOf(String str, int fromIndex) {
    char[] v1 = value;
    char[] v2 = str.value;
    int max = offset + (count - str.count);
    test:
    for (int i = offset + ((fromIndex < 0) ? 0 : fromIndex); i <= max ; i++) {
        int n = str.count;
        int j = i;
        int k = str.offset;
        while (n-- != 0) {
            if (v1[j++] != v2[k++]) {
                continue test;
            }
        }
        return i - offset;
    }
    return -1;
}
```

Nota: Sólo se puede llamar a la sentencia continue desde dentro de un bucle.

Y finalmente la sentencia return. Esta sentencia se utiliza para salir del método actual y volver a la sentencia siguiente a la que originó la llamada en el método original. Existen dos formas de return: una que devuelve un valor y otra que no lo hace. Para devolver un valor, simplemente se pone el valor (o una expresión que calcule el valor) detrás de la palabra return:

```
return ++count;
```

El valor devuelto por return debe corresponder con el tipo del valor de retorno de la declaración del método.

Cuando un método se declara como void utiliza la forma de return que no devuelve ningún valor:

```
return;
```

Arrays y Cadenas

Al igual que otros lenguajes de programación, Java permite juntar y manejar múltiples valores a través de un objeto array (matriz). También se pueden manejar datos compuestos de múltiples caracteres utilizando el objeto String (cadena).

Arrays

Esta sección te enseñará todo lo que necesitas para crear y utilizar arrays en tus programas Java.

Como otras variables, antes de poder utilizar un array primero se debe declarar. De nuevo, al igual que otras variables, la declaración de un array tiene dos componentes primarios: el tipo del array y su nombre. Un tipo de array incluye el tipo de dato de los elementos que va contener el array. Por ejemplo, el tipo de dato para un array que sólo va a contener elementos enteros es un array de enteros. No puede existir un array de tipo de datos genérico en el que el tipo de sus elementos esté indefinido cuando se declara el array. Aquí tienes la declaración de un array de enteros:

```
int[] arrayDeEnteros;
```

La parte `int[]` de la declaración indica que `arrayDeEnteros` es un array de enteros. La declaración no asigna ninguna memoria para contener los elementos del array.

Si se intenta asignar un valor o acceder a cualquier elemento de `arrayDeEnteros` antes de haber asignado la memoria para él, el compilador dará un error como este y no compilará el programa:

```
testing.java:64: Variable arraydeenteros may not have been initialized.
```

Para asignar memoria a los elementos de un array, primero se debe ejemplarizar el array. Se puede hacer esto utilizando el operador `new` de Java. (Realmente, los pasos que se deben seguir para crear un array son similares a los se deben seguir para crear un objeto de una clase: declaración, ejemplarización e inicialización.

La siguiente sentencia asigna la suficiente memoria para que `arrayDeEnteros` pueda contener diez enteros.

```
int[] arraydeenteros = new int[10]
```

En general, cuando se crea un array, se utiliza el operador `new`, más el tipo de dato de los elementos del array, más el número de elementos deseados encerrado entre corchetes cuadrados (`[` y `]`).

```
TipodeElemento[] NombredeArray = new TipodeElementos[tamanoArray]
```

Ahora que se ha asignado memoria para un array ya se pueden asignar valores a los elementos y recuperar esos valores:

```
for (int j = 0; j < arrayDeEnteros.length; j++) {  
    arrayDeEnteros[j] = j;
```

```
        System.out.println("[j] = " + arrayDeEnteros[j]);  
    }  
}
```

Como se puede ver en el ejemplo anterior, para referirse a un elemento del array, se añade corchetes cuadrados al nombre del array. Entre los corchetes cuadrados se indica (bien con una variable o con una expresión) el índice del elemento al que se quiere acceder. Observa que en Java, el índice del array empieza en 0 y termina en la longitud del array menos uno.

Hay otro elemento interesante en el pequeño ejemplo anterior. El bucle for itera sobre cada elemento de arrayDeEnteros asignándole valores e imprimiendo esos valores. Observa el uso de arrayDeEnteros.length para obtener el tamaño real del array. length es una propiedad proporcionada para todos los arrays de Java.

Los arrays pueden contener cualquier tipo de dato legal en Java incluyendo los tipos de referencia como son los objetos u otros array. Por ejemplo, el siguiente ejemplo declara un array que puede contener diez objetos String.

```
String[] arrayDeStrings = new String[10];
```

Los elementos en este array son del tipo referencia, esto es, cada elemento contiene una referencia a un objeto String. En este punto, se ha asignado suficiente memoria para contener las referencias a los Strings, pero no se ha asignado memoria para los propios strings. Si se intenta acceder a uno de los elementos de arraydeStrings obtendrá una excepción 'NullPointerException' porque el array está vacío y no contiene ni cadenas ni objetos String. Se debe asignar memoria de forma separada para los objetos String:

```
for (int i = 0; i < arraydeStrings.length; i++) {  
    arraydeStrings[i] = new String("Hello " + i);  
}
```

Strings

Una secuencia de datos del tipo carácter se llama un string (cadena) y en el entorno Java está implementada por la clase String (un miembro del paquete java.lang).

```
String[] args;
```

Este código declara explícitamente un array, llamado args, que contiene objetos del tipo String. Los corchetes vacíos indican que la longitud del array no se conoce en el momento de la compilación, porque el array se pasa en el momento de la ejecución.

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas " y "):

```
"Hola mundo!"
```

El compilador asigna implícitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables - es decir, no se pueden modificar una vez que han sido creados. El paquete java.lang proporciona una clase diferente,

StringBuffer, que se podrá utilizar para crear y manipular caracteres al vuelo.

Concatenación de Cadenas

Java permite concatenar cadenas facilmente utilizando el operador +. El siguiente fragmento de código concatena tres cadenas para producir su salida:

```
"La entrada tiene " + contador + " caracteres."
```

Dos de las cadenas concatenadas son cadenas literales: "La entrada tiene " y " caracteres.". La tercera cadena - la del medio - es realmente un entero que primero se convierte a cadena y luego se concatena con las otras.

Crear Objetos

En Java, se crea un objeto mediante la creacción de un objeto de una clase o, en otras palabras, ejemplarizando una clase. Aprenderás cómo crear una clase más adelante en [Crear Clases](#). Hasta entonces, los ejemplos contenidos aquí crean objetos a apartir de clases que ya existen en el entorno Java.

Frecuentemente, se verá la creacción de un objeto Java con un sentencia como esta:

```
Date hoy = new Date();
```

Esta sentencia crea un objeto Date (Date es una clase del paquete java.util). Esta sentencia realmente realiza tres acciones: declaración, ejemplarización e inicialización. Date hoy es una declaración de variable que sólo le dice al compilador que el nombre hoy se va a utilizar para referirse a un objeto cuyo tipo es Date, el operador new ejemplariza la clase Date (creando un nuevo objeto Date), y Date() inicializa el objeto.

Declarar un Objeto

Ya que la declaración de un objeto es una parte innecesaria de la creacción de un objeto, las declaraciones aparecen frecuentemente en la misma línea que la creacción del objeto. Como cualquier otra declaración de variable, las declaraciones de objetos pueden aparecer solitarias como esta:

```
Date hoy;
```

De la misma forma, declarar una variable para contener un objeto es exactamente igual que declarar una variable que va a contener un tipo primitivo:

```
tipo nombre
```

donde tipo es el tipo de dato del objeto y nombre es el nombre que va a utilizar el objeto. En Java, las clases e interfaces son como tipos de datos. Entonces tipo puede ser el nombre de una clase o de un interface.

Las declaraciones notifican al compilador que se va a utilizar nombre para referirse a una variable cuyo tipo es tipo. Las declaraciones no crean nuevos objetos. Date hoy no crea un objeto Date, sólo crea un nombre de variable para contener un objeto Date. Para ejemplarizar la clase Date, o cualquier otra clase, se utiliza el operador new.

Ejemplarizar una Clase

El operador new ejemplariza una clase mediante la asignación de memoria para el objeto nuevo de ese tipo. new necesita un sólo argumento: una llamada al método constructor. Los métodos

constructores son métodos especiales proporcionados por cada clase Java que son reponsables de la inicialización de los nuevos objetos de ese tipo. El operador new crea el objeto, el constructor lo inicializa.

Aquí tienes un ejemplo del uso del operador new para crear un objeto Rectangle (Rectangle es una clase del paquete java.awt):

```
new Rectangle(0, 0, 100, 200);
```

En el ejemplo, Rectangle(0, 0, 100, 200) es una llamada al constructor de la clase Rectangle.

El operador new devuelve una referencia al objeto recién creado. Esta referencia puede ser asignada a una variable del tipo apropiado.

```
Rectangle rect = new Rectangle(0, 0, 100, 200);
```

(Recuerda que una clase esencialmente define un tipo de dato de referencia. Por eso, Rectangle puede utilizarse como un tipo de dato en los programas Java. El valor de cualquier variable cuyo tipo sea un tipo de referencia, es una referencia (un puntero) al valor real o conjunto de valores representado por la variable.

Inicializar un Objeto

Como mencioné anteriormente, las clases porporcionan métodos constructores para incializar los nuevos objetos de ese tipo. Una clase podría proporcionar múltiples constructores para realizar diferentes tipos de inicialización en los nuevos objetos. Cuando veas la implementación de una clase, reconocerás los constructores porque tienen el mismo nombre que la clase y no tienen tipo de retorno. Recuerda la creacción del objeto Date en el sección inicial. El constructor utilizado no tenía ningún argumento:

```
Date()
```

Un constructor que no tiene ningún argumento, como el mostrado arriba, es conocido como constructor por defecto. Al igual que Date, la mayoría de las clases tienen al menos un constructor, el constructor por defecto.

Si una clase tiene varios constructores, todos ellos tienen el mismo nombre pero se deben diferenciar en el número o el tipo de sus argmentos. Cada constructor inicializa el nuevo objeto de una forma diferente. Junto al constructor por defecto, la clase Date proporciona otro constructor que inicializa el nuevo objeto con un nuevo año, mes y día:

```
Date cumpleaños = new Date(1963, 8, 30);
```

El compilador puede diferenciar los constructores a través del tipo y del número de sus argumentos.

Utilizar Objetos

Una vez que se ha creado un objeto, probablemente querrás hacer algo con él. Supón, por ejemplo, que después de crear un nuevo rectángulo, quieres moverlo a una posición diferente (es decir, el rectángulo es un objeto en un programa de dibujo y el usuario quiere moverlo a otra posición de la página).

La clase `Rectangle` proporciona dos formas equivalentes de mover el rectángulo:

1. Manipular directamente las variables `x` e `y` del objeto.
2. Llamar el método `move()`.

La opción 2 se considera "más orientada a objetos" y más segura porque se manipulan las variables del objeto indirectamente a través de una capa protectora de métodos, en vez de manejarlas directamente. Manipular directamente las variables de un objeto se considera propenso a errores; se podría colocar el objeto en un estado de inconsistencia. Sin embargo, una clase no podría (y no debería) hacer que sus variables estuvieran disponibles para la manipulación directa por otros objetos, si fuera posible que esas manipulaciones situaran el objeto en un estado de inconsistencia. Java proporciona un mecanismo mediante el que las clases puede restringir o permitir el acceso a sus variables y métodos a otros objetos de otros tipos.

Esta sección explica la llamada a métodos y la manipulación de variables que se han hecho accesibles a otras clases. Para aprender más sobre el control de acceso a miembros puedes ir [Controlar el Acceso a Miembros de una Clase](#).

Las variables `x` e `y` de `Rectangle` son accesibles desde otras clases. Por eso podemos asumir que la manipulación directa de estas variables es segura.

Referenciar Variables de un Objeto

Primero, enfoquemos cómo inspeccionar y modificar la posición del rectángulo mediante la manipulación directa de las variables `x` e `y`. La siguiente sección mostrará como mover el rectángulo llamando al método `move()`.

Para acceder a las variables de un objeto, sólo se tiene que añadir el nombre de la variable al del objeto referenciado introduciendo un punto en el medio (`.`).

`objetoReferenciado.variable`

Supón que tienes un rectángulo llamado `rect` en tu programa. puedes acceder a las variables `x` e `y` con `rect.x` y `rect.y`, respectivamente. Ahora que ya tienes un nombre para las variables de `rect`, puedes utilizar ese nombre en sentencias y expresiones Java como si fueran nombres de variables "normales". Así, para mover el rectángulo a una nueva posición podrías escribir:

```
rect.x = 15;           // cambia la posición x
rect.y = 37;           // cambia la posición y
```

La clase Rectangle tiene otras dos variables--width y height--que son accesibles para objetos fuera de Rectangle. Se puede utilizar la misma notación con ellas: rect.width y rect.height. Entonces se puede calcular el área del rectángulo utilizando esta sentencia:

```
area = rect.height * rect.width;
```

Cuando se accede a una variable a través de un objeto, se está refiriendo a las variables de un objeto particular. Si cubo fuera también un rectángulo con una altura y anchura diferentes de rect, esta instrucción:

```
area = cubo.height * cubo.width;
```

calcula el área de un rectángulo llamado cubo y dará un resultado diferente que la instrucción anterior (que calculaba el área de un rectángulo llamado rect).

Observa que la primera parte del nombre de una variable de un objeto (el **objetoReferenciado** en **objetoReferenciado.variable**) debe ser una referencia a un objeto. Como se puede utilizar un nombre de variable aquí, también se puede utilizar en cualquier expresión que devuelva una referencia a un objeto. Recuerda que el operador new devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por new para acceder a las variables del nuevo objeto:

```
height = new Rectangle().height;
```

Llamar a Métodos de un Objeto

Llamar a un método de un objeto es similar a obtener una variable del objeto. Para llamar a un método del objeto, simplemente se añade al nombre del objeto referenciado el nombre del método, separados por un punto ('.'), y se proporcionan los argumentos del método entre paréntesis. Si el método no necesita argumentos, se utilizan los paréntesis vacíos.

```
objetoReferenciado.nombreMétodo(listaArgumentos);
○
objetoReferenciado.nombreMétodo();
```

Veamos que significa esto en términos de movimiento del rectángulo. Para mover rect a una nueva posición utilizando el método move() escribe esto:

```
rect.move(15, 37);
```

Esta sentencia Java llama al método move() de rect con dos parámetros enteros, 15 y 37. Esta sentencia tiene el efecto de mover el objeto rect igual que se hizo en las sentencias anteriores en las que se

modificaban directamente los valores x e y del objeto:

```
rect.x = 15;  
rect.y = 37;
```

Si se quiere mover un rectángulo diferente, uno llamado cubo, la nueva posición se podría escribir:

```
cubo.move(244, 47);
```

Como se ha visto en estos ejemplos, las llamadas a métodos se hacen directamente a un objeto específico; el objeto especificado en la llamada al método es el que responde a la instrucción.

Las llamadas a métodos también se conocen como mensajes. Como en la vida real, los mensajes se deben dirigir a un receptor particular. Se pueden obtener distintos resultados dependiendo del receptor de su mensaje. En el ejemplo anterior, se ha enviado el mensaje `move()` al objeto llamado `rect` para que éste mueva su posición. Cuando se envía el mensaje `move()` al objeto llamado `cubo`, el que se mueve es `cubo`. Son resultados muy distintos.

Una llamada a un método es una expresión (puedes ver [Expresiones](#) para más información) y evalúa a algún valor. El valor de una llamada a un método es su valor de retorno, si tiene alguno. Normalmente se asignará el valor de retorno de un método a una variable o se utilizará la llamada al método dentro del ámbito de otra expresión o sentencia.

El método `move()` no devuelve ningún valor (está declarado como `void`). Sin embargo, el método `inside()` de `Rectangle` sí lo hace. Este método toma dos coordenadas x e y, y devuelve `true` si este punto está dentro del rectángulo. Se puede utilizar el método `inside()` para hacer algo especial en algún punto, como decir la posición del ratón cuando está dentro del rectángulo:

```
if (rect.inside(mouse.x, mouse.y)) {  
    . . .  
    // ratón dentro del rectángulo  
    . . .  
} else {  
    . . .  
    // ratón fuera del rectángulo  
    . . .  
}
```

Recuerda que una llamada a un método es un mensaje al objeto nombrado. En este caso, el objeto nombrado es `rect`. Entonces:

```
rect.inside(mouse.x, mouse.y)
```

le pregunta a `rect` si la posición del cursor del ratón se encuentra entre las coordenadas `mouse.x` y `mouse.y`. Se podría obtener una respuesta diferente si envía el mismo mensaje a `cubo`.

Como se explicó [anteriormente](#), el objetoReferenciado en la llamada al método objetoReferenciado.método() debe ser una referencia a un objeto. Como se puede utilizar un nombre de variable aquí, también se puede utilizar en cualquier expresión que devuelva una referencia a un objeto. Recuerda que el operador new devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por new para acceder a las variables del nuevo objeto:

```
new Rectangle(0, 0, 100, 50).equals(anotherRect)
```

La expresión new Rectangle(0, 0, 100, 50) evalúa a una referencia a un objeto que se refiere a un objeto Rectangle. Entonces, como verás, se puede utilizar la notació de punto ('.') para llamar al método equals() del nuevo objeto Rectangle para determinar si el rectángulo nuevo es igual al especificado en la lista de argumentos de equals().

Eliminar Objetos no Utilizados

Muchos otros lenguajes orientados a objetos necesitan que se siga la pista de los objetos que se han creado y luego se destruyan cuando no se necesiten. Escribir código para manejar la memoria de esta forma es aburrido y propenso a errores. Java permite ahorrarse esto, permitiendo crear tantos objetos como se quiera (sólo limitados por los que el sistema pueda manejar) pero nunca tienen que ser destruidos. El entorno de ejecución Java borra los objetos cuando determina que no se van a utilizar más. Este proceso es conocido como recolección de basura.

Un objeto es elegible para la recolección de basura cuando no existen más referencias a ese objeto. Las referencias que se mantienen en una variable desaparecen de forma natural cuando la variable sale de su ámbito. O cuando se borra explícitamente un objeto referencia mediante la selección de un valor cuyo tipo de dato es una referencia a null.

Recolector de Basura

El entorno de ejecución de Java tiene un recolector de basura que periódicamente libera la memoria ocupada por los objetos que no se van a necesitar más. El recolector de basura de Java es un barredor de marcas que escanea dinámicamente la memoria de Java buscando objetos, marcando aquellos que han sido referenciados. Después de investigar todos los posibles paths de los objetos, los que no están marcados (esto es, no han sido referenciados) se les conoce como basura y son eliminados.

El colector de basura funciona en un thread (hilo) de baja prioridad y funciona tanto síncrona como asíncronamente dependiendo de la situación y del sistema en el que se esté ejecutando el entorno Java.

El recolector de basura se ejecuta síncronamente cuando el sistema funciona fuera de memoria o en respuesta a una petición de un programa Java. Un programa Java le puede pedir al recolector de basura que se ejecute en cualquier momento mediante una llamada a `System.gc()`.

Nota: Pedir que se ejecute el recolector de basura no garantiza que los objetos sean recolectados.

En sistemas que permiten que el entorno de ejecución Java note cuando un thread ha empezado a interrumpir a otro thread (como Windows 95/NT), el recolector de basura de Java funciona asíncronamente cuando el sistema está ocupado. Tan pronto como otro thread se vuelva activo, se pedirá al recolector de basura que obtenga un estado consistente y termine.

Finalización

Antes de que un objeto sea recolectado, el recolector de basura le da una oportunidad para limpiarse él mismo mediante la llamada al método `finalize()` del propio objeto. Este proceso es conocido como finalización.

Durante la finalización un objeto se podrían liberar los recursos del sistema como son los ficheros, etc y liberar referencias en otros objetos para hacerse elegible por la recolección de basura.

El método `finalize()` es un miembro de la clase `java.lang.Object`. Una clase debe sobrescribir el método `finalize()` para realizar cualquier finalización necesaria para los objetos de ese tipo.

Declarar Clases

Ahora que ya sabemos como crear, utilizar y destruir objetos, es hora de aprender cómo escribir clases de las que crear esos objetos.

Una clase es un proyecto o prototipo que se puede utilizar para crear muchos objetos. La implementación de una clase comprende dos componentes: la declaración y el cuerpo de la clase:

```
DeclaraciónDeLaClase {
    CuerpoDeLaClase
}
```

La Declaración de la Clase

Como mínimo, la declaración de una clase debe contener la palabra clave `class` y el nombre de la clase que está definiendo. Así la declaración más sencilla de una clase se parecería a esto:

```
class NombredeClase {
    . . .
}
```

Por ejemplo, esta clase declara una nueva clase llamada `NumeroImaginario`:

```
class NumeroImaginario {
    . . .
}
```

Los nombres de las clases deben ser un identificador legal de Java y, por convención, deben empezar por una letra mayúscula. Muchas veces, todo lo que se necesitará será una declaración mínima. Sin embargo, la declaración de una clase puede decir más cosas sobre la clase. Más específicamente, dentro de la declaración de la clase se puede:

- declarar cual es la superclase de la clase.
- listar los interfaces implementados por la clase
- declarar si la clase es pública, abstracta o final

Declarar la Superclase de la Clase

En Java, todas las clases tienen una superclase. Si no se especifica una superclase para una clase, se asume que es la clase `Object` (declarada en `java.lang`). Entonces la superclase de `NumeroImaginario` es `Object` porque la declaración no explicitó ninguna otra clase. Para obtener más información sobre la clase `Object`, puede ver [La clase Object](#).

Para especificar explícitamente la superclase de una clase, se debe poner la palabra clave `extends` más el nombre de la superclase entre el nombre de la clase que se ha creado y el corchete abierto que abre el cuerpo de la clase, así:

```
class NombredeClase extends NombredeSuperClase {
    . . .
}
```

Por ejemplo, supón que quieres que la superclase de `NumeroImaginario` sea la clase `Number` en vez de la clase `Object`. Se podría escribir esto:

```
class NumeroImaginario extends Number {
    . . .
}
```

Esto declara explícitamente que la clase `Number` es la superclase de `NumeroImaginario`. (La clase `Number` es parte del paquete `java.lang` y es la base para los enteros, los números en coma flotante y otros números).

Declarar que `Number` es la superclase de `NumeroImaginario` declara implícitamente que `NumeroImaginario` es una subclase de `Number`. Una subclase hereda las variables y los métodos de su superclase.

Crear una subclase puede ser tan sencillo como incluir la cláusula `extends` en su declaración de clase. Sin embargo, se tendrán que hacer otras provisiones en su código cuando se crea una subclase, como sobrescribir métodos. Para obtener más información sobre la creación de subclases, puede ver [Subclases, Superclases, y Herencia](#).

Listar los Interfaces Implementados por la Clase

Cuando se declara una clase, se puede especificar que interface, si lo hay, está implementado por la clase. Pero, ¿Qué es un interface? Un interface declara un conjunto de métodos y constantes sin especificar su implementación para ningún método. Cuando una clase exige la implementación de un interface, debe proporcionar la implementación para todos los métodos declarados en el interface.

Para declarar que una clase implementa uno o más interfaces, se debe utilizar la palabra clave `implements` seguida por una lista de los interfaces implementados por la clase delimitada por comas. Por ejemplo, imagina un interface llamado `Aritmetico` que define los métodos llamados `suma()`, `resta()`, etc... La clase `NumeroImaginario` puede declarar que implementa el interface `Aritmetico` de esta forma:

```
class NumeroImaginario extends Number implements Aritmetico {  
    . . .  
}
```

se debe garantizar que propociona la implementación para los métodos `suma()`, `resta()` y demás métodos declarados en el interface `Aritmetico`. Si en `NumeroImaginario` falta alguna implementación de los métodos definidos en `Aritmetico`, el compilador mostrará un mensaje de error y no compilará el programa:

```
nothing.java:5: class NumeroImaginario must be declared abstract. It does not define  
java.lang.Number add(java.lang.Number, java.lang.Number) from interface Aritmetico.  
class NumeroImaginario extends Number implements Aritmetico {  
    ^
```

Por convención, la cláusula `implements` sigue a la cláusula `extends` si ésta existe.

Observa que las firmas de los métodos declarados en el interface `Aritmetico` deben corresponder con las firmas de los métodos implementados en la clase `NumeroImaginario`. Tienes más información sobre cómo crear y utilizar interfaces en [Crear y Utilizar Interfaces](#).

Clases Public, Abstract, y Final

Se puede utilizar uno de estos tres modificadores en una declaración de clase para declarar que esa clase es pública, abstracta o final. Los modificadores van delante de la palabra clave `class` y son opcionales.

El modificador `public` declara que la clase puede ser utilizada por objetos que estén fuera del paquete actual. Por defecto, una clase sólo puede ser utiliza por otras clases del mismo paquete en el que están declaradas.

```
public class NumeroImaginario extends Number implements Aritmetico {  
    . . .  
}
```

Por convención, cuando se utiliza la palabra `public` en una declaración de clase debemos asegurarnos de que es el primer item de la declaración.

El modificador `abstract` declara que la clase es una clase abstracta. Una clase abstracta podría contener métodos abstractos (métodos sin implementación). Una clase abstracta está diseñada para ser una superclase y no puede ejemplarizarse. Para una discusión sobre las clases abstractas y cómo escribirlas puedes ver [Escribir Clases y Métodos Abstractos](#).

Utilizando el modificador `final` se puede declarar que una clase es final, que no puede tener subclases. Existen (al menos) dos razones por las que se podría querer hacer esto: razones de seguridad y razones de diseño. Para un mejor explicaión sobre las clases finales puedes ver

[Escribir Clases y Métodos Finales.](#)

Observa que no tiene sentido para una clase ser abstracta y final. En otras palabras, una clase que contenga métodos no implementados no puede ser final. Intentar declarar una clase como final y abstracta resultará en un error en tiempo de compilación.

Sumario de la Daclaración de una Clase

En suma, una declaración de clase se parecería a esto:

```
[ modificadores ] class NombredeClase [ extends NombredeSuperclase ]  
[ implements NombredeInterface ] {  
    . . .  
}
```

Los puntos entre [y] son opcionales. Una declaración de clase define los siguientes aspectos de una clase:

- modificadores declaran si la clase es abstracta, pública o final.
- NombredeClase selecciona el nombre de la clase que está declarando
- NombredeSuperClase es el nombre de la superclase de NombredeClase
- NombredeInterface es una lista delimitada por comas de los interfaces implementados por NombredeClase

De todos estos items, sólo la palabra clave class y el nombre de la clase son necesarios. Los otros son opcionales. Si no se realiza ninguna declaración explícita para los items opcionales, el compilador Java asume ciertos valores por defecto (una subclase de Object no final, no pública, no abstracta y que no implementa interfaces).

El Cuerpo de la Clase

Anteriormente se vió una descripción general de la implementación de una clase:

```
DeclaraciondeClase {  
    CuerpodeClase  
}
```

La [página anterior](#) describe todos los componentes de la declaración de una clase. Esta página describe la estructura general y la organización del cuerpo de la clase.

El cuerpo de la clase compone la implementación de la propia clase y contiene dos secciones diferentes: la declaración de variables y la de métodos. Una variable miembro de la clase representa un estado de la clase y sus métodos implementan el comportamiento de la clase. Dentro del cuerpo de la clase se definen todas las variables miembro y los métodos soportados por la clase.

Típicamente, primero se declaran las variables miembro de la clase y luego se proporciona las declaraciones e implementaciones de los métodos, aunque este orden no es necesario.

```
DeclaracióndeClase {  
    DeclaracionesdeVariablesMiembros  
    DeclaracionesdeMétodos  
}
```

Aquí tienes una pequeña clase que declara tres variables miembro y un método:

```
class Ticket {  
    Float precio;  
    String destino;  
    Date fechaSalida;  
    void firma(Float forPrecio, String forDestino, Date forFecha) {  
        precio = forPrecio;  
        destino = forDestino;  
        fechaSalida = forFecha;  
    }  
}
```

Para más información sobre cómo declarar variables miembro, puedes ver [Declarar Variables Miembro](#). Y para obtener más información sobre cómo implementar métodos, puedes ver [Implementar Métodos](#).

Además de las variables miembro y los métodos que se declaran explícitamente dentro del cuerpo de la clase, una clase puede heredar algo de su superclase. Por ejemplo, todas las clases del entorno Java son una descendencia (directa o indirecta) de la clase Object. La clase Object define el estado básico y el comportamiento que todos los objetos deben tener como habilidad para comparar unos objetos con otros, para convertir una cadena, para esperar una condición variable, para notificar a otros objetos que una condición variable ha cambiado,

etc... Así, como descendentes de esta clase, todos los objetos del entorno Java heredan sus comportamientos de la clase Object.

Ozito

Declarar Variables Miembro

Como mínimo, una declaración de variable miembro tiene dos componentes: el tipo de dato y el nombre de la variable.

```
tipo nombreVariable;           // Declaración mínima de una variable miembro
```

Una declaración mínima de variable miembro es como la declaración de variables que se escribe en cualquier otro lugar de un programa Java, como las variables locales o parámetros de métodos. El siguiente código declara una variable miembro entera llamada unEntero dentro de la clase ClaseEnteros.

```
class ClaseEnteros {
    int unEntero;

    . . .
    // define los métodos aquí
    . . .
}
```

Observa que la declaración de variables miembro aparece dentro de la implementación del cuerpo de la clase pero no dentro de un método. Este posicionamiento dentro del cuerpo de la clase determina que una variable es una variable miembro.

Al igual que otras variables en Java, las variables miembro deben tener un tipo. Un tipo de variable determina los valores que pueden ser asignados a las variables y las operaciones que se pueden realizar con ellas. Ya deberías estar familiarizado con los tipos de datos en Java mediante la lectura de la lección anterior: [Variables y Tipos de Datos](#).

Un nombre de una variable miembro puede ser cualquier identificador legal de Java y por convención empieza con una letra minúscula (los nombres de clase típicamente empiezan con una letra mayúscula). No se puede declarar más de una variable con el mismo nombre en la misma clase. Por ejemplo, el siguiente código es legal:

```
class ClaseEnteros {
    int unEntero;
    int unEntero() {           // un método con el mismo nombre que una variable
        . . .
    }
}
```

Junto con el nombre y el tipo, se pueden especificar varios atributos para las variables miembro cuando se las declara: incluyendo si los objetos pueden acceder a la variable, si la variable es una variable de clase o una variable de ejemplar, y si la variable es una constante.

Una declaración de variable se podría parecer a esto:

```
[especificadordeAcceso] [static] [final] [transient] [volatile] tipo nombredeVariable
```

Los puntos entre [y] son opcionales. Lo items en negrita se deben reemplazar por palabras clave o por nombres.

Una declaración de variable miembro define los siguientes aspectos de la variable:

- **especificadordeAcceso** define si otras clases tienen acceso a la variable. Se puede controlar el acceso a los métodos utilizando los mismos especificadores, por eso [Controlar el Acceso a Variables Miembro de una Clase](#) cubre cómo se puede controlar el acceso a las variables miembro o los métodos.
- **static** indica que la variable es una variable miembro de la clase en oposición a una variable miembro del ejemplar. Se puede utilizar static para declarar métodos de clase. [Miembros de Clase y de Ejemplar](#) explica la declaración de variables de clase y de ejemplar y escribir métodos de ejemplar o de clase.
- **final** indica que la variable es una constante

- transient la variable no es una parte persistente del estado del objeto
- volatile significa que la variable es modificada de forma asíncrona.

La explicación de las variables final, transient, y volatile viene ahora:

Declarar Constantes

Para crear una variable miembro constante en Java se debe utilizar la palabra clave final en su declaración de variable. La siguiente declaración define una constante llamada AVOGADRO cuyo valor es el número de Avogadro (6.023×10^{23}) y no puede ser cambiado:

```
class Avo {
    final double AVOGADRO = 6.023e23;
}
```

Por convención, los nombres de los valores constantes se escriben completamente en mayúsculas. Si un programa intenta cambiar una variable, el compilador muestra un mensaje de error similar al siguiente, y rehusa a compilar su programa.

```
AvogadroTest.java:5: Can't assign a value to a final variable: AVOGADRO
```

Declarar Variables Transitorias

Por defecto, las variables miembro son una parte persistente del estado de un objeto. Las variables que forman parte persistente del estado del objeto debe guardarse cuando el objeto se archiva. Se puede utilizar la palabra clave transient para indicar a la máquina virtual Java que la variable indicada no es una parte persistente del objeto.

Al igual que otros modificadores de variables en el sistema Java, se puede utilizar transient en una clase o declaración de variable de ejemplar como esta:

```
class TransientExample {
    transient int hobo;
    . . .
}
```

Este ejemplo declara una variable entera llamada hobo que no es una parte persistente del estado de la clase TransientExample.

Declarar Variables Volátiles

Si una clase contiene una variable miembro que es modificada de forma asíncrona, mediante la ejecución de threads concurrentes, se puede utilizar la palabra clave volatile de Java para notificar esto al sistema Java.

La siguiente declaración de variable es un ejemplo de como declarar que una variable va a ser modificada de forma asíncrona por threads concurrentes:

```
class VolatileExample {
    volatile int contador;
    . . .
}
```

Implementación de Métodos

Similarmente a la implementación de una clase, la implementación de un método consiste en dos partes, la declaración y el cuerpo del método.

```
declaracióndeMétodo {  
    cuerpodemétodo  
}
```

La Declaración de Método

Una declaración de método proporciona mucha información sobre el método al compilador, al sistema en tiempo de ejecución y a otras clases y objetos. Junto con el nombre del método, la declaración lleva información como el tipo de retorno del método, el número y el tipo de los argumentos necesarios, y qué otras clases y objetos pueden llamar al método.

Los únicos elementos necesarios una declaración de método son el nombre y el tipo de retorno del método. Por ejemplo, el código siguiente declara un método llamado `estaVacio()` en la clase `Pila` que devuelve un valor booleano (`true` o `false`):

```
class Pila {  
    . . .  
    boolean estaVacio() {  
        . . .  
    }  
}
```

Devolver un Valor desde un Método

Java necesita que un método declare el tipo de dato del valor que devuelve. Si un método no devuelve ningún valor, debe ser declarado para devolver `void` (nulo).

Los métodos pueden devolver tipos de datos primitivos o tipos de datos de referencia. El método `estaVacio()` de la clase `Pila` devuelve un tipo de dato primitivo, un valor booleano:

```
class Pila {  
    static final int PILA_VACIA = -1;  
    Object[] stackelements;  
    int topelement = PILA_VACIA;  
    . . .  
    boolean estaVacio() {  
        if (topelement == PILA_VACIA)  
            return true;  
        else  
            return false;  
    }  
}
```

Sin embargo, el método `pop` de la clase `PILA` devuelve un tipo de dato de referencia: un objeto.

```
class Pila {  
    static final int PILA_VACIA = -1;  
    Object[] stackelements;  
    int topelement = PILA_VACIA;  
    . . .
```

```

Object pop() {
    if (topelement == PILA_VACIA)
        return null;
    else {
        return stackelements[topelement--];
    }
}
}

```

Los métodos utilizan el operador return para devolver un valor. Todo método que no sea declarado como void debe contener una sentencia return.

El tipo de dato del valor devuelto por la sentencia return debe corresponder con el tipo de dato que el método tiene que devolver; no se puede devolver un objeto desde un método que fue declarado para devolver un entero.

Cuando se devuelva un objeto, el tipo de dato del objeto devuelto debe ser una subclase o la clase exacta indicada. Cuando se devuelva un tipo interface, el objeto retornado debe implementar el interface especificado.

Un Nombre de Método

Un nombre de método puede ser cualquier indentificador legal en Java. Existen tres casos especiales a tener en cuenta con los nombres de métodos:

1. Java soporta la sobrecarga de métodos, por eso varios métodos pueden compartir el mismo nombre. Por ejemplo, supón que se ha escrito una clase que puede proporcionar varios tipos de datos (cadenas, enteros, etc...) en un área de dibujo. Se podría escribir un método que supiera como tratar a cada tipo de dato. En otros lenguajes, se tendría que pensar un nombre distinto para cada uno de los métodos. `dibujaCadena()`, `dibujaEntero`, etc... En Java, se puede utilizar el mismo nombre para todos los métodos pasándole un tipo de parámetro diferente a cada uno de los métodos. Entonces en la clase de dibujo, se podrán declarar tres métodos llamados `draw<>()` y que cada uno aceptara un tipo de parámetro diferente:

```

class DibujodeDatos {
    void draw(String s) {
        . . .
    }
    void draw(int i) {
        . . .
    }
    void draw(float f) {
        . . .
    }
}

```

Nota: La información que hay dentro de los paréntesis de la declaración son los argumentos del método. Los argumentos se cubren en la siguiente página: [Pasar Información a un Método](#).

Los métodos son diferenciados por el compilador basándose en el número y tipo de sus argumentos. Así `draw(String s)` y `draw(int i)` son métodos distintos y únicos. No se puede declarar un método con la misma firma: `draw(String s)` y `draw(String t)` son idénticos y darán un error del compilador.

Habrás observado que los métodos sobrecargados deben devolver el mismo tipo de

dato, por eso `void draw(String s)` y `int draw(String t)` declarados en la misma clase producirán un error en tiempo de compilación.

2. Todo método cuyo nombre sea igual que el de su clase es un constructor y tiene una tarea especial que realizar. Los constructores se utilizan para inicializar un objeto nuevo del tipo de la clase. Los constructores sólo pueden ser llamados con el operador `new`. Para aprender cómo escribir un constructor, puedes ver [Escribir un Método Constructor](#).
3. Una clase puede sobrescribir un método de sus superclases. El método que sobrescribe debe tener el mismo, nombre, tipo de retorno y lista de parámetros que el método al que ha sobrescrito. [Sobrescribir Métodos](#) te enseñará como sobrescribir los métodos de una superclase.

Características Avanzadas de la Declaración de Métodos

Junto con los dos elementos necesarios, una declaración de método puede contener otros elementos. Estos elementos declaran los argumentos aceptados por el método, si el método es un método de clase, etc...

Juntándolo todo, una declaración de método se parecería a esto:

```
[especificadordeAcceso] [static] [abstract] [final] [native] [synchronized]  
tipodeRetorno nombredelMétodo ([listadeparámetros]) [throws listadeExcepciones]
```

Cada uno de estos elementos de una declaración se cubre en alguna parte de este tutorial.

Pasar Información a un Método.

Cuando se escribe un método, se declara el número y tipo de los argumentos requeridos por ese método. Esto se hace en la firma del método. Por ejemplo, el siguiente método calcula el pago mensual de una hipoteca basándose en la cantidad prestada, el interés, la duración de la hipoteca (número de meses) y el valor futuro de la hipoteca (presumiblemente el valor futuro sea cero, porque al final de la hipoteca, ya la habrás pagado):

```
double hipoteca(double cantidad, double interes, double valorFinal, int numPeriodos)
{
    double I, parcial1, denominador, respuesta;

    I = interes / 100.0;
    parcial1 = Math.pow((1 + I), (0.0 - numPeriodos));
    denominador = (1 - parcial1) / I;
    respuesta = ((-1 * cantidad) / denominador) - ((valorFinal * parcial1) /
denominador);
    return respuesta;
}
```

Este método toma cuatro argumentos: la cantidad prestada, el interés, el valor futuro y el número de meses. Los tres primeros son números de coma flotante de doble precisión y el cuarto es un entero.

Al igual que este método, el conjunto de argumentos de cualquier método es una lista de declaraciones de variables delimitadas por comas donde cada declaración de variable es un par tipo/nombre:

tipo nombre

Como has podido ver en el ejemplo anterior, sólo tienes que utilizar el nombre del argumento para referirte al valor del argumento.

Tipos de Argumentos

En Java, se puede pasar como argumento a un método cualquier tipo de dato válido en Java. Esto incluye tipos primitivos, como enteros, dobles, etc.. y tipos de referencia como arrays, objetos, etc...

Aquí tienes un ejemplo de un constructor que acepta una array como argumento. En este ejemplo el constructor inicializa un objeto Polygon a partir de una lista de puntos (Point es una clase del paquete java.awt que representa una coordenada xy):

```
Polygon polygonFrom(Point[] listadePuntos) {
    . . .
}
```

Al contrario que en otros lenguajes, no se puede pasar un método a un método Java. Pero si se podría pasar un objeto a un método y luego llamar a los métodos del objeto.

Nombres de Argumentos

Cuando se declara un argumento para un método Java, se proporciona el nombre para ese argumento. Este nombre es utilizado dentro del cuerpo del método para referirse al valor del argumento.

Un argumento de un método puede tener el mismo nombre que una variable de la clase. En este caso, se dice que el argumento oculta a la variable miembro. Normalmente los argumentos que ocultan una variable miembro se utilizan en los constructores para inicializar una clase. Por ejemplo, observa la clase Circle y su constructor:

```
class Circle {
    int x, y, radius;
    public Circle(int x, int y, int radius) {
        . . .
    }
}
```

La clase Circle tiene tres variables miembro x, y y radius. Además, el constructor de la clase Circle acepta tres argumentos cada uno de los cuales comparte el nombre con la variable miembro para la que el argumento proporciona un valor inicial.

Los nombres de argumentos ocultan los nombres de las variables miembro. Por eso utilizar x, y o radius dentro del cuerpo de la función, se refiere a los argumentos, no a las variables miembro. Para acceder a las variables miembro, se debe referenciarlas a través de this--el objeto actual.

```
class Circle {
    int x, y, radius;
    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

Los nombres de los argumentos de un método no pueden ser el mismo que el de otros argumentos del mismo método, el nombre de cualquier variable local del método o el nombre de cualquier parámetro a una cláusula catch() dentro del mismo método.

Paso por Valor

En los métodos Java, los argumentos son pasados por valor. Cuando se le llama, el método recibe el valor de la variable pasada. Cuando el argumento es de un tipo primitivo, pasar por valor significa que el método no puede cambiar el valor. Cuando el argumento es del tipo de referencia, pasar por valor significa que el método no puede cambiar el objeto referenciado, pero sí puede invocar a los métodos del objeto y puede modificar las variables accesibles dentro del objeto.

Consideremos esta serie de sentencias Java que intentan recuperar el color actual de un objeto Pen en una aplicación gráfica:

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r + ", green = " + g + ", blue = " + b);
. . .
```

En el momento que se llama al método getRGBColor(), las variables r, g, y b tienen un valor de -1. El llamador espera que el método getRGBColor() le devuelva los valores de rojo, verde y azul para el color actual en las variables r, g, y b.

Sin embargo, el sistema Java pasa los valores de las variables(-1) al método getRGBColor(); no una referencia a las variables r, g, y b. Con esto se podría visualizar la llamada a getRGBColor() de esta forma: getRGBColor(-1, -1, -1).

Cuando el control pasa dentro del método getRGBColor(), los argumentos entran dentro del ámbito (se les asigna espacio) y son inicializados a los valores pasados al método:

```
class Pen {
    int valorRojo, valorVerde, valorAzul;
    void getRGBColor(int rojo, int verde, int azul) {
        // rojo, verde y azul han sido creados y sus valores son -1
    }
}
```



```

    . . .
}
}

```

Con esto `getRGBColor()` obtiene acceso a los valores de r, g, y b del llamador a través de sus argumentos rojo, verde, y azul, respectivamente. El método obtiene su propia copia de los valores para utilizarlos dentro del ámbito del método. Cualquier cambio realizado en estas copias locales no serán reflejados en las variables originales del llamador.

Ahora veremos la implementación de `getRGBColor()` dentro de la clase `Pen` que implicaba la firma de método anterior:

```

class Pen {
    int valorRojo, valorVerde, valorAzul;
    . . .
    // Este método no trabaja como se espera
    void getRGBColor(int rojo, int verde, int azul) {
        rojo = valorRojo;
        verde=valorVerde;
        azul=valorAzul;
    }
}

```

Este método no trabajará como se espera. Cuando el control llega a la sentencia `println()` en el siguiente fragmento de código, los argumentos rojo, verde y azul de `getRGBColor()` ya no existen. Por lo tanto las asignaciones realizadas dentro del método no tendrán efecto; r, g, y b seguirán siendo igual a -1.

```

. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("rojo = " + r + ", verde = " + g + ", azul = " + b);
. . .

```

El paso de las variables por valor le ofrece alguna seguridad a los programadores: los métodos no pueden modificar de forma no intencionada una variable que está fuera de su ámbito. Sin embargo, alguna vez se querrá que un método modifique alguno de sus argumentos. El método `getRGBColor()` es un caso apropiado. El llamador quiere que el método devuelva tres valores a través de sus argumentos. Sin embargo, el método no puede modificar sus argumentos, y, además, un método sólo puede devolver un valor a través de su valor de retorno. Entonces, ¿cómo puede un método devolver más de un valor, o tener algún efecto (modificar algún valor) fuera de su ámbito?

Para que un método modifique un argumento, debe ser un tipo de referencia como un objeto o un array. Los objetos y arrays también son pasados por valor, pero el valor de un objeto es una referencia. Entonces el efecto es que los argumentos de tipos de referencia son pasados por referencia. De aquí el nombre. Una referencia a un objeto es la dirección del objeto en la memoria. Ahora, el argumento en el método se refiere a la misma posición de memoria que el llamador.

Reescribamos el método `getRGBColor()` para que haga lo que queremos. Primero introduzcamos un nuevo objeto `RGBColor`, que puede contener los valores de rojo, verde y azul de un color en formato RGB:

```

class RGBColor {
    public int rojo, verde, azul;;
}

```

Ahora podemos reescribir `getRGBColor()` para que acepte un objeto `RGBColor` como argumento. El método `getRGBColor()` devuelve el color actual de lápiz, en los valores de las variables miembro rojo, verde y azul de su argumento `RGBColor`:

```

class Pen {

```



```
int valorRojo, valorVerde, valorAzul;
void getRGBColor(RGBColor unColor) {
    unColor.rojo = valorRojo;
    unColor.verde = valorVerde;
    unColor.azul = valorAzul;
}
}
```

Y finalmente, reescribimos la secuencia de llamada:

```
. . .
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("ojo = " + penColor.rojo + ", verde = " + penColor.verde + ",
                    azul = " + penColor.azul);
. . .
```

Las modificaciones realizadas al objeto RGBColor dentro del método getRGBColor() afectan al objeto creado en la secuencia de llamada porque los nombres penColor (en la secuencia de llamada) y unColor (en el método getRGBColor()) se refieren al mismo objeto.

El Cuerpo del Método

En el siguiente ejemplo, el cuerpo de método para los métodos `estaVacio()` y `poner()` están en **negrita**:

```
class Stack {
    static final int PILA_VACIA = -1;
    Object[] elementosPila;
    int elementoSuperior = PILA_VACIA;
    . . .
    boolean estaVacio() {
        if (elementoSuperior == PILA_VACIA)
            return true;
        else
            return false;
    }
    Object poner() {
        if (elementoSuperior == PILA_VACIA)
            return null;
        else {
            return elementosPila[elementoSuperior--];
        }
    }
}
```

Junto a los elementos normales del lenguaje Java, se puede utilizar `this` en el cuerpo del método para referirse a los miembros del objeto actual. El objeto actual es el objeto del que uno de cuyos miembros está siendo llamado. También se puede utilizar `super` para referirse a los miembros de la superclase que el objeto actual haya ocultado mediante la sobrescritura. Un cuerpo de método también puede contener declaraciones de variables que son locales de ese método.

`this`

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable miembro y uno de los argumentos del método que tengan el mismo nombre.

Por ejemplo, el siguiente constructor de la clase `HSBColor` inicializa alguna variable miembro de un objeto de acuerdo a los argumentos pasados al constructor. Cada argumento del constructor tiene el mismo nombre que la variable del objeto cuyo valor contiene el argumento.

```
class HSBColor {
    int hue, saturacion, brillo;
```

```

        HSBColor (int luminosidad, int saturacion, int brillo) {
            this.luminosidad = luminosidad;
            this.saturacion = saturacion;
            this.brillo = brillo;
        }
    }

```

Se debe utilizar this en este constructor para evitar la ambigüedad entre el argumento luminosidad y la variable miembro luminosidad (y así con el resto de los argumentos). Escribir luminosidad = luminosidad; no tendría sentido. Los nombres de argumentos tienen mayor precedencia y ocultan a los nombres de las variables miembro con el mismo nombre. Para referirse a la variable miembro se debe hacer explícitamente a través del objeto actual--this.

También se puede utilizar this para llamar a uno de los métodos del objeto actual. Esto sólo es necesario si existe alguna ambigüedad con el nombre del método y se utiliza para intentar hacer el código más claro.

super

Si el método oculta una de las variables miembro de la superclase, se puede referir a la variable oculta utilizando super. De igual forma, si el método sobrescribe uno de los métodos de la superclase, se puede llamar al método sobrescrito a través de super.

Consideremos esta clase:

```

class MiClase {
    boolean unaVariable;
    void unMetodo() {
        unaVariable = true;
    }
}

```

y una subclase que oculta unaVariable y sobrescribe unMetodo():

```

class OtraClase extends MiClase {
    boolean unaVariable;
    void unMetodo() {
        unaVariable = false;
        super.unMetodo();
        System.out.println(unaVariable);
        System.out.println(super.unaVariable);
    }
}

```

Primero unMetodo() selecciona unaVariable (una declarada en OtraClase que oculta a la declarada en MiClase) a false. Luego unMetodo() llama a su método sobrescrito con esta sentencia:

```
super.unMetodo( );
```

Esto selecciona la versión oculta de unaVariable (la declarada en MiClase) a true. Luego unMetodo muestra las dos versiones de unaVariable con diferentes valores:

```
false  
true
```

Variables Locales

Dentro del cuerpo de un método se puede declarar más variables para usarlas dentro del método. Estas variables son variables locales y viven sólo mientras el control permanezca dentro del método. Este método declara un variable local i y la utiliza para operar sobre los elementos del array.

```
Object encontrarObjetoEnArray(Object o, Object[] arrayDeObjetos) {  
    int i;        // variable local  
    for (i = 0; i < arrayDeObjetos.length; i++) {  
        if (arrayDeObjetos[i] == o)  
            return o;  
    }  
    return null;  
}
```

Después de que este método retorne, i ya no existirá más.

Miembros de la Clase y del Ejemplar

Cuando se declara una variable miembro como unFloat en MiClase:

```
class MiClase {  
    float unFloat;  
}
```

declara una variable de ejemplar. Cada vez que se crea un ejemplar de la clase, el sistema crea una copia de todas las variables de ejemplar de la clase. Se puede acceder a una variable del ejemplar del objeto desde un objeto como se describe en [Utilizar Objetos](#).

Las variables de ejemplar están en contraste con las variables de clase (que se declaran utilizando el modificador static). El sistema asigna espacio para las variables de clase una vez por clase, sin importar el número de ejemplares creados de la clase. Todos los objetos creados de esta clase comparten la misma copia de las variables de clase de la clase, se puede acceder a las variables de clase a través de un ejemplar o través de la propia clase.

Los métodos son similares: una clase puede tener métodos de ejemplar y métodos de clase. Los métodos de ejemplar operan sobre las variables de ejemplar del objeto actual pero también pueden acceder a las variables de clase. Por otro lado, los métodos de clase no pueden acceder a las variables del ejemplar declarados dentro de la clase (a menos que se cree un objeto nuevo y acceda a ellos através del objeto). Los métodos de clase también pueden ser invocados desde la clase, no se necesita un ejemplar para llamar a los métodos de la clase.

Por defecto, a menos que se especifique de otra forma, un miembro declarado dentro de una clase es un miembro del ejemplar. La clase definida abajo tiene una variable de ejemplar -- un entero llamado x -- y dos métodos de ejemplar -- x() y setX() -- que permite que otros objetos pregunten por el valor de x:

```
class UnEnteroLlamadoX {  
    int x;  
    public int x() {  
        return x;  
    }  
    public void setX(int newX) {  
        x = newX;  
    }  
}
```

Cada vez que se ejemplariza un objeto nuevo desde una clase, se obtiene una copia de cada una de las variables de ejemplar de la clase. Estas copias están asociadas con el objeto nuevo. Por eso, cada vez que se ejemplariza un nuevo objeto UnEnteroLlamadoX de la clase, se obtiene una copia de x que está asociada con el nuevo objeto UnEnteroLlamadoX.

Todos los ejemplares de una clase comparten la misma implementación de un método de ejemplar; todos los ejemplares de UnEnteroLlamadoX comparten la misma implementación de x() y setX(). Observa que estos métodos se refieren a la variable de ejemplar del objeto x por su nombre. "Pero, ¿si todos los ejemplares de UnEnteroLlamadoX comparten la misma implementación de x() y setX() esto no es ambigüo?" La respuesta es no. Dentro de un método de ejemplar, el nombre de una variable de ejemplar se refiere a la variable de ejemplar del objeto actual (asumiendo que la variable de ejemplar no está ocultada por un parámetro del método). Ya que, dentro de x() y setX(), x es equivalente a this.x.

Los objetos externos a UnEnteroLlamadoX que deseen acceder a x deben hacerlo a través de un ejemplar particular de UnEnteroLlamadoX. Supongamos que este código estuviera en otro método del objeto. Crea dos objetos diferentes del tipo UnEnteroLlamadoX, y selecciona sus

valores de x a diferentes valores Y luego lo muestra:

```
. . .
UnEnteroLlamadoX miX = new UnEnteroLlamadoX();
UnEnteroLlamadoX otroX = new UnEnteroLlamadoX();
miX.setX(1);
otroX.x = 2;
System.out.println("miX.x = " + miX.x());
System.out.println("otroX.x = " + otroX.x());
. . .
```

Observa que el código utilizado en `setX()` para seleccionar el valor de `x` para `miX` pero solo asignando el valor `otroX.x` directamente. De otra forma, el código manipula dos copias diferentes de `x`: una contenida en el objeto `miX` y la otra en el objeto `otroX`. La salida producida por este código es:

```
miX.x = 1
otroX.x = 2
```

mostrando que cada ejemplar de la clase `UnEnteroLlamadoX` tiene su propia copia de la variable de ejemplar `x` y que cada `x` tiene un valor diferente.

Cuando se declara una variable miembro se puede especificar que la variable es una variable de clase en vez de una variable de ejemplar. Similarmente, se puede especificar que un método es un método de clase en vez de un método de ejemplar. El sistema crea una sola copia de una variable de clase la primera vez que encuentra la clase en la que está definida la variable. Todos los ejemplares de esta clase comparten la misma copia de las variables de clase. Los métodos de clase solo pueden operar con variables de clase -- no pueden acceder a variables de ejemplar definidas en la clase.

Para especificar que una variable miembro es una variable de clase, se utiliza la palabra clave `static`. Por ejemplo, cambiemos la clase `UnEnteroLlamadoX` para que su variable `x` sea ahora una variable de clase:

```
class UnEnteroLlamadoX {
    static int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Ahora veamos el mismo código mostrado [anteriormente](#) que crea dos ejemplares de `UnEnteroLlamadoX`, selecciona sus valores de `x`, y muestra esta salida diferente:

```
miX.x = 2
otroX.x = 2
```

La salida es diferente porque `x` ahora es una variable de clase por lo que sólo hay una copia de la variable y es compartida por todos los ejemplares de `UnEnteroLlamadoX` incluyendo `miX` y `otroX`.

Cuando se llama a `setX()` en cualquier ejemplar, cambia el valor de `x` para todos los ejemplares de `UnEnteroLlamadoX`.

Las variables de clase se utilizan para aquellos puntos en lo que se necesite una sola copia que debe estar accesible para todos los objetos heredados por la clase en la que la variable fue declarada. Por ejemplo, las variables de clase se utilizan frecuentemente con final para definir

constantes (esto es más eficiente en el consumo de memoria, ya que las constantes no pueden cambiar y sólo se necesita una copia).

Similarmente, cuando se declare un método, se puede especificar que el método es un método de clase en vez de un método de ejemplar. Los métodos de clase sólo pueden operar con variables de clase y no pueden acceder a las variables de ejemplar definidas en la clase.

Para especificar que un método es un método de clase, se utiliza la palabra clave `static` en la declaración de método. Cambiemos la clase `UnEnteroLlamadoX` para que su variable miembro `x` sea de nuevo una variable de ejemplar, y sus dos métodos sean ahora métodos de clase:

```
class UnEnteroLlamadoX {
    private int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Cuando se intente compilar esta versión de `UnEnteroLlamadoX`, se obtendrán errores de compilación:

```
UnEnteroLlamadoX.java:4: Can't make a static reference to nonstatic variable x in
class UnEnteroLlamadoX.
    return x;
           ^
```

```
UnEnteroLlamadoX.java:7: Can't make a static reference to nonstatic variable x in
class UnEnteroLlamadoX.
    x = newX;
    ^
```

2 errors

Esto es porque los métodos de la clase no pueden acceder a variables de ejemplar a menos que el método haya creado un ejemplar de `UnEnteroLlamadoX` primero y luego acceda a la variable a través de él.

Construyamos de nuevo `UnEnteroLlamadoX` para hacer que su variable `x` sea una variable de clase:

```
class UnEnteroLlamadoX {
    static private int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Ahora la clase se compilará y el código [anterior](#) que crea dos ejemplares de `UnEnteroLlamadoX`, selecciona sus valores `x`, y muestra en su salida los valores de `x`:

```
miX.x = 2
otroX.x = 2
```

De nuevo, cambiar `x` a través de `miX` también lo cambia para los otros ejemplares de `UnEnteroLlamadoX`.

Otra diferencia entre miembros del ejemplar y de la clase es que los miembros de la clase son accesibles desde la propia clase. No se necesita ejemplarizar la clase para acceder a los miembros de clase. Reescribamos el código [anterior](#) para acceder a `x()` y `setX()` directamente desde la clase `UnEnteroLlamadoX`:

```
. . .
UnEnteroLlamadoX.setX(1);
System.out.println("UnEnteroLlamadoX.x = " + UnEnteroLlamadoX.x());
. . .
```

Observa que ya no se tendrá que crear `miX` u `otroX`. Se puede seleccionar `x` y recuperarlo directamente desde la clase `UnEnteroLlamadoX`. No se puede hacer esto con miembros del ejemplar. Solo se puede invocar métodos de ejemplar a través de un objeto y sólo puede acceder a las variables de ejemplar desde un objeto. Se puede acceder a las variables y métodos de clase desde un ejemplar de la clase o desde la clase misma.

Controlar el Acceso a los Miembros de la Clase

Uno de los beneficios de las clases es que pueden proteger sus variables y métodos miembros frente al acceso de otros objetos. ¿Por qué es esto importante? Bien, consideremos esto. Se ha escrito una clase que representa una petición a una base de datos que contiene toda clase de información secreta, es decir, registros de empleados o proyectos secretos de la compañía.

Ciertas informaciones y peticiones contenidas en la clase, las soportadas por los métodos y variables accesibles públicamente en su objeto son correctas para el consumo de cualquier otro objeto del sistema. Otras peticiones contenidas en la clase son sólo para el uso personal de la clase. Estas otras soportadas por la operación de la clase no deberían ser utilizadas por objetos de otros tipos. Se querría proteger esas variables y métodos personales a nivel del lenguaje y prohibir el acceso desde objetos de otros tipos.

En Java se puede utilizar los especificadores de acceso para proteger tanto las variables como los métodos de la clase cuando se declaran. El lenguaje Java soporta cuatro niveles de acceso para las variables y métodos miembros: `private`, `protected`, `public`, y, todavía no especificado, acceso de paquete.

La siguiente tabla le muestra los niveles de acceso permitidos por cada especificador:

Especificador	clase	subclase	paquete	mundo
<code>private</code>	X			
<code>protected</code>	X	X*	X	
<code>public</code>	X	X	X	X
<code>package</code>	X		X	

La primera columna indica si la propia clase tiene acceso al miembro definido por el especificador de acceso. La segunda columna indica si las subclases de la clase (sin importar dentro de que paquete se encuentren estas) tienen acceso a los miembros. La tercera columna indica si las clases del mismo paquete que la clase (sin importar su parentesco) tienen acceso a los miembros. La cuarta columna indica si todas las clases tienen acceso a los miembros.

Observa que la intersección entre `protected` y subclase tiene un '*' - este caso de acceso particular tiene una explicación en más detalle [más adelante](#).

Echemos un vistazo a cada uno de los niveles de acceso más detalladamente:

Private

El nivel de acceso más restringido es `private`. Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden poner en peligro el estado del objeto o del programa donde se está ejecutando. Los miembros privados son como secretos, nunca deben contarsele a nadie.

Para declarar un miembro privado se utiliza la palabra clave `private` en su declaración. La clase siguiente contiene una variable miembro y un método privados:

```
class Alpha {
    private int soyPrivado;
    private void metodoPrivado() {
        System.out.println("metodoPrivado");
    }
}
```

Los objetos del tipo Alpha pueden inspeccionar y modificar la variable soyPrivado y pueden invocar el método metodoPrivado(), pero los objetos de otros tipos no pueden acceder. Por ejemplo, la clase Beta definida aquí:

```
class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPrivado = 10;        // ilegal
        a.metodoPrivado();        // ilegal
    }
}
```

no puede acceder a la variable soyPrivado ni al método metodoPrivado() de un objeto del tipo Alpha porque Beta no es del tipo Alpha.

Si una clase está intentando acceder a una variable miembro a la que no tiene acceso--el compilador mostrará un mensaje de error similar a este y no compilará su programa:

```
Beta.java:9: Variable iamprivate in class Alpha not accessible from class Beta.
    a.iamprivate = 10;        // ilegal
    ^
1 error
```

Y si un programa intenta acceder a un método al que no tiene acceso, generará un error de compilación parecido a este:

```
Beta.java:12: No method matching privateMethod() found in class Alpha.
    a.privateMethod();        // ilegal
1 error
```

Protected

El siguiente especificador de nivel de acceso es 'protected' que permite a la propia clase, las subclases (con la excepción a la que nos referimos anteriormente), y todas las clases dentro del mismo paquete que accedan a los miembros. Este nivel de acceso se utiliza cuando es apropiado para una subclase da la clase tener acceso a los miembros, pero no las clases no relacionadas. Los miembros protegidos son como secretos familiares - no importa que toda la familia lo sepa, incluso algunos amigos allegados pero no se quiere que los extraños lo sepan.

Para declarar un miembro protegido, se utiliza la palabra clave protected. Primero echemos un vistazo a cómo afecta este especificador de acceso a las clases del mismo paquete.

Consideremos esta versión de la clase Alpha que ahora se declara para estar incluida en el paquete Griego y que tiene una variable y un método que son miembros protegidos:

```
package Griego;

class Alpha {
    protected int estoyProtegido;
    protected void metodoProtegido() {
        System.out.println("metodoProtegido");
    }
}
```

Ahora, supongamos que la clase Gamma, también está declarada como miembro del paquete Griego (y no es una subclase de Alpha). La Clase Gamma puede acceder legalmente al miembro estoyProtegido del objeto Alpha y puede llamar legalmente a su método metodoProtegido():

```
package Griego;

class Gamma {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyProtegido = 10;    // legal
        a.metodoProtegido();      // legal
    }
}
```

Esto es muy sencillo. Ahora, investiguemos cómo afecta el especificador portected a una subclase de Alpha.

Introduzcamos una nueva clase, Delta, que descende de la clase Alpha pero reside en un paquete diferente - Latin. La clase Delta puede acceder tanto a estoyProtegido como a metodoProtegido(), pero solo en objetos del tipo Delta o sus subclases. La clase Delta no puede acceder a estoyProtegido o metodoProtegido() en objetos del tipo Alpha. metodoAccesor() en el siguiente ejemplo intenta acceder a la variable miembro estoyProtegido de un objeto del tipo Alpha, que es ilegal, y en un objeto del tipo Delta que es legal. Similarmente, metodoAccesor() intenta invocar a metodoProtegido() en un objeto del tipo Alpha, que también es ilegal:

```
import Griego.*;

package Latin;

class Delta extends Alpha {
    void metodoAccesor(Alpha a, Delta d) {
        a.estoyProtegido = 10;    // ilegal
        d.estoyProtegido = 10;    // legal
        a.metodoProtegido();      // ilegal
        d.metodoProtegido();      // legal
    }
}
```

Si una clase es una subclase o se cuenta en el mismo paquete de la clase con el miembro protegido, la clase tiene acceso al miembro protegido.

Public

El especificador de acceso más sencillo es 'public'. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran ólo si su acceso no produce resultados indeseados si un extraño los utiliza. Aquí no hay secretos familiares; no importa que lo sepa todo el mundo.

Para declarar un miembro público se utiliza la palabra clave public. Por ejemplo,

```
package Griego;

class Alpha {
    public int soyPublico;
    public void metodoPublico() {
        System.out.println("metodoPublico");
    }
}
```

Reescribamos nuestra clase Beta una vez más y la ponemos en un paquete diferente que la clase Alpha y nos aseguramos que no están relacionadas (no es una subclase) de

Alpha:

```
import Griego.*;

package Romano;

class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPublico = 10;           // legal
        a.metodoPublico();           // legal
    }
}
```

Como se puede ver en el ejemplo anterior, Beta puede inspeccionar y modificar legalmente la variable `soyPublico` en la clase Alpha y puede llamar legalmente al método `metodoPublico()`.

Acceso de Paquete

Y finalmente, el último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros. Este nivel de acceso asume que las clases del mismo paquete son amigas de confianza. Este nivel de confianza es como la que extiende a sus mejores amigos y que incluso no la tiene con su familia.

Por ejemplo, esta versión de la clase Alpha declara una variable y un método con acceso de paquete. Alpha reside en el paquete Griego:

```
package Griego;

class Alpha {
    int estoyEmpaquetado;
    void metodoEmpaquetado() {
        System.out.println("metodoEmpaquetado");
    }
}
```

La clase Alpha tiene acceso a `estoyEmpaquetado` y a `metodoEmpaquetado()`. Además, todas las clases declaradas dentro del mismo paquete como Alpha también tienen acceso a `estoyEmpaquetado` y `metodoEmpaquetado()`. Supongamos que tanto Alpha como Beta son declaradas como parte del paquete Griego:

```
package Griego;

class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyEmpaquetado = 10;           // legal
        a.metodoEmpaquetado();             // legal
    }
}
```

Entonces Beta puede acceder legalmente a `estoyEmpaquetado` y `metodoEmpaquetado()`.

Constructores

Todas las clases Java tienen métodos especiales llamados Constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Los constructores tienen el mismo nombre que la clase --el nombre del constructor de la clase Rectangle es Rectangle(), el nombre del constructor de la clase Thread es Thread(), etc...

Java soporta la sobrecarga de los nombres de métodos para que una clase puede tener cualquier número de constructores, todos los cuales tienen el mismo nombre. Al igual que otros métodos sobrecargados, los constructores se diferencian unos de otros en el número y tipo de sus argumentos.

Consideremos la clase Rectangle del paquete java.awt que proporciona varios constructores diferentes, todos llamados Rectangle(), pero cada uno con número o tipo diferentes de argumentos a partir de los cuales se puede crear un nuevo objeto Rectangle. Aquí tiene las firmas de los constructores de la clase java.awt.Rectangle:

```
public Rectangle()  
public Rectangle(int width, int height)  
public Rectangle(int x, int y, int width, int height)  
public Rectangle(Dimension size)  
public Rectangle(Point location)  
public Rectangle(Point location, Dimension size)
```

El primer constructor de Rectangle inicializa un nuevo Rectangle con algunos valores por defecto razonables, el segundo constructor inicializa el nuevo Rectangle con la altura y anchura especificadas, el tercer constructor inicializa el nuevo Rectangle en la posición especificada y con la altura y anchura especificadas, etc...

Típicamente, un constructor utiliza sus argumentos para inicializar el estado del nuevo objeto. Entonces, cuando se crea un objeto, se debe elegir el constructor cuyos argumentos reflejen mejor cómo se quiere inicializar el objeto.

Basándose en el número y tipos de los argumentos que se pasan al constructor, el compilador determina cual de ellos utilizar, Así el compilador sabe que cuando se escribe:

```
new Rectangle(0, 0, 100, 200);
```

el compilador utilizará el constructor que requiere cuatro argumentos enteros, y cuando se escribe:

```
new Rectangle(miObjetoPoint, miObjetoDimension);
```

utilizará el constructor que requiere como argumentos un objeto Point y un objeto Dimension.

Cuando escribas tus propias clases, no tienes porque proporcionar constructores. El constructor por defecto, el constructor que no necesita argumentos, lo proporciona automáticamente el sistema para todas las clases. Sin embargo, frecuentemente

se querrá o necesitará proporcionar constructores para las clases.

Se puede declarar e implementar un constructor como se haría con cualquier otro método en una clase. El nombre del constructor debe ser el mismo que el nombre de la clase y, si se proporciona más de un constructor, los argumentos de cada uno de los constructores deben diferenciarse en el número o tipo. No se tiene que especificar el valor de retorno del constructor.

El constructor para esta subclase de Thread, un hilo que realiza animación, selecciona algunos valores por defecto como la velocidad de cuadro, el número de imágenes y carga las propias imágenes:

```
class AnimationThread extends Thread {
    int framesPerSecond;
    int numImages;
    Image[] images;

    AnimationThread(int fps, int num) {
        int i;

        super("AnimationThread");
        this.framesPerSecond = fps;
        this.numImages = num;

        this.images = new Image[numImages];
        for (i = 0; i <= numImages; i++) {
            . . .
            // Carga las imágenes
            . . .
        }
    }
}
```

Observa cómo el cuerpo de un constructor es igual que el cuerpo de cualquier otro método -- contiene declaraciones de variables locales, bucles, y otras sentencias. Sin embargo, hay una línea en el constructor de AnimationThread que no se verá en un método normal--la segunda línea:

```
super("AnimationThread");
```

Esta línea invoca al constructor proporcionado por la superclase de AnimationThread--Thread. Este constructor particular de Thread acepta una cadena que contiene el nombre del Thread. Frecuentemente un constructor se aprovechará del código de inicialización escrito para la superclase de la clase.

En realidad, algunas clases deben llamar al constructor de su superclase para que el objeto trabaje de forma apropiada. Típicamente, llamar al constructor de la superclase es lo primero que se hace en el constructor de la subclase: un objeto debe realizar primero la inicialización de nivel superior.

Cuando se declaren constructores para las clases, se pueden utilizar los especificadores de acceso normales para especificar si otros objetos pueden crear ejemplares de sus clase:

private

Ninguna otra clase puede crear un objeto de su clase. La clase puede contener métodos públicos y esos métodos pueden construir un objeto y devolverlo, pero nada más.

protected

Sólo las subclases de la clase pueden crear ejemplares de ella.

public

Cualquiera pueda crear un ejemplar de la clase.

package-access

Nadie externo al paquete puede construir un ejemplar de su clase. Esto es muy útil si se quiere que las clase que tenemos en un paquete puedan crear ejemplares de la clase pero no se quiere que lo haga nadie más.

Escribir un Método finalize()

Antes de que un objeto sea recolectado por el recolector de basura, el sistema llama al método finalize(). La intención de este método es liberar los recursos del sistema, como ficheros o conexiones abiertas antes de empezar la recolección.

Una clase puede proporcionar esta finalización simplemente definiendo e implementando un método llamado finalize(). El método finalize() debe declararse de la siguiente forma:

```
protected void finalize () throws throwable
```

Esta clase abre un fichero cuando se construye:

```
class AbrirUnFichero {
    FileInputStream unFichero = null;
    AbrirUnFichero(String nombreFichero) {
        try {
            unFichero = new FileInputStream(nombreFichero);
        } catch (java.io.FileNotFoundException e) {
            System.err.println("No se pudo abrir el fichero " + nombreFichero);
        }
    }
}
```

Para un buen comportamiento, la clase AbrirUnFichero debería cerrar el fichero cuando haya finalizado. Aquí tienes el método finalize() para la clase AbrirUnFichero:

```
protected void finalize () throws throwable {
    if (unFichero != null) {
        unFichero.close();
        unFichero = null;
    }
}
```

El método finalize() está declarado en la clase java.lang.Object. Así cuando escribas un método finalize() para tus clases estás sobrescribiendo el de su superclase. En [Sobreescribir Métodos](#) encontrarás más información sobre la sobreescritura de métodos.

Si la superclase tiene un método finalize(), probablemente este método deberá llamar al método finalize() de su superclase después de haber terminado sus tareas de limpieza. Esto limpiará cualquier recurso obtenido sin saberlo a través de los métodos heredados desde la superclase.

```
protected void finalize() throws Throwable {
    . . .
    // aquí va el código de limpieza de esta clase
    . . .
    super.finalize();
}
```


Subclases, Superclases, y Herencia

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama subclase. La clase de la que está derivada se denomina superclase.

De hecho, en Java, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿Dónde empieza todo esto?. La clase más alta, la clase de la que todas las demás descienden, es la clase Object, definida en java.lang. Object es la raíz de la herencia de todas las clases.

Las subclases heredan el estado y el comportamiento en forma de las variables y los métodos de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede modificarlos o sobrescribirlos. Por eso, según se va bajando por el árbol de la herencia, las clases se convierten en más y más especializadas:

Definición: Una subclase es una clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de todos sus ancestros. El término superclase se refiere a la clase que es el ancestro más directo, así como a todas las clases ascendentes.

Crear Subclases

Se declara que una clase es una subclase de otra clase dentro de [La declaración de Clase](#). Por ejemplo, supongamos que queremos crear una subclase llamada SubClase de otra clase llamada SuperClase. Se escribiría esto:

```
class SubClass extends SuperClass {  
    . . .  
}
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamente que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo del árbol de la herencia. Para hacer esta explicación un poco más sencilla, cuando este tutorial se refiere a la superclase de una clase significa el ancestro más directo de la clase así como a todas sus clases ascendentes.

Una clase Java sólo puede tener una superclase directa. Java no soporta la herencia múltiple.

Crear una subclase puede ser tan sencillo como incluir la cláusula `extends` en la declaración de la clase. Sin embargo, normalmente se deberá realizar alguna cosa más cuando se crea una subclase, como sobrescribir métodos, etc...

¿Qué variables miembro hereda una subclase?

Regla: Una subclase hereda todas las variables miembros de su superclase que puedan ser accesibles desde la subclase (a menos que la variable miembro esté oculta en la subclase).

Esto es, las subclases:

- heredan aquellas variables miembros declaradas como `public` o `protected`
- heredan aquellas variables miembros declaradas sin especificador de acceso (normalmente conocidas como "Amigas") siempre que la subclase esté en el mismo paquete que la clase
- no hereda las variables miembros de la superclase si la subclase declara una variable miembro que utiliza el mismo nombre. La variable miembro de la subclase se dice que oculta a la variable miembro de la superclase.
- no hereda las variables miembro `private`

Ocultar Variables Miembro

Como se mencionó en la sección anterior, las variables miembros definidas en la subclase ocultan las variables miembro que tienen el mismo nombre en la superclase.

Como esta característica del lenguaje Java es poderosa y conveniente, puede ser una fuente de errores: ocultar una variable miembro puede hacerse deliberadamente o por accidente. Entonces, cuando nombres tus variables miembro se cuidadoso y oculta sólo las variables miembro que realmente deseas ocultar.

Una característica interesante de las variables miembro en Java es que una clase puede acceder a una variable miembro oculta a través de su superclase. Considere este par de superclase y subclase:

```
class Super {  
    Number unNumero;  
}  
class Sub extends Super {  
    Float unNumero;  
}
```

La variable unNumero de Sub oculta a la variable unNumero de Super. Pero se puede acceder a la variable de la superclase utilizando:

```
super.unNumero
```

super es una palabra clave del lenguaje Java que permite a un método referirse a las variables ocultas y métodos sobrescritos de una superclase.

¿Qué métodos hereda una Subclase?

La regla que especifica los métodos heredados por una subclase es similar a la de las variables miembro.

Regla: Una subclase hereda todos los métodos de sus superclase que son accesibles para la subclase (a menos que el método sea sobrescrito por la subclase).

Esto es, una Subclase:

- hereda aquellos métodos declarados como public o protected
- hereda aquellos métodos sin especificador de acceso, siempre que la subclase esté en el mismo paquete que la clase
- no hereda un método de la superclase si la subclase declara un método que utiliza el mismo nombre. Se dice que el método de la subclase sobrescribe al método de la superclase.
- no hereda los métodos private

Sobreescribir Métodos

La habilidad de una subclase para sobreescribir un método de su superclase permite a una clase heredar de su superclase aquellos comportamientos "más cercanos" y luego suplementar o modificar el comportamiento de la superclase.

Sobreescribir Métodos

Una subclase puede sobreescribir completamente la implementación de un método heredado o puede mejorar el método añadiéndole funcionalidad.

Reemplazar la Implementación de un Método de una Superclase

Algunas veces, una subclase querría reemplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con la esperanza de que la mayoría, si no todas, sus subclases reemplacen completamente la implementación de ese método.

Un ejemplo de esto es el método `run()` de la clase `Thread`. La clase `Thread` proporciona una implementación vacía (el método no hace nada) para el método `run()`, porque por definición, este método depende de la subclase. La clase `Thread` posiblemente no puede proporcionar una implementación medianamente razonable del método `run()`.

Para reemplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre que el del método de la superclase y se sobreescribe el método con la misma firma que la del método sobreescrito:

```
class ThreadSegundoPlano extends Thread {  
    void run() {  
        . . .  
    }  
}
```

La clase `ThreadSegundoPlano` sobreescribe completamente el método `run()` de su superclase y reemplaza completamente su implementación.

Añadir Implementación a un Método de la Superclase

Otras veces una subclase querrá mantener la implementación del método de su superclase y posteriormente ampliar algún comportamiento específico de la subclase. Por ejemplo, los métodos constructores de una subclase lo hacen normalmente--la subclase quiere preservar la inicialización realizada por la superclase, pero proporciona inicialización adicional específica de la subclase.

Supongamos que queremos crear una subclase de la clase `Window` del paquete `java.awt`. La clase `Window` tiene un constructor que requiere un argumento del tipo `Frame` que es el padre de la ventana:

```
public Window(Frame parent)
```

Este constructor realiza alguna inicialización en la ventana para que trabaje dentro del sistema de ventanas. Para asegurarnos de que una subclase de `Window` también trabaja dentro del sistema de ventanas, deberemos

proporcionar un constructor que realice la misma inicialización. Mucho mejor que intentar recrear el proceso de inicialización que ocurre dentro del constructor de Windows, podríamos utilizar lo que la clase Windows ya hace. Se puede utilizar el código del constructor de Windows llamándolo desde dentro del constructor de la subclase Window:

```
class Ventana extends Window {
    public Ventana(Frame parent) {
        super(parent);
        . . .
        // Ventana especifica su inicialización aquí
        . . .
    }
}
```

El constructor de Ventana llama primero al constructor de su superclase, y no hace nada más. Típicamente, este es el comportamiento deseado de los constructores--las superclases deben tener la oportunidad de realizar sus tareas de inicialización antes que las de su subclase. Otros tipos de métodos podrían llamar al constructor de la supeclase al final del método o en el medio.

Métodos que una Subclase no Puede Sobreescibir

- Una subclase no puede sobreescibir métodos que hayan sido declarados como final en la superclase (por definición, los métodos finales no pueden ser sobreescritos). Si intentamos sobreescibir un método final, el compilador mostrará un mensaje similar a este y no compilará el programa:

```
FinalTest.java:7: Final methods can't be overridden. Method void iamfinal()
is final in class ClassWithFinalMethod.
    void iamfinal() {
        ^
1 error
```

Para una explicación sobre los métodos finales, puedes ver: [Escribir Métodos y Clases Finales](#).

- Una subclase tampoco puede sobreescibir métodos que se hayan declarado como static en la superclase. En otras palabras, una subclase no puede sobreescibir un método de clase. Puedes ver [Miembros del Ejemplar y de la Clase](#) para obtener una explicación sobre los métodos de clase.

Métodos que una Subclase debe Sobreescibir

Las subclases deben sobreescibir aquellos métodos que hayan sido declarados como abstract en la superclase, o la propia subclase debe ser abstracta. [Escribir Clases y Métodos Abstractos](#) explica con más detalle los métodos y clases abstractos.

Escribir Clases y Métodos Finales

Clases Finales

Se puede declarar que una clase sea final; esto es, que la clase no pueda tener subclases. Existen (al menos) dos razones por las que se querría hacer esto: razones de seguridad y de diseño.

Seguridad: Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de subversión, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase.

La clase `String` del paquete `java.lang` es una clase final sólo por esta razón. La clase `String` es tan vital para la operación del compilador y del intérprete que el sistema Java debe garantizar que siempre que un método o un objeto utilicen un `String`, obtenga un objeto `java.lang.String` y no algún otro string. Esto asegura que ningún string tendrán propiedades extrañas, inconsistentes o indeseables.

Si se intenta compilar una subclase de una clase final, el compilador mostrará un mensaje de error y no compilará el programa. Además, los bytescodes verifican que no esta teniendo lugar una subversión, al nivel de byte comprobando que una clase no es una subclase de una clase final.

Diseño: Otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

Para especificar que una clase es una clase final, se utiliza la palabra clave `final` antes de la palabra clave `class` en la declaración de la clase. Por ejemplo, si quisieramos declarar `AlgoritmodeAjedrez` como una clase final (perfecta), la declaración se parecería a esto:

```
final class AlgoritmodeAjedrez {  
    . . .  
}
```

Cualquier intento posterior de crear una subclase de `AlgoritmodeAjedrez` resultará en el siguiente error del compilador:

```
Chess.java:6: Can't subclass final classes: class AlgoritmodeAjedrez  
class MejorAlgoritmodeAjedrez extends AlgoritmodeAjedrez {  
    ^  
1 error
```


Métodos Finales

Si la creación de clases finales parece algo dura para nuestras necesidades, y realmente lo que se quiere es proteger son algunos métodos de una clase para que no sean sobrescritos, se puede utilizar la palabra clave final en la declaración de método para indicar al compilador que este método no puede ser sobrescrito por las subclases.

Se podría desear hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto. Por ejemplo, en lugar de hacer AlgoritmodeAjedrez como una clase final, podríamos hacer siguienteMovimiento() como un método final:

```
class AlgoritmodeAjedrez {  
    . . .  
    final void siguienteMovimiento(Pieza piezaMovida,  
                                   PosicionenTablero nuevaPosicion) {  
    }  
    . . .  
}
```

Escribir Clases y Métodos Abstractos

Clases Abstractas

Algunas veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser ejemplarizado. Por ejemplo, la comida en la vida real. ¿Has visto algún ejemplar de comida? No. Lo que has visto son ejemplares de manzanas, pan, y chocolate. Comida representa un concepto abstracto de cosas que son comestibles. No tiene sentido que exista un ejemplar de comida.

Similarmente en la programación orientada a objetos, se podría modelar conceptos abstractos pero no querer que se creen ejemplares de ellos. Por ejemplo, la clase `Number` del paquete `java.lang` representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase `Number` sólo tiene sentido como superclase de otras clases como `Integer` y `Float` que implementan números de tipos específicos. Las clases como `Number`, que implementan conceptos abstractos y no deben ser ejemplarizadas, son llamadas clases abstractas. Una clase abstracta es una clase que sólo puede tener subclases--no puede ser ejemplarizada.

Para declarar que una clase es una clase abstracta, se utiliza la palabra clave `abstract` en la declaración de la clase.

```
abstract class Number {  
    . . .  
}
```

Si se intenta ejemplarizar una clase abstracta, el compilador mostrará un error similar a este y no compilará el programa:

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be  
instantiated.  
    new AbstractTest();  
    ^  
1 error
```

Métodos Abstractos

Una clase abstracta puede contener métodos abstractos, esto es, métodos que no tienen implementación. De esta forma, una clase abstracta puede definir un interface de programación completo, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar el interface de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

Veamos un ejemplo de cuando sería necesario crear una clase abstracta con métodos abstractos. En una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc.. Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Podemos aprovecharnos de esas similitudes y declararlos todos a partir de un mismo objeto padre-`ObjetoGrafico`.

Sin embargo, los objetos gráficos también tienen diferencias substanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos. Por otro lado, todos los ObjetosGraficos deben saber como dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, ObjetoGrafico, para proporcionar las variables miembro y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método moverA().

También se deberían declarar métodos abstractos como dibujar(), que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase). La clase ObjetoGrafico se parecería a esto:

```
abstract class ObjetoGrafico {
    int x, y;
    . . .
    void moverA(int nuevaX, int nuevaY) {
        . . .
    }
    abstract void dibujar();
}
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectangulo deberán proporcionar una implementación para el método dibujar().

```
class Circulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
class Rectangulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

Una clase abstracta no necesita contener un método abstracto. Pero todas las clases que contengan un método abstracto o no proporcionen implementación para cualquier método abstracto declarado en sus superclases debe ser declarada como una clase abstracta.

La Clase Object

La clase Object está situada en la parte más alta del árbol de la herencia en el entorno de desarrollo de Java. Todas las clases del sistema Java son descendentes (directos o indirectos) de la clase Object. Esta clase define los estados y comportamientos básicos que todos los objetos deben tener, como la posibilidad de compararse unos con otros, de convertirse a cadenas, de esperar una condición variable, de notificar a otros objetos que la condición variable a cambiado y devolver la clase del objeto.

El método equals()

equals() se utiliza para comparar si dos objetos son iguales. Este método devuelve true si los objetos son iguales, o false si no lo son. Observe que la igualdad no significa que los objetos sean el mismo objeto. Consideremos este código que compara dos enteros:

```
Integer uno = new Integer(1), otroUno = new Integer(1);

if (uno.equals(otroUno))
    System.out.println("Los objetos son Iguales");
```

Este código mostrará Los objetos son Iguales aunque uno y otroUno referencian a dos objetos distintos. Se les considera iguales porque su contenido es el mismo valor entero.

Las clases deberíacutear sobreescribir este método proporcionando la comprobación de igualdad apropiada. Un método equals() debería comparar el contenido de los objetos para ver si son funcionalmente iguales y devolver true si es así.

El método getClass()

El método getClass() es un método final (no puede sobreescribirse) que devuelve una representación en tiempo de ejecución de la clase del objeto. Este método devuelve un objeto Class al que se le puede pedir varia información sobre la clase, como su nombre, el nombre de su superclase y los nombres de los interfaces que implementa. El siguiente método obtiene y muestra el nombre de la clase de un objeto:

```
void PrintClassName(Object obj) {
    System.out.println("La clase del Objeto es " +
        obj.getClass().getName());
}
```

Un uso muy manejado del método getClass() es crear un ejemplar de una clase sin conocer la clase en el momento de la compilación. Este método de ejemplo, crea un nuevo ejemplar de la misma clase que obj

que puede ser cualquier clase heredada desde Object (lo que significa que podría ser cualquier clase):

```
Object createNewInstanceOf(Object obj) {  
    return obj.getClass().newInstance();  
}
```

El método toString()

Este método devuelve una cadena de texto que representa al objeto. Se puede utilizar toString para mostrar un objeto. Por ejemplo, se podría mostrar una representación del Thread actual de la siguiente forma:

```
System.out.println(Thread.currentThread().toString());  
System.out.println(new Integer(44).toString());
```

La representación de un objeto depende enteramente del objeto. El String de un objeto entero es el valor del entero mostrado como texto. El String de un objeto Thread contiene varios atributos sobre el thread, como su nombre y prioridad. Por ejemplo, las dos líneas anteriores darían la siguiente salida:

```
Thread[main,5,main]  
4
```

El método toString() es muy útil para depuración y también puede sobrescribir este método en todas las clases.

Otros métodos de Object cubiertos en otras lecciones o secciones

La clase Object proporciona un método, finalize() que limpia un objeto antes de recolectar la basura. Este método se explica en la lección [Eliminar Objetos no Utilizados](#). También en: [Escribir un Método finalize\(\)](#) puedes ver cómo sobrescribir este método para manejar las necesidades de finalización de las clases

La clase Object también proporciona otros cinco métodos:

- notify()
- notifyAll()
- wait() (tres versiones)

que son críticos cuando se escriben programas Java con múltiples thread. Estos métodos ayudan a asegurarse que los thread están sincronizados y se cubren en [Threads de Control](#).

¿Qué es un Interface?

Definición: Un interface es una colección de definiciones de métodos (sin implementaciones) y de valores constantes.

Los interfaces se utilizan para definir un protocolo de comportamiento que puede ser implementado por cualquier clase del árbol de clases.

Los interfaces son útiles para:

- capturar similitudes entre clases no relacionadas sin forzar una relación entre ellas.
- declarar métodos que una o varias clases necesitan implementar.
- revelar el interface de programación de un objeto sin revelar sus clases (los objetos de este tipo son llamados objetos anónimos y pueden ser útiles cuando compartas un paquete de clases con otros desarrolladores).

En Java, un interface es un tipo de dato de referencia, y por tanto, puede utilizarse en muchos de los sitios donde se pueda utilizar cualquier tipo (como en un argumento de métodos y una declaración de variables). Podrás ver todo esto en:

[Utilizar un Interface como un Tipo.](#)

Los Interfaces No Proporcionan Herencia Múltiple

Algunas veces se trata a los interfaces como una alternativa a la herencia múltiple en las clases. A pesar de que los interfaces podrían resolver algunos problemas de la herencia múltiple, son animales bastante diferentes. En particular:

- No se pueden heredar variables desde un interface.
- No se pueden heredar implementaciones de métodos desde un interface.
- La herencia de un interface es independiente de la herencia de la clase--las clases que implementan el mismo interface pueden o no estar relacionadas a través del árbol de clases.

Definir un Interface

Para crear un Interface, se debe escribir tanto la declaración como el cuerpo del interface:

```
declaraciondeInterface {  
    cuerposeInterface  
}
```

La Declaración de Interface declara varios atributos del interface, como su nombre o si se extiende desde otro interface. El Cuerpo de Interface contiene las constantes y las declaraciones de métodos del Interface.

La Declaración de Interface

Como mínimo, una declaración de interface contiene la palabra clave interface y el nombre del interface que se va a crear:

```
interface Contable {  
    . . .  
}
```

Nota: Por convención, los nombres de interfaces empiezan con una letra mayúscula al igual que las clases. Frecuentemente los nombres de interfaces terminan en "able" o "ible".

Una declaración de interface puede tener otros dos componentes: el especificador de acceso public y una lista de "superinterfaces". Un interface puede extender otros interfaces como una clase puede extender o subclassificar otra clase. Sin embargo, mientras que una clase sólo puede extender una superclase, los interfaces pueden extender de cualquier número de interfaces. Así, una declaración completa de interface se parecería a esto:

```
[public] interface Nombredentinterface [extends listadeSuperInterfaces] {  
    . . .  
}
```

El especificador de acceso public indica que el interface puede ser utilizado por todas las clases en cualquier paquete. Si el interface no se especifica como público, sólo será accesible para las clases definidas en el mismo paquete que el interface.

La cláusula extends es similar a la utilizada en la declaración de una clase, sin embargo, un interface puede extender varios interfaces (mientras una clase sólo puede extender una), y un interface no puede extender clases. Esta lista de superinterfaces es una lista delimitada por comas de todos los interfaces extendidos por el nuevo interface.

Un interface hereda todas las constantes y métodos de sus superinterfaces a menos que el interface oculte una constante con el mismo nombre o redeclare un método con una nueva declaración.

El cuerpo del Interface

El cuerpo del interface contiene las declaraciones de métodos para los métodos definidos en el interface. [Implementar Métodos](#) muestra cómo escribir una declaración de método. Además de las declaraciones del métodos, un interface puede contener declaraciones de constantes. En [Declarar Variables Miembros](#) existe más información sobre cómo construir una declaración de una variable miembro.

Nota: Las declaraciones de miembros en un interface no permiten el uso de algunos modificadores y desaconsejan el uso de otros. No se podrán utilizar `transient`, `volatile`, o `synchronized` en una declaración de miembro en un interface. Tampoco se podrá utilizar los especificadores `private` y `protected` cuando se declaren miembros de un interface.

Todos los valores constantes definidos en un interfaces son implícitamente públicos, estáticos y finales. El uso de estos modificadores en una declaración de constante en un interface está desaconsejado por falta de estilo. Similarmente, todos los métodos declarados en un interface son implícitamente públicos y abstractos.

Este código define un nuevo interface llamado `coleccion` que contiene un valor constante y tres declaraciones de métodos:

```
interface coleccion {
    int MAXIMO = 500;

    void añadir(Object obj);
    void borrar(Object obj);
    Object buscar(Object obj);
    int contadorActual();
}
```

El interface anterior puede ser implementado por cualquier clase que represente una colección de objetos como pueden ser pilas, vectores, enlaces, etc...

Observa que cada declaración de método está seguida por un punto y coma (;) porque un interface no proporciona implementación para los métodos declarados dentro de él.

Implementar un Interface

Para utilizar un interface se debe escribir una clase que lo implemente. Una clase declara todos los interfaces que implementa en su declaración de clase. Para declarar que una clase implementa uno o más interfaces, se utiliza la palabra clave `implements` seguida por una lista delimitada por comas con los interfaces implementados por la clase.

Por ejemplo, consideremos el interface `coleccion` presentado en la página [anterior](#). Ahora, supongamos que queremos escribir una clase que implemente un pila FIFO (primero en entrar, primero en salir). Como una pila FIFO contiene otros objetos tiene sentido que implemente el interface `coleccion`. La clase `PilaFIFO` declara que implementa el interface `coleccion` de esta forma:

```
class PilaFIFO implements coleccion {  
    . . .  
    void añadir(Object obj) {  
        . . .  
    }  
    void borrar(Object obj) {  
        . . .  
    }  
    Object buscar(Object obj) {  
        . . .  
    }  
    int contadorActual() {  
        . . .  
    }  
}
```

así se garantiza que proporciona implementación para los métodos `añadir()`, `borrar()`, `buscar()` y `contadorActual()`.

Por convención, la clausula `implements` sigue a la clausula `extends` si es que ésta existe.

Observa que las firmas de los métodos del interface `coleccion` implementados en la clase `PilaFIFO` debe corresponder exactamente con las firmas de los métodos declarados en la interface `coleccion`.

Utilizar un Interface como un Tipo

Como se mencionó anteriormente, cuando se define un nuevo interface, en esencia se está definiendo un tipo de referencia. Se pueden utilizar los nombres de interface en cualquier lugar donde se usaría un nombre de dato de tipos primitivos o un nombre de datos del tipo de referencia.

Por ejemplo, supongamos que se ha escrito un programa de hoja de cálculo que contiene un conjunto tabular de celdas y cada una contiene un valor. Querríamos poder poner cadenas, fechas, enteros, ecuaciones, en cada una de las celdas de la hoja. Para hacer esto, las cadenas, las fechas, los enteros y las ecuaciones tienen que implementar el mismo conjunto de métodos. Una forma de conseguir esto es encontrar el ancestro común de las clases e implementar ahí los métodos necesarios. Sin embargo, esto no es una solución práctica porque el ancestro común más frecuente es `Object`. De hecho, los objetos que puede poner en las celdas de su hoja de cálculo no están relacionadas entre sí, sólo por la clase `Object`. Pero no puede modificar `Object`.

Una aproximación podría ser escribir una clase llamada `ValordeCelda` que representara los valores que pudiera contener una celda de la hoja de cálculo. Entonces se podrían crear distintas subclases de `ValordeCelda` para las cadenas, los enteros o las ecuaciones. Además de ser mucho trabajo, esta aproximación arbitraria fuerza una relación entre esas clases que de otra forma no sería necesaria, y debería duplicar e implementar de nuevo clases que ya existen.

Se podría definir un interface llamado `CellAble` que se parecería a esto:

```
interface CellAble {  
    void draw();  
    void toString();  
    void toFloat();  
}
```

Ahora, supongamos que existen objetos `Línea` y `Columna` que contienen un conjunto de objetos que implementan el interface `CellAble`. El método `setObjectAt()` de la clase `Línea` se podría parecer a esto:

```
class Línea {  
    private CellAble[] contents;  
    . . .  
    void setObjectAt(CellAble ca, int index) {  
        . . .  
    }  
    . . .  
}
```

Observa el uso del nombre del interface en la declaración de la variable miembro `contents` y en la declaración del argumento `ca` del método. Cualquier objeto que implemente el interface `CellAble`, sin importar que exista o no en el árbol de clases,

puede estar contenido en el array contents y podría ser pasado al método setObjectAt().

Ozito

Crear Paquetes

Los paquetes son grupos relacionados de clases e interfaces y proporcionan un mecanismo conveniente para manejar un gran juego de clases e interfaces y evitar los conflictos de nombres. Además de los paquetes de Java puede crear tus propios paquetes y poner en ellos definiciones de clases y de interfaces utilizando la sentencia `package`.

Supongamos que se está implementando un grupo de clases que representan una colección de objetos gráficos como círculos, rectángulos, líneas y puntos. Además de estas clases tendrás que escribir un interface `Draggable` para que las clases que lo implementen puede moverse con el ratón. Si quieres que estas clases estén disponibles para otros programadores, puedes empaquetarlas en un paquete, digamos, `graphics` y entregar el paquete a los programadores (junto con alguna documentación de referencia como qué hacen las clases y los interfaces y qué interfaces de programación son públicos).

De esta forma, otros programadores pueden determinar fácilmente para qué es tu grupo de clases, cómo utilizarlos, y cómo relacionarlos unos con otros y con otras clases y paquetes. Los nombres de clases no tienen conflictos con los nombres de las clases de otros paquetes porque las clases y los interfaces dentro de un paquete son referenciados en términos de su paquete (técnicamente un paquete crea un nuevo espacio de nombres).

Se declara un paquete utilizando la sentencia `package`:

```
package graphics;

interface Draggable {
    . . .
}

class Circle {
    . . .
}

class Rectangle {
    . . .
}
```

La primera línea del código anterior crea un paquete llamado `graphics`. Todas las clases e interfaces definidas en el fichero que contiene esta sentencia son miembros del paquete. Por lo tanto, `Draggable`, `Circle`, y `Rectangle` son miembros del paquete `graphics`.

Los ficheros `.class` generados por el compilador cuando se compila el fichero que contiene el fuente para `Draggable`, `Circle` y `Rectangle` debe situarse en un directorio llamado `graphics` en algún lugar se el path `CLASSPATH`. `CLASSPATH` es una

lista de directorios que indica al sistema donde ha instalado varias clases e interfaces compiladas Java. Cuando busque una clase, el intérprete Java busca un directorio en su CLASSPATH cuyo nombre coincida con el nombre del paquete del que la clase es miembro. Los ficheros .class para todas las clases e interfaces definidas en un paquete deben estar en ese directorio de paquete.

Los nombres de paquetes pueden contener varios componentes (separados por puntos). De hecho, los nombres de los paquetes de Java tienen varios componentes: java.util, java.lang, etc... Cada componente del nombre del paquete representa un directorio en el sistema de ficheros. Así, los ficheros .class de java.util están en un directorio llamado util en otro directorio llamado java en algún lugar del CLASSPATH.

CLASSPATH

Para ejecutar una aplicación Java, se especifica el nombre de la aplicación Java que se desea ejecutar en el interprete Java. Para ejecutar un applet, se especifica el nombre del applet en una etiqueta <APPLET> dentro de un fichero HTML. El navegador que ejecute el applet pasa el nombre del applet al intérprete Java. En cualquier caso, la aplicación o el applet que se está ejecutando podría estar en cualquier lugar del sistema o de la red. Igualmente, la aplicación o el applet pueden utilizar otras clases y objetos que están en la misma o diferentes localizaciones.

Como las clases pueden estar en cualquier lugar, se debe indicar al interprete Java donde puede encontrarlas. Se puede hacer esto con la variable de entorno CLASSPATH que comprende una lista de directorios que contienen clases Java compiladas. La construcción de CLASSPATH depende de cada sistema.

Cuando el interprete obtiene un nombre de clase, desde la línea de comandos, desde un navegador o desde una aplicación o un applet, el interprete busca en todos los directorios de CLASSPATH hasta que encuentra la clase que está buscando.

Se deberá poner el directorio de nivel más alto que contiene las clases Java en el CLASSPATH. Por convención, mucha gente tiene un directorio de clases en su directorio raíz donde pone todo su código Java. Si tu tienes dicho directorio, deberías ponerlo en el CLASSPATH. Sin embargo, cuando se trabaja con applets, es conveniente poner el applet en un directorio clases debajo del directorio donde está el fichero HTML que contiene el applet. Por esta, y otras razones, es conveniente poner el directorio actual en el CLASSPATH.

Las clases incluidas en el entorno de desarrollo Java están disponibles automáticamente porque el interprete añade el directorio correcto a al CLASSPATH cuando arranca.

Observa que el orden es importante. Cuando el interprete Java está buscando una clase, busca por orden en los directorios indicados en CLASSPATH hasta que encuentra la clase con el nombre correcto. El interprete Java ejecuta la primera clase con el nombre correcto que encuentre y no busca en el resto de directorios. Normalmente es mejor dar a las clases nombres únicos, pero si no se puede evitar, asegurate de que el CLASSPATH busca las clases en el orden apropiado. Recuerda esto cuando selecciones tu CLASSPATH y el árbol del código fuente.

Nota: Todas las clases e interfaces pertenecen a un paquete. Incluso si no especifica uno con la sentencia package. Si no se especifica las clases e interfaces se convierten en miembros del paquete por defecto, que no tiene nombre y que siempre es importado.

Ozito

Utilizar Clases e Interfaces desde un Paquete

Para importar una clase específica o un interface al fichero actual (como la clase Circle desde el paquete graphics creado en la sección anterior) se utiliza la sentencia de import:

```
import graphics.Circle;
```

Esta sentencia debe estar al principio del fichero antes de cualquier definición de clase o de interface y hace que la clase o el interface esté disponible para su uso por las clases y los interfaces definidos en el fichero.

Si se quiere importar todas las clases e interfaces de un paquete, por ejemplo, el paquete graphics completo, se utiliza la sentencia import con un caracter comodín, un asterisco '*'.

```
import graphics.*;
```

Si intenta utilizar una clase o un interface desde un paquete que no ha sido importado, el compilador mostrará este error:

```
testing.java:4: Class Date not found in type declaration.  
    Date date;  
    ^
```

Observa que sólo las clases e interfaces declarados como públicos pueden ser utilizados en clases fuera del paquete en el fueron definidos.

El paquete por defecto (un paquete sin nombre) siempre es importado. El sistema de ejecución también importa automáticamente el paquete java.lang.

Si por suerte, el nombre de una clase de un paquete es el mismo que el nombre de una clase en otro paquete, se debe evitar la ambigüedad de nombres precediendo el nombre de la clase con el nombre del paquete. Por ejemplo, previamente se ha definido una clase llamada Rectangle en el paquete graphics. El paquete java.awt también contiene una clase Rectangle. Si estos dos paquetes son importados en la misma clase, el siguiente código sería ambiguo:

```
Rectangle rect;
```

En esta situación se tiene que ser más específico e indicar exactamente que clase Rectangle se quiere:

```
graphics.Rectangle rect;
```

Se puede hacer esto anteponiendo el nombre del paquete al nombre de la clase y separando los dos con un punto.

Los paquetes de Java

El entorno de desarrollo standard de Java comprende ocho paquetes.

El Paquete de Lenguaje Java

El paquete de lenguaje Java, también conocido como `java.lang`, contiene las clases que son el corazón del lenguaje Java. Las clases de este paquete se agrupan de la siguiente manera:

Object

El abuelo de todas las clases--la clase de la que parten todas las demás. Esta clase se cubrió anteriormene en la lección [La Clase Object](#).

Tipos de Datos Encubiertos

Una colección de clases utilizadas para encubrir variables de tipos primitivos: `Boolean`, `Character`, `Double`, `Float`, `Integer` y `Long`. Cada una de estas clases es una subclase de la clase abstracta `Number`.

Strings

Dos clases que implementan los datos de caracteres. [Las Clases String y StringBuffer](#) es una lección donde aprenderás el uso de estos dos tipos de Strings.

System y Runtime

Estas dos clases permiten a los programas utilizar los recursos del sistema. `System` proporciona un interface de programación independiente del sistema para recursos del sistema y `Runtime` da acceso directo al entorno de ejecución específico de un sistema. [Utilizar Recursos del Sistema](#) Describe las clases `System` y `Runtime` y sus métodos.

Thread

Las clases `Thread`, `ThreadDeath` y `ThreadGroup` implementan las capacidades multitareas tan importantes en el lenguaje Java. El paquete `java.lang` también define el interface `Runnable`. Este interface es conveniente para activar la clase Java sin subclasificar la clase `Thread`. A través de un ejemplo de aproximación [Threads de Control](#) te enseñará los Threads Java.

Class

La clase `Class` proporciona una descripción en tiempo de ejecución de una clase y la clase `ClassLoader` permite cargar clases en los programas durante la ejecución.

Math

Una librería de rutinas y valores matemáticos como pi.

Exceptions, Errors y Throwable

Cuando ocurre un error en un programa Java, el programa lanza un objeto que indica qué problema era y el estado del interprete cuando ocurrió el error. Sólo los objetos derivados de la clase Throwable pueden ser lanzados. Existen dos subclasses principales de Throwable: Exception y Error. Exception es la forma que deben intentar capturar los programas normales. Error se utiliza para los errores catastróficos--los programas normales no capturan Errores. El paquete java.lang contiene las clases Throwable, Exception y Error, y numerosas subclasses de Exception y Error que representan problemas específicos. [Manejo de Errores Utilizando Excepciones](#) te muestra cómo utilizar las excepciones para manejar errores en sus programas Java.

Process

Los objetos Process representa el proceso del sistema que se crea cuando se utiliza el sistema en tiempo de ejecución para ejecutar comandos del sistema. El paquete java.lang define e implementa la clase genérica Process.

El compilador importa automáticamente este paquete. Ningún otro paquete se importa de forma automática.

El Paquete I/O de Java

El paquete I/O de Java (java.io) proporciona un juego de canales de entrada y salida utilizados para leer y escribir ficheros de datos y otras fuentes de entrada y salida. Las clases e interfaces definidos en java.io se cubren completamente en [Canales de Entrada y Salida](#).

El Paquete de Utilidades de Java

Este paquete, java.util, contiene una colección de clases útiles. Entre ellas se encuentran muchas estructuras de datos genéricas (Dictionary, Stack, Vector, Hashtable) un objeto muy útil para dividir cadenas y otro para la manipulación de calendarios. El paquete java.util también contiene el interface Observer y la clase Observable que permiten a los objetos notificarse unos a otros cuando han cambiado. Las clases de java.util no se cubren en este tutorial aunque algunos ejemplos utilizan estas clases.

El Paquete de Red de Java

El paquete `java.net` contiene definiciones de clases e interfaces que implementan varias capacidades de red. Las clases de este paquete incluyen una clase que implementa una conexión URL. Se puede utilizar estas clases para implementar aplicaciones cliente-servidor y otras aplicaciones de comunicaciones. [Conectividad y Seguridad del Cliente](#) tiene varios ejemplos de utilización de estas clases, incluyendo un ejemplo cliente-servidor que utiliza datagramas.

El Paquete Applet

Este paquete contiene la clase `Applet` -- la clase que se debe subclasificar si se quiere escribir un applet. En este paquete se incluye el interface `AudioClip` que proporciona una abstracción de alto nivel para audio. [Escribir Applets](#).

Los Paquetes de Herramientas para Ventanas Abstractas

Tres paquetes componen las herramientas para Ventanas Abstractas: `java.awt`, `java.awt.image`, y `java.awt.peer`.

El paquete AWT

El paquete `java.awt` proporciona elementos GUI utilizados para obtener información y mostrarla en la pantalla como ventanas, botones, barras de desplazamiento, etc..

El paquete AWT Image

El paquete `java.awt.image` contiene clases e interfaces para manejar imágenes de datos, como la selección de un modelo de color, el cortado y pegado, el filtrado de colores, la selección del valor de un pixel y la grabación de partes de la pantalla.

El paquete AWT Peer

El paquete `java.awt.peer` contiene clases e interfaces que conectan los componentes AWT independientes de la plataforma a su implementación dependiente de la plataforma (como son los controles de Microsoft Windows).

Cambios en el JDK 1.1 que afectan a: *Objetos, Clases, e Interfaces*

[La declaración de Clase](#)

El JDK 1.1 permite declarar clases internas. Puedes ver [Cambios en el JDK 1.1: Clases Internas](#).

[Los paquetes de Java](#)

Se han añadido quince nuevos paquetes al juego conjunto java.* y se ha eliminado uno. Puedes ver [Cambios en el JDK 1.1: los Paquetes java.*](#).

[El Paquete java.lang](#)

Se han añadido tres clases al paquete java.lang. Puedes ver [Cambios en el JDK 1.1: el paquete java.lang](#).

[El paquete java.io](#)

Se han añadido nuevas clases al paquete java.io para soportar la lectura y escritura de caracteres de 16 Bits Unicode. Puedes ver [Cambios en el JDK 1.1: el paquete java.io](#).

[El paquete java.util](#)

Se han añadido nuevas clases al paquete java.util para soporte de internacionalización y manejo de eventos. Puedes ver [Cambios en el JDK 1.1: el paquete java.util](#).

[El paquete java.net](#)

Se han añadido muchas clases al paquete java.net.

[El paquete Applet](#)

Para un sumario de los cambios en el paquete java.applet puedes ver, [PENDIENTE].

[El Paquete java.awt](#)

Para informaicón sobre los cambios en los paquetes java.awt puedes ver las páginas [Cambios en el GUI: el AWT crece](#)

Las Clases String y StringBuffer

El paquete `java.lang` contiene dos clases de cadenas: `String` y `StringBuffer`. Ya hemos visto la clase `String` en varias ocasiones en este tutorial. La clase `String` se utiliza cuando se trabaja con cadenas que no pueden cambiar. Por otro lado, `StringBuffer`, se utiliza cuando se quiere manipular el contenido de una cadena.

El método `reverseIt()` de la siguiente clase utiliza las clases `String` y `StringBuffer` para invertir los caracteres de una cadena. Si tenemos una lista de palabras, se puede utilizar este método en conjunción de un pequeño programa para crear una lista de palabras rítmicas (una lista de palabras ordenadas por las sílabas finales). Sólo se tienen que invertir las cadenas de la lista, ordenar la lista e invertir las cadenas otra vez.

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método `reverseIt()` acepta un argumento del tipo `String` llamado `source` que contiene la cadena que se va a invertir. El método crea un `StringBuffer`, `dest`, con el mismo tamaño que `source`. Luego hace un bucle inverso sobre los caracteres de `source` y los añade a `dest`, con lo que se invierte la cadena. Finalmente el método convierte `dest`, de `StringBuffer` a `String`.

Además de iluminar las diferencias entre `String` y `StringBuffer`, esta lección ilustra varias características de las clases `String` y `StringBuffer`: Creación de `Strings` y `StringBuffers`, utilizar métodos accesorios para obtener información sobre `String` o `StringBuffer`, modificar un `StringBuffer` y convertir un tipo `String` a otro.

¿Por qué dos Clases String?

```
class ReverseString {  
    public static String reverseIt(String source) {  
        int i, len = source.length();  
        StringBuffer dest = new StringBuffer(len);  
  
        for (i = (len - 1); i >= 0; i--) {  
            dest.append(source.charAt(i));  
        }  
        return dest.toString();  
    }  
}
```

El entorno de desarrollo Java proporciona dos clases para manipular y almacenar datos del tipo carácter: **String**, para cadenas constantes, y **StringBuffer**, para cadenas que pueden cambiar.

String se utiliza cuando no se quiere que cambie el valor de la cadena. Por ejemplo, si escribimos un método que necesite una cadena de caracteres y el método no va a modificar la cadena, deberíamos utilizar un objeto **String**. Normalmente, queremos utilizar **Strings** para pasar caracteres a un método y para devolver caracteres desde un método. El método `reverseIt()` toma un **String** como argumento y devuelve un **String**.

La clase **StringBuffer** proporcionada para cadenas variables; se utiliza cuando sabemos que el valor de la cadena puede cambiar. Normalmente utilizaremos **StringBuffer** para construir datos de caracteres como en el método `reverseIt()`.

Como son constantes, los **Strings** son más económicos (utilizan menos memoria) que los **StringBuffers** y pueden ser compartidos. Por eso es importante utilizar **String** siempre que sea apropiado.

Crear String y StringBuffer

```
class ReverseString {  
    public static String reverseIt(String source) {  
        int i, len = source.length();  
        StringBuffer dest = new StringBuffer(len);  
  
        for (i = (len - 1); i >= 0; i--) {  
            dest.append(source.charAt(i));  
        }  
        return dest.toString();  
    }  
}
```

El método `reverseIt()` crea un `StringBuffer` llamado `dest` cuya longitud inicial es la misma que la de `source`. `StringBuffer dest` declara al compilador que `dest` se va a utilizar para referirse a un objeto del tipo `StringBuffer`, el operador `new` asigna memoria para un nuevo objeto y `StringBuffer(len)` inicializa el objeto. Estos tres pasos--declaración, ejemplarización e inicialización-- se describen en: [Crear Objetos](#).

Crear un String

Muchos `Strings` se crean a partir de cadenas literales. Cuando el compilador encuentra una serie de caracteres entre comillas (" y "), crea un objeto `String` cuyo valor es el propio texto. Cuando el compilador encuentra la siguiente cadena, crea un objeto `String` cuyo valor es `Hola Mundo`.

```
"Hola Mundo."
```

También se pueden crear objetos `String` como se haría con cualquier otro objeto Java: utilizando `new`.

```
new String("Hola Mundo.");
```

Crear un StringBuffer

El método constructor utilizado por `reverseIt()` para inicializar `dest` requiere un entero como argumento que indique el tamaño inicial del nuevo `StringBuffer`.

```
StringBuffer(int length)
```

`reverseIt()` podría haber utilizado el constructor por defecto para dejar indeterminada la longitud del buffer hasta un momento posterior. Sin embargo, es más eficiente especificar la longitud del buffer si se conoce, en vez de asignar memoria cada vez que se añadan caracteres al buffer.

Métodos Accesores

```
class ReverseString {  
    public static String reverseIt(String source) {  
        int i, len = source.length();  
        StringBuffer dest = new StringBuffer(len);  
  
        for (i = (len - 1); i >= 0; i--) {  
            dest.append(source.charAt(i));  
        }  
        return dest.toString();  
    }  
}
```

Las variables de ejemplar de un objeto están encapsuladas dentro del objeto, ocultas en su interior, seguras frente a la inspección y manipulación por otros objetos. Con ciertas excepciones bien definidas, los métodos del objeto no son los únicos a través de los cuales un objeto puede inspeccionar o alterar las variables de otro objeto. La encapsulación de los datos de un objeto lo protege de la corrupción de otros objetos y oculta los detalles de implementación a los objetos extraños. Esta encapsulación de datos detrás de los métodos de un objeto es una de las piedras angulares de la programación orientada a objetos.

Los métodos utilizados para obtener información de un objeto son conocidos como métodos accesores. El método `reverseIt()` utiliza dos métodos accesores de `String` para obtener información sobre el string `source`.

Primero utiliza el método accesor: `length()` para obtener la longitud de la cadena `source`.

```
int len = source.length();
```

Observa que a `reverseIt()` no le importa si el `String` mantiene su longitud como un entero, como un número en coma flotante o incluso si calcula la longitud al vuelo. `reverseIt()` simplemente utiliza el interface público del método `length()` que devuelve la longitud del `String` como un entero. Es todo lo que necesita saber `reverseIt()`.

Segundo, utiliza el método accesor: `charAt()` que devuelve el carácter que está situado en la posición indicada en su argumento.

```
source.charAt(i)
```

El carácter devuelto por `charAt()` es el que se añade al `StringBuffer dest`. Como la variable del bucle `i` empieza al final de `source` y avanza hasta el principio de la cadena, los caracteres se añaden en orden inverso al `StringBuffer`.

Más Métodos Accesores

Además de `length()` y `charAt()`, `String` soporta otros métodos accesores que proporcionan acceso a subcadenas y que indican la posición de caracteres específicos en la cadena. `StringBuffer` tiene sus propios métodos accesores similares.

Modificar un StringBuffer

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método `reverseIt()` utiliza el método `append()` de `StringBuffer` para añadir un carácter al final de la cadena de destino: `dest`. Si la adición de caracteres hace que aumente el tamaño de `StringBuffer` más allá de su capacidad actual, el `StringBuffer` asigna más memoria. Como la asignación de memoria es una operación relativamente cara, debemos hacer un código más eficiente inicializando la capacidad del `StringBuffer` de forma razonable para el primer contenido, así minimizaremos el número de veces que se tendrá que asignar memoria.

Por ejemplo, el método `reverseIt()` construye un `StringBuffer` con una capacidad inicial igual a la de la cadena fuente, asegurándose sólo una asignación de memoria para `dest`.

La versión del método `append()` utilizado en `reverseIt()` es sólo uno de los métodos de `StringBuffer` para añadir datos al final de un `StringBuffer`. Existen varios métodos `append()` para añadir varios tipos, como `float`, `int`, `boolean`, e incluso objetos, al final del `StringBuffer`. El dato es convertido a cadena antes de que tenga lugar la operación de adición.

Insertar Caracteres

Algunas veces, podríamos querer insertar datos en el medio de un `StringBuffer`. Se puede hacer esto utilizando el método `insert()`. Este ejemplo ilustra cómo insertar una cadena dentro de un `StringBuffer`.

```
StringBuffer sb = new StringBuffer("Bebe Caliente!");
sb.insert(6, "Java ");
System.out.println(sb.toString());
```

Este retazo de código imprimirá:

```
Bebe Java Caliente!
```

Con muchos métodos `insert()` de `StringBuffer` se puede especificar el índice anterior donde se quiere insertar el dato. En el ejemplo anterior: "Java " tiene que insertarse antes de la 'C' de "Caliente". Los índices

empiezan en 0, por eso el índice de la 'C' es el 6. Para insertar datos al principio de un StringBuffer se utiliza el índice 0. Para añadir datos al final del StringBuffer se utiliza un índice con la longitud actual del StringBuffer o `append()`.

Seleccionar Caracteres

Otro modificador muy útil de StringBuffer es `setCharAt()`, que selecciona un carácter en la posición especificada del StringBuffer. `setCharAt()` es útil cuando se reutiliza un StringBuffer.

Convertir Objetos a Strings

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El Método toString()

A veces es conveniente o necesario convertir un objeto a una cadena o String porque se necesitará pasarlo a un método que sólo acepta Strings. Por ejemplo, `System.out.println()` no acepta `StringBuffers`, por lo que necesita convertir el `StringBuffer` a `String` para poder imprimirlo. El método `reverseIt()` utiliza el método `toString()` de `StringBuffer` para convertirlo en un `String` antes de retornar.

```
return dest.toString();
```

Todas las clases heredan `toString()` desde la clase `Object` y muchas clases del paquete `java.lang` sobrescriben este método para proporcionar una implementación más acorde con la propia clase. Por ejemplo, las clases `Character`, `Integer`, `Boolean`, etc.. sobrescriben `toString()` para proporcionar una representación en `String` de los objetos.

El Método valueOf()

Como es conveniente, la clase `String` proporciona un método estático `valueOf()`. Se puede utilizar este método para convertir variables de diferentes tipos a un `String`. Por ejemplo, para imprimir el número pi:

```
System.out.println(String.valueOf(Math.PI));
```

Convertir Cadenas a Números

La clase `String` no proporciona ningún método para convertir una cadena en un número. Sin embargo, cuatro clases de los "tipos envolventes" (`Integer`, `Double`, `Float`, y `Long`) proporcionan unos métodos de clase llamados `valueOf()` que convierten una cadena en un objeto de ese

tipo. Aquí tenemos un pequeño ejemplo del método `valueOf()` de la clase `Float`:

```
String piStr = "3.14159";  
Float pi = Float.valueOf(piStr);
```

Ozito

Los Strings y el Compilador de Java

El compilador de Java utiliza las clases `String` y `StringBuffer` detrás de la escena para manejar las cadenas literales y la concatenación.

Cadenas Literales

En Java se deben especificar las cadenas literales entre comillas:

```
"Hola Mundo!"
```

Se pueden utilizar cadenas literales en cualquier lugar donde se pueda utilizar un objeto `String`. Por ejemplo, `System.out.println()` acepta un argumento `String`, por eso se puede utilizar una cadena literal en su lugar:

```
System.out.println("Hola Mundo!");
```

También se pueden utilizar los métodos de `String` directamente desde una cadena literal:

```
int len = "Adios Mundo Cruel".length();
```

Como el compilador crea automáticamente un nuevo objeto `String` para cada cadena literal que se encuentra, se puede utilizar una cadena literal para inicializar un `String`:

```
String s = "Hola Mundo";
```

El constructor anterior es equivalente pero mucho más eficiente que este otro, que crea dos objetos `String` en vez de sólo uno:

```
String s = new String("Hola Mundo");
```

El compilador crea la primera cadena cuando encuentra el literal `"Hola Mundo!"`, y la segunda cuando encuentra `new String()`.

Concatenación y el Operador +

En Java, se puede utilizar el operador `+` para unir o concatenar cadenas:

```
String cat = "cat";  
System.out.println("con" + cat + "enacion");
```

Esto decepciona un poco porque, como ya se sabe, los `Strings` no pueden modificarse. Sin embargo detrás de la escena el compilador utiliza `StringBuffer` para implementar la concatenación. El ejemplo anterior se compilaría de la siguiente forma:

```
String cat = "cat";  
System.out.println(new StringBuffer().append("con").append(cat).append("enacion"));
```

También se puede utilizar el operador `+` para añadir valores a una cadena que no son propiamente cadenas:

```
System.out.println("Java's Number " + 1);
```

El compilador convierte el valor no-cadena (el entero 1 en el ejemplo anterior) a un objeto `String` antes de realizar la concatenación.

Cambios en el JDK 1.1: Que afectan a las clases String y StringBuffer

[Las clases String y StringBuffer](#)

La clase String ha cambiado para soportar la internacionalización.

Puedes ver [Cambios en el JDK 1.1: La clase String](#) para más detalles. La clase StringBuffer no se ha modificado en el JDK 1.1.

Seleccionar Atributos del Programa

Los programas Java se ejecutan dentro de algún entorno. Esto es, en un entorno donde hay una máquina, un directorio actual, preferencias del usuario, el color de la ventana, la fuente, el tamaño de la fuente y otros atributos ambientales.

Además de estos atributos del sistema, un programa puede activar ciertos atributos configurables específicos del programa. Los atributos del programa son frecuentemente llamados preferencias y permiten al usuario configurar varias opciones de arranque.

Un programa podría necesitar información sobre el entorno del sistema para tomar decisiones sobre algo o como hacerlo. Un programa también podría modificar ciertos atributos propios o permitir que el usuario los cambie. Por eso, un programa necesita poder leer y algunas veces modificar varios atributos del sistema y atributos específicos del programa. Los programas Java puede manejar atributos del programa a través de tres mecanismos: propiedades, argumentos de la línea de comandos de la aplicación y parámetros de applets.

- [Seleccionar y Utilizar Propiedades](#)
- [Argumentos de la Línea de Comandos de la Aplicación](#)
 - [Convenciones de Argumentos de la Línea de Comandos](#)
 - [Analizar los Argumentos de la Línea de Comandos](#)
- Parámetros de Applets Los parámetros de applets son similares a los argumentos de la línea de comandos pero se utilizan con applets, no con aplicaciones. Estos parámetros se utilizan para seleccionar uno o más atributos de una llamada al applet. Estos parámetros se verán más adelante cuando tratemos con los applets.
- [Notas sobre el JDK 1.1](#)

Seleccionar y Utilizar Propiedades

En Java, los atributos del programa están representados por la clase `Properties` del paquete `java.util`. Un objeto `Properties` contiene un juego de parejas clave/valor. Estas parejas clave/valor son como las entradas de un diccionario: la clave es la palabra, y el valor es la definición.

Tanto la clave como el valor son cadenas. Por ejemplo, `os.name` es la clave para una de las propiedades del sistema por defecto de Java--el valor contiene el nombre del sistema operativo actual. Utilice la clave para buscar una propiedad en la lista de propiedades y obtener su valor. En mi sistema, cuando busco la propiedad `os.name`, su valor `Windows 95/NT`. El tuyo probablemente será diferente.

Las propiedades específicas de un programa deben ser mantenidas por el propio programa. Las propiedades del sistema las mantiene la clase `java.lang.System`. Para más información sobre las propiedades del sistema puedes referirte a: [Propiedades del Sistema](#) en la lección [Utilizar los Recursos del Sistema](#).

Se puede utilizar la clase `Properties` del paquete `java.util` para manejar atributos específicos de un programa. Se puede cargar los pares clave/valor dentro de un objeto `Properties` utilizando un stream, grabar las propiedades a un stream y obtener información sobre las propiedades representadas por el objeto `Properties`.

Seleccionar un Objeto `Properties`

Frecuentemente, cuando un programa arranca, utiliza un código similar a este para seleccionar un objeto `Properties`:

```
. . .
    // selecciona las propiedades por defecto
    Properties defaultProps = new Properties();
    FileInputStream defaultStream = new FileInputStream("defaultProperties");
    defaultProps.load(defaultStream);
    defaultStream.close();

    // selecciona las propiedades reales
    Properties applicationProps = new Properties(defaultProps);
    FileInputStream appStream = new FileInputStream("appProperties");
    applicationProps.load(appStream);
    appStream.close();
. . .
```

Primero la aplicación selecciona un objeto `Properties` para las propiedades por defecto. Este objeto contiene un conjunto de propiedades cuyos valores no se utilizan explícitamente en ninguna parte. Este fragmento de código utiliza el método `load()` para leer el valor por defecto desde un fichero de disco llamado `defaultProperties`. Normalmente las aplicaciones guardan sus propiedades en ficheros de disco.

Luego, la aplicación utiliza un constructor diferente para crear un segundo objeto `Properties applicationProps`. Este objeto utiliza `defaultProps` para proporcionarle sus valores por defecto.

Después el código carga un juego de propiedades dentro de applicationProps desde un fichero llamado appProperties. Las propiedades cargadas en appProperties pueden seleccionarse en base al usuario o en base al sistema, lo que sea más apropiado para cada programa. Lo que es importante es que el programa guarde las Propiedades en un posición "CONOCIDA" para que la próxima llamada al programa pueda recuperarlas. Por ejemplo, el navegador HotJava graba las propiedades en el directorio raíz del usuario.

Se utiliza el método save() para escribir las propiedades en un canal:

```
FileOutputStream defaultsOut = new FileOutputStream("defaultProperties");
applicationProps.save(defaultsOut, "---No Comment---");
defaultsOut.close();
```

El método save() necesita un stream donde escribir, y una cadena que se utiliza como comentario al principio de la salida.

Obtener Información de las Propiedades

Una vez que se han seleccionado las propiedades de un programa, se puede pedir información sobre las propiedades contenidas. La clase Properties proporciona varios métodos para obtener esta información de las propiedades:

getProperty() (2 versiones)

Devuelve el valor de la propiedad especificada. Una versión permite un valor por defecto, si no se encuentra la clave, se devuelve el valor por defecto.

list()

Escribe todas las propiedades en el canal especificado. Esto es útil para depuración.

propertyNames()

Devuelve una lista con todas las claves contenidas en el objeto Properties.

Consideraciones de Seguridad: Observa que el acceso a las propiedades está sujeto a la aprobación del manejador de Seguridad. El programa de ejemplo es una aplicación solitaria, que por defecto, no tiene manejador de seguridad. Si se intenta utilizar este código en un applet, podría no trabajar dependiendo del navegador. Puedes ver: [Entender las Capacidades y Restricciones de un Applet](#) para obtener más información sobre las restricciones de seguridad en los applets.

Argumentos de la Línea de Comandos

Una aplicación Java puede aceptar cualquier número de argumentos desde la línea de comandos. Los argumentos de la línea de comandos permiten al usuario variar la operación de una aplicación, Por ejemplo, una aplicación podría permitir que el usuario especificara un modo verboso--esto es, especificar que la aplicación muestre toda la información posible-- con el argumento `-verbose`.

Cuando llama a una aplicación, el usuario teclea los argumentos de la línea de comandos después del nombre de la aplicación. Supongamos, por ejemplo, que existe una aplicación Java, llamada `Sort`, que ordena las líneas de un fichero, y que los datos que se quiere ordenar están en un fichero llamado `friends.txt`. Si estuviéramos utilizando Windows 95/NT, llamaría a la aplicación `Sort` con su fichero de datos de la siguiente forma:

```
C:\> java Sort friends.txt
```

En el lenguaje Java, cuando se llama a una aplicación, el sistema de ejecución pasa los argumentos de la línea de comandos al método `main` de la aplicación, mediante un array de `Strings`. Cada `String` del array contiene un argumento. En el ejemplo anterior, los argumentos de la línea de comandos de la aplicación `Sort` son un array con una sola cadena que contiene `"friends.txt"`.

Ejemplo de Argumentos

Esta sencilla aplicación muestra todos los argumentos de la línea de comandos uno por uno en cada línea:

```
class Echo {  
    public static void main (String[] args) {  
        for (int i = 0; i < args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

Intenta Esto: Llama a la aplicación `Echo`. Aquí tienes un ejemplo de como llamarla utilizando Windows 95/NT:

```
C:\> java Echo Bebe Java Caliente  
Bebe  
Java  
Caliente
```

Habrás observado que la aplicación muestra cada palabra en una línea distinta. Esto es así porque el espacio separa los argumentos de la línea de comandos. Si quieres que `Bebe Java Caliente` sea interpretado como un sólo argumento debes ponerlo entre comillas:

```
% java Echo "Bebe Java Caliente"  
Bebe Java Caliente
```

Convenciones

Existen varias convenciones que se deberán observar cuando se acepten y procesen argumentos de la línea de comandos con una aplicación Java.

Analizar Argumentos de la Línea de Comandos

La mayoría de las aplicaciones aceptan varios argumentos de la línea de comandos que le permiten al usuario variar la ejecución de la aplicación, Por ejemplo, el comando UNIX que imprime el contenido de un directorio-- `ls --` acepta argumentos que determinan qué atributos de ficheros se van a mostrar y el orden en que lo van a hacer.

Normalmente, el usuairo puede especificar los argumentos en cualquier orden por lo tanto requiere que la aplicación sea capaz de analizarlos.

Convenciones para los Argumentos de la Línea de Comandos

El lenguaje Java sigue las convenciones de UNIX que definen tres tipos diferentes de argumentos:

- [Palabras](#) (también conocidos como opciones)
- [Argumentos que requieren argumentos](#)
- [Banderas](#)

Además, una aplicación debe observar las siguientes convenciones para utilizar los argumentos de la línea de comandos en Java:

- El guión (-) precede a las opciones, banderas o series de banderas.
- Los argumentos pueden ir en cualquier orden, excepto cuando sea un argumento que requiere otros argumentos.
- Las banderas pueden listarse en cualquier orden, separadamente o combinadas: `-xn o -nx o -x -n`.
- Típicamente los nombres de fichero van al final.
- El programa imprime un mensaje de error de utilización cuando no se reconoce un argumento de la línea de comandos. Estas sentencias pueden tener esta forma:

utilización: `nombre_aplicación [argumentos_opcionales] argumentos_requeridos`

Opciones

Los argumentos como `-verbose` son argumentos de palabra y deben especificarse completamente en la línea de comandos. Por ejemplo, `-ver` no correspondería con `-verbose`.

Se pueden utilizar sentencias como esta para comprobar los argumentos de palabras:

```
if (argument.equals("-verbose"))  
    vflag = true;
```

Esta sentencia comprueba si la palabra `-verbose` está en la línea de argumentos y activa una bandera en el programa para que este se ejecute en modo verbose.

Argumentos que Requieren Argumentos

Algunos argumentos necesitan más información. Por ejemplo, un argumento como `-output` podría permitir que el usuario redirigiera la salida del programa. Sin embargo, la opción `-output` en solitario no ofrece la información suficiente a la aplicación: ¿Cómo sabe la aplicación dónde redirigir la salida? Por lo tanto el usuario debe especificar también un nombre de fichero. Normalmente, el ítem siguiente de la línea de comandos proporciona la información adicional para el argumento que así lo requiere. Se puede utilizar la siguiente sentencia para emparejar argumentos que requieren argumentos:

```
if (argument.equals("-output")) {  
    if (nextarg < args.length)  
        outputfile = args[nextarg++];  
    else  
        System.err.println("-output requiere un nombre de fichero");  
}
```

Observa que el código se asegura de que el usuario ha especificado realmente un

argumento siguiente antes de intentar utilizarlo.

Banderas

Las banderas son caracteres que modifican el comportamiento del programa de alguna manera. Por ejemplo, la bandera `-t` proporcionada al comando `ls` de UNIX indica que la salida debe ordenarse por la fecha de los ficheros. La mayoría de las aplicaciones permiten al usuario especificar banderas separadas en cualquier orden:

`-x -n` `-n -x`

Además, para hacer un uso más sencillo, las aplicaciones deberán permitir que el usuario concatene banderas y las especifique en cualquier orden:

`-nx` `-xn`

El programa de ejemplo descrito en la página [siguiente](#) implementa un sencillo algoritmo para procesar banderas que pueden especificarse en cualquier orden, separadas o de forma combinada.

Analizar Argumentos de la Línea de Comandos

Este programa, llamado ParseCmdLine, proporciona una forma básica para construir tu propio analizador de argumentos:

```
class ParseCmdLine {
    public static void main(String[] args) {

        int i = 0, j;
        String arg;
        char flag;
        boolean vflag = false;
        String ficheroSalida = "";

        while (i < args.length && args[i].startsWith("-")) {
            arg = args[i++];

            // Utiliza este tipo de chequeo para argumentos de "palabra"
            if (arg.equals("-verboso")) {
                System.out.println("modo verboso mode activado");
                vflag = true;
            }

            // Utiliza este tipo de chequeo para argumentos que requieren argumentos
            else if (arg.equals("-output")) {
                if (i < args.length)
                    ficheroSalida = args[i++];
                else
                    System.err.println("-output requiere un nombre de fichero");
                if (vflag)
                    System.out.println("Fichero de Salida = " + ficheroSalida);
            }

            // Utiliza este tipo de chequeo para una serie de banderas
            else {
                for (j = 1; j < arg.length(); j++) {
                    flag = arg.charAt(j);
                    switch (flag) {
                        case 'x':
                            if (vflag) System.out.println("Opción x");
                            break;
                        case 'n':
                            if (vflag) System.out.println("Opción n");
                            break;
                        default:
                            System.err.println("ParseCmdLine: opción ilegal " + flag);
                            break;
                    }
                }
            }
        }

        if (i == args.length)
            System.err.println("Utilización: ParseCmdLine [-verboso] [-xn]
```

```
                [-output unfichero] nombre de Fichero");  
    else  
        System.out.println("Correcto!");  
    }  
}
```

Acepta un argumento de cada uno de los tipos: un argumento de palabra, un argumento que requiere un argumento y dos banderas. Además, este programa necesita un nombre de fichero. Aquí tienes una sentencia de utilización de este programa:

Utilización: `ParseCmdLine [-verboso] [-xn] [-output unfichero] nombrefichero`

Los argumentos entre los corchetes son opciones: el argumento **nombrefichero** es obligatorio.

Ozito

Cambios del API que afectan a los *Atributos del Programa*

Se ha añadido una segunda lista de métodos a la clase Properties en la versión 1.1 del JDK. Puedes ver [Cambios en el JDK1.1: La clase Properties](#) para más detalles. for details.

Utilizar los Recursos del Sistema

Algunas veces, un programa necesita acceder a los recursos del sistema, como los canales de I/O estandar o la hora del sistema. Un programa podría utilizar estos recursos directamente desde el entorno de ejecución, pero sólo podría ejecutarse en el entorno donde fue escrito. Cada vez que se quisiera ejecutar un programa en un nuevo entorno se deberá "portar" el programa reescribiendo las secciones del código dependientes del sistema.

El entorno de desarrollo de Java resuelve este problema permitiendo a los programas que utilicen los recursos del sistema a través de un interface de programación independiente del sistema implementado por la clase [System](#) (miembro del paquete [java.lang](#)).

Utilizar la Clase System

Al contrario que la mayoría de las clases, no se debe ejemplarizar la clase System para utilizarla. Para ser más precisos, no se puede ejemplarizar-- es una clase final y todos sus constructores son privados.

Todas las variables y métodos de la clase System son métodos y variables de clase -- están declaradas como static. Para una completa explicación sobre las variables y métodos de clase y en qué se diferencian de las variables y métodos de ejemplar, puede referirse a [Miembros del Ejemplar y de la Clase](#).

Para utilizar una variable de clase, se usa directamente desde el nombre de la clase utilizando la notación de punto ('.') de Java. Por ejemplo, para referirse a la variable out de la clase System, se añade el nombre de la variable al nombre de la clase separados por un punto. Así:

```
System.out
```

Se puede llamar a los métodos de clase de una forma similar. Por ejemplo, para llamar al método getProperty() de la clase System se añade el nombre del método al nombre de la clase separados por un punto:

```
System.getProperty(argument);
```

El siguiente programa Java utiliza dos veces la clase System, primero para obtener el nombre del usuario actual y luego para mostrarlo.

```
class UserNameTest {
    public static void main(String[] args) {
        String name;
        name = System.getProperty("user.name");
        System.out.println(name);
    }
}
```

Habrás observado que el programa nunca ejemplariza un objeto de la clase System. Solo referencia al método getProperty() y la variable out directamente desde la clase.

El ejemplo anterior utiliza el método getProperty() para buscar en la base de datos de propiedades una propiedad llamada "user.name". [Propiedades del Sistema](#) más adelante en esta lección cuenta más cosas sobre las propiedades del sistema y el método getProperty().

El ejemplo también utiliza System.out, un PrintStream que implementa el canal de salida estandar. El método println() imprime el argumento en el canal de salida estandar. La [siguiente página](#) de esta lección explica el canal de salida estandar y los otros dos canales proporcionados por la clase System.

Los Canales de I/O Estandar

Los conceptos de los canales de I/O estandar son un concepto de la librería C que ha sido asimilado dentro del entorno Java. Existen tres canales estandar, todos los cuales son manejados por la clase `java.lang.System`:

Entrada estandar --referenciado por `System.in`

utilizado para la entrada del programa, típicamente lee la entrada introducida por el usuario.

Salida estandar --referenciado por `System.out`

utilizado para la salida del programa, típicamente muestra información al usuario.

Error estandar --referenciado por `System.err`

utilizado para mostrar mensajes de error al usuario.

Canal de Entrada Estandar

La clase `System` proporciona un canal para leer texto -- el canal de entrada estandar. El programa de ejemplo de Tuercas y Tornillos del Lenguaje Java utiliza el canal de entrada estandar para contar los caracteres tecleados por el usuario.

Los Canales de Salida y de Error Estandards

Probablemente los puntos más utilizados de la clase `System` sean los canales de salida y de error estandar, que se utilizan para mostrarle texto al usuario.

El canal de salida estandar se utiliza normalmente para las salidas de comandos, esto es, para mostrar el resultado de un comando del usuario. El canal de error estandar se utiliza para mostrar cualquier error que ocurra durante la ejecución del programa.

Los métodos `print()`, `println()`, y `write()`

Tanto la salida estandar como el error estandar derivan de la clase `PrintStream`. Así, se utiliza uno de los tres métodos de `PrintStream` para imprimir el texto en el canal: `print()`, `println()`, y `write()`.

Los métodos `print()` y `println()` son esencialmente el mismo: los dos escriben su argumento `String` en el canal. La única diferencia entre estos dos métodos es que `println()` añade un carácter de nueva línea al final de la salida, y `print()` no lo hace. En otras palabras, esto

```
System.out.print("Duke no es un Pingüino!\n");
```

es equivalente a esto

```
System.out.println("Duke no es un Pingüino!");
```

Observa el carácter extra `\n` en la llamada al primer método; es el carácter para nueva línea. `println()` añade automáticamente este carácter a su salida.

El método `write()` es menos utilizado que los anteriores y se utiliza para escribir bytes en un canal. `write()` se utiliza para escribir datos que no sean ASCII.

Argumentos para `print()` y `println()`

Los métodos `print()` y `println()` toman un sólo argumento. Este argumento puede ser cualquiera de los siguientes tipos: `Object`, `String`, `char[]`, `int`, `long`, `float`, `double`, y `boolean`. Además existe una versión extra de `println()` que no tiene argumentos que imprime una nueva línea en el canal.

Imprimir Objetos de Diferentes Tipos de Datos

El siguiente programa utiliza `println()` para sacar datos de varios tipos por el canal de salida estandard.

```
class DataTypePrintTest {
    public static void main(String[] args) {

        Thread objectData = new Thread();
        String stringData = "Java Mania";
        char[] charArrayData = { 'a', 'b', 'c' };
        int integerData = 4;
        long longData = Long.MIN_VALUE;
        float floatData = Float.MAX_VALUE;
        double doubleData = Math.PI;
        boolean booleanData = true;

        System.out.println(objectData);
        System.out.println(stringData);
        System.out.println(charArrayData);
        System.out.println(integerData);
        System.out.println(longData);
        System.out.println(floatData);
        System.out.println(doubleData);
        System.out.println(booleanData);
    }
}
```

El programa anterior tendría esta salida:

```
Thread[Thread-4,5,main]
```

```
Java Mania
abc
4
-9223372036854775808
3.40282e+38
3.14159
true
```

Observa que se puede imprimir un objeto -- el primer `println()` imprime un objeto `Thread` y el segundo imprime un objeto `String`. Cuando se utilice `print()` o `println()` para imprimir un objeto, los datos impresos dependen del tipo del objeto. En el ejemplo, imprimir un `String` muestra el contenido de la cadena. Sin embargo, imprimir un `Thread` muestra una cadena con este formato:

```
ThreadClass[nombre,prioridad,grupo]
```

Propiedades del Sistema

La clase System mantiene un conjunto de propiedades -- parejas clave/valor -- que definen atributos del entorno de trabajo actual. Cuando arranca el sistema de ejecución por primera vez, las propiedades del sistema se inicializan para contener información sobre el entorno de ejecución. Incluyendo información sobre el usuario actual, la versión actual del runtime de Java, e incluso el carácter utilizado para separar componentes en un nombre de fichero.

Aquí tiene una lista completa de las propiedades del sistema que puede obtener el sistema cuando arranca y lo que significan:

Clave	Significado	Acceden los Applets
-----	-----	-----
"file.separator"	File separator (e.g., "/")	si
"java.class.path"	Java classpath	no
"java.class.version"	Java class version number	si
"java.home"	Java installation directory	no
"java.vendor"	Java vendor-specific string	si
"java.vendor.url"	Java vendor URL	si
"java.version"	Java version number	si
"line.separator"	Line separator	si
"os.arch"	Operating system architecture	si
"os.name"	Operating system name	si
"path.separator"	Path separator (e.g., ":")	si
"user.dir"	User's current working directory	no
"user.home"	User home directory	no
"user.name"	User account name	no

Los programas Java pueden leer o escribir las propiedades del sistema a través de varios métodos de la clase System. Se puede utilizar una clave para buscar una propiedad en la lista de propiedades, o se puede obtener el conjunto completo de propiedades de una vez. También se puede cambiar el conjunto de propiedades completamente.

Consideraciones de Seguridad: Los Applets pueden acceder a las propiedades del sistema pero no a todas. Para obtener una lista completa de las propiedades del sistema que pueden y no pueden ser utilizadas por los applets, puedes ver : [Leer las Propiedades del Sistema](#). Los applets no pueden escribir las propiedades del sistema.

Leer las Propiedades del Sistema

La clase System tiene dos métodos que se pueden utilizar para leer las propiedades del sistema: `getProperty()` y `getProperties`.

La clase System tiene dos versiones diferentes de `getProperty()`. Ambas versiones devuelven el valor de la propiedad nombrada en la lista de argumentos. La más simple de las dos `getProperty()` toma un sólo argumento: la clave de la propiedad que quiere buscar. Por ejemplo, para obtener el valor de `path.separator`, utilizamos la siguiente sentencia:

```
System.getProperty("path.separator");
```

Este método devuelve una cadena que contiene el valor de la propiedad. Si la propiedad

no existe, esta versión de `getProperty()` devuelve `null`.

Lo que nos lleva a la siguiente versión de `getProperty()`. Esta versión requiere dos argumentos `String`: el primer argumento es la clave que buscamos y el segundo es el valor por defecto devuelto si la clave no se encuentra o no tiene ningún valor. Por ejemplo, esta llamada a `getProperty()` busca la propiedad del sistema llamada `subliminal.message`. Esto no es una propiedad válida del sistema, por lo que en lugar de devolver `null`, este método devolverá el valor proporcionado por el segundo argumento: `"Compra Java ahora"`.

```
System.getProperty("subliminal.message", "Compra Java ahora!");
```

Se deberá utilizar esta versión de `getProperty()` si no se quiere correr el riesgo de una excepción `NullPointerException`, o si realmente se quiere proporcionar un valor por defecto para una propiedad que no tiene valor o que no ha podido ser encontrada.

El último método proporcionado por la clase `System` para acceder a los valores de las propiedades es el método `getProperties()` que devuelve [Propiedad](#) un objeto que contiene el conjunto completo de las propiedades del sistema. Se pueden utilizar varios métodos de la clase `Properties` para consultar valores específicos o para listar el conjunto completo de propiedades. Para más información sobre la clase `Properties`, puedes ver [Seleccionar y utilizar Propiedades](#).

Escribir Propiedades del Sistema

Se puede modificar el conjunto existente de las propiedades del sistema, utilizando el método `setProperties()` de la clase `System`. Este método toma un objeto `Properties` que ha sido inicializado para contener parejas de clave/valor para las propiedades que se quieren modificar. Este método reemplaza el conjunto completo de las propiedades del sistema por las nuevas representadas por el objeto `Properties`.

Aquí tenemos un pequeño programa de ejemplo que crea un objeto `Properties` y lo inicializa desde un fichero:

```
subliminal.message=Buy Java Now!
```

El programa de ejemplo utiliza `System.setProperties()` para instalar el nuevo objeto `Properties` como el conjunto actual de propiedades del sistema.

```
import java.io.FileInputStream;
import java.util.Properties;

class PropertiesTest {
    public static void main(String[] args) {
        try {
            // selecciona el nuevo objeto propiedades a partir de
            "myProperties.txt"
            FileInputStream propFile = new FileInputStream("myProperties.txt");
            Properties p = new Properties(System.getProperties());
            p.load(propFile);

            // selecciona las propiedades del sistema
            System.setProperties(p);
            System.getProperties().list(System.out);    // selecciona las nuevas
propiedades
        } catch (java.io.FileNotFoundException e) {
            System.err.println("Can't find myProperties.txt.");
        } catch (java.io.IOException e) {
            System.err.println("I/O failed.");
        }
    }
}
```

```
}  
}
```

Observa que el programa de ejemplo crea un objeto `Properties`, `p`, que se utiliza como argumento para `setProperties()`:

```
Properties p = new Properties(System.getProperties());
```

Esta sentencia inicializa el nuevo objeto `Properties`, `p` con el conjunto actual de propiedades del sistema, que en el caso de este pequeño programa es el juego de propiedades inicializado por el sistema de ejecución. Luego el programa carga las propiedades adicionales en `p` desde el fichero `myProperties.txt` y selecciona las propiedades del sistema en `p`. Esto tiene el efecto de añadir las propiedades listadas en `myProperties.txt` al juego de propiedades creado por el sistema de ejecución durante el arranque. Observa que se puede crear `p` sin ningún objeto `Properties` como este:

```
Properties p = new Properties();
```

Si se hace esto la aplicación no tendrá acceso a las propiedades del sistema.

Observa también que las propiedades del sistema se pueden sobrescribir! Por ejemplo, si `myProperties.txt` contiene la siguiente línea, la propiedad del sistema `java.vendor` será sobrescrita:

```
java.vendor=Acme Software Company
```

En general, ten cuidado de no sobrescribir las propiedades del sistema.

El método `setProperties()` cambia el conjunto de las propiedades del sistema para la aplicación que se está ejecutando. Estos cambios no son persistentes. Esto es, cambiar las propiedades del sistema dentro de una aplicación no tendrá ningún efecto en próximas llamadas al intérprete Java para esta u otras aplicaciones. El sistema de ejecución re-inicializa las propiedades del sistema cada vez que arranca. Si se quiere que los cambios en las propiedades del sistema sean persistentes, se deben escribir los valores en un fichero antes de salir y leerlos de nuevo cuando arranque la aplicación.

Forzar la Finalización y la Recolección de Basura

El sistema de ejecución de Java realiza las tareas de manejo de memoria por tí. Cuando un programa ha terminado de utilizar un objeto-- esto es, cuando ya no hay más referencias a ese objeto- el objeto es finalizado y luego se recoge la basura.

Estas tareas suceden asíncronamente en segundo plano. Sin embargo, se puede forzar la finalización de un objeto y la recolección de basura utilizando los métodos apropiados de la clase System.

Finalizar Objetos

Antes de recolectar la basura de un objeto, el sistema de ejecución de Java le da la oportunidad de limpiarse a sí mismo. Este paso es conocido como finalización y se consigue mediante una llamada al método `finalize()` del objeto. El objeto debe sobrescribir este método para realizar cualquier tarea de limpieza final como la liberación de recursos del sistema como ficheros o conexiones. Para más información sobre el método `finalize()` puedes ver: [Escribir un método `finalize\(\)`](#).

Se puede forzar que ocurra la finalización de un objeto llamando al método `runFinalization()` de la clase System.

```
System.runFinalization();
```

Este método llama a los métodos `finalize()` de todos los objetos que están esperando para ser recolectados.

Ejecutar el Recolector de Basura

Se le puede pedir al recolector de basura que se ejecute en cualquier momento llamando al método `gc()` de la clase System:

```
System.gc();
```

Se podría querer ejecutar el recolector de basura para asegurarnos que lo hace en el mejor momento para el programa en lugar de hacerlo cuando le sea más conveniente al sistema de ejecución.

Por ejemplo, un programa podría desear ejecutar el recolector de basura antes de entrar en un cálculo o una sección de utilización de memoria extensiva, o cuando sepa que va a estar ocupado algún tiempo. El recolector de basura requiere unos 20 milisegundos para realizar su tarea, por eso un programa sólo debe ejecutarlo cuando no tenga ningún impacto en su programa -- esto es, que el programa anticipe que el recolector de basura va a tener tiempo suficiente para terminar su trabajo.

Otros Métodos de la Clase System

La clase System contiene otros métodos que proporcionan varias funcionalidades incluyendo la copia de arrays y la obtención de la hora actual.

Copiar Arrays

El método `arraycopy()` de la clase System se utiliza para realizar una copia eficiente de los datos de un array a otro. Este método requiere cinco argumentos:

```
public static
    void copiaarray(Object fuente, int indiceFuente, Object destino, int
indiceDestino, int longitud)
```

Los dos argumentos del tipo Object indican los array de origen y de destino. Los tres argumentos enteros indican la posición en los array de origen y destino y el número de elementos a copiar.

El siguiente programa utiliza `copiaarray()` para copiar algunos elementos desde `copiaDesde` a `copiaA`.

```
class Prueba {
    public static void main(String[] args) {
        byte[] copiaDesde = { 'd', 'e', 's', 'c', 'a', 'f', 'e', 'i', 'n', 'a', 'd',
'o' };
        byte[] copiaA = new byte[7];

        System.copiaarray(copiaDesde, 3, copiaA, 0, 7);
        System.out.println(new String(copiaA, 0));
    }
}
```

Con la llamada al método `copiaarray()` en este programa de ejemplo comienza la copia en el elemento número 3 del array fuente -- recuerda que los índices de los arrays empiezan en cero, por eso la copia empieza en el elemento 'c'. Luego `copiaarray()` pone los elementos copiados en la posición 0 del array destino `copiaA`. Copia 7 elementos: 'c', 'a', 'f', 'e', 'i', 'n', y 'a'. Efectivamente, el método `copiaarray()` saca "cafeina" de "descafeinado".

Observa que el array de destino debe ser asignado antes de llamar a `arraycopy()` y debe ser lo suficientemente largo para contener los datos copiados.

Obtener la Hora Actual

El método `currentTimeMillis()` devuelve la hora actual en milisegundos desde las 00:00:00 del 1 de Enero de 1970. Este método se utiliza comunmente en pruebas de rendimiento; obtener la hora actual, realizar la operación que se quiere controlar, obtener de nuevo la hora actual--la diferencia entre las dos horas es el tiempo que ha tardado en realizarse la operación.

En interfaces gráficos de usuarios el tiempo entre dos pulsaciones del ratón se utiliza para determinar si el usuario ha realizado un doble click. El siguiente applet utiliza `currentTimeMillis()` para calcular el número de milisegundos entre los dos clicks del ratón. Si el tiempo entre los dos clicks es menor de 200 milisegundos, los dos clicks de ratón se interpretan como un doble click.

Su navegador no entiende la etiqueta APPLET.

Aquí tienes el [código fuente](#) para el applet `TimingIsEverything`:

```
import java.awt.*;
```

```

public class TimingIsEverything extends java.applet.Applet {

    public long firstClickTime = 0;
    public String displayStr;

    public void init() {
        displayStr = "Ha un Doble Click aquí";
    }
    public void paint(Graphics g) {
        Color fondo = new Color(255, 204, 51);
        g.setColor(fondo);
        g.drawRect(0, 0, size().width-1, size().height-1);
        g.setColor(Color.black);
        g.drawString(displayStr, 40, 30);
    }
    public boolean mouseDown(java.awt.Event evt, int x, int y) {
        long clickTime = System.currentTimeMillis();
        long clickInterval = clickTime - firstClickTime;
        if (clickInterval < 200) {
            displayStr = "Doble Click!! (Intervalo = " + clickInterval + ")";
            firstClickTime = 0;
        } else {
            displayStr = "Un solo Click!!";
            firstClickTime = clickTime;
        }
        repaint();
        return true;
    }
}

```

Se podría utilizar el valor devuelto por este método para calcular la hora y fecha actuales. Sin embargo, encontrarás más conveniente obtener la hora y fecha actuales desde la clase [Date](#) del paquete java.util.

Salir del Entorno de Ejecución.

Para salir del intérprete Java, llama al método `System.exit()`. Debes pasarle un código de salida en un entero y el interprete saldrá con ese código de salida:

```
System.exit(-1);
```

Nota: El método `exit()` hace salir del intérprete Java, no sólo del programa Java-- ten cuidado cuando utilice esta función.

Consideraciones de Seguridad: La llamada al método `exit()` está sujeta a las restricciones de seguridad. Por eso dependiendo del navegador donde se esté ejecutando el applet, una llamada a `exit()` desde un applet podría resultar en una excepción `SecurityException`.

Seleccionar y Obtener el Manejador de Seguridad

El controlador de seguridad es un objeto que refuerza cierta vigilancia de seguridad para una aplicación Java. Se puede seleccionar el controlador de seguridad actual para una aplicación utilizando el método `setSecurityManager()` de la clase `System`, y se puede recuperar el controlador de seguridad actual utilizando el método `getSecurityManager()`.

Consideraciones de Seguridad: El controlador de seguridad de una aplicación sólo puede seleccionarse una vez. Normalmente, un navegador selecciona su controlador de seguridad

durante el arranque. Por eso, la mayoría de las veces, los applets no pueden seleccionar el controlador de seguridad porque ya ha sido seleccionado. Si un applet hace esto resultará una `SecurityException`.

Cambios en el JDK 1.1: Utilizar los Recursos del Sistema

[Los Streams de I/O Estándards](#)

La clase System soporta tres nuevos métodos que permiten aumentar los streams de I/O estándares. Puedes ver [Cambios en el JDK 1.1: La clase System](#).

[Propiedades del Sistema](#)

El método getenv ha sido eliminado. Puedes ver [Cambios en el JDK 1.1: La clase System](#).

[Finalizar Objetos](#)

El nuevo método runFinalizersOnExit, de la clase System permite seleccionar si los finalizadores son llamados cuando tu programa finalice. Puedes ver [Cambios en el JDK 1.1: La clase System](#).

Métodos Misceláneos del Sistema

[El ejemplo de copia de Array](#)

[ArrayCopyTest.java](#) utiliza un constructor de String que ha sido eliminado. Puedes ver [Cambios en el JDK 1.1: Ejemplo de copia de Arrays](#).

[El Applet TimingIsEverything](#)

[TimingIsEverything.java](#) utiliza un API del AWT que está obsoleto. Puedes ver [Cambios en el JDK 1.1: El Applet TimingIsEverything](#).

Manejo de Errores utilizando Excepciones

Existe una regla de oro en el mundo de la programación: en los programas ocurren errores. Esto es sabido. Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja?, ¿Puede recuperarlo el programa?

El lenguaje Java utiliza excepciones para proporcionar capacidades de manejo de errores. En esta lección aprenderás qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de Java.

Ozito

¿Qué es una Excepción y por qué debo tener cuidado?

El término excepción es una forma corta de la frase "suceso excepcional" y puede definirse de la siguiente forma:

Definición: Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites. Cuando dicho error ocurre dentro de un método Java, el método crea un objeto 'exception' y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología Java, crear un objeto exception y manejarlo por el sistema de ejecución se llama lanzar una excepción.

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de "algunos" métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el "manejador de excepción" adecuado. Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción manejada por el manejador. Así la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido captura la excepción.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y consecuentemente y el programa Java también).

Mediante el uso de excepciones para manejar errores, los programas Java tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales:

- [Ventaja 1: Separar el Manejo de Errores del Código "Normal"](#)
- [Ventaja 2: Propagar los Errores sobre la Pila de Llamadas](#)
- [Ventaja 3: Agrupar los Tipos de Errores y la Diferenciación de éstos](#)

Ventaja 1: Separar el Manejo de Errores del Código "Normal"

En la programación tradicional, la detección, el informe y el manejo de errores se convierte en un código muy liado. Por ejemplo, supongamos que tenemos una función que lee un fichero completo dentro de la memoria. En pseudo-código, la función se podría parecer a esto:

```
leerFichero {  
    abrir el fichero;  
    determinar su tamaño;  
    asignar suficiente memoria;  
    leer el fichero a la memoria;  
    cerrar el fichero;  
}
```

A primera vista esta función parece bastante sencilla, pero ignora todos aquellos errores potenciales:

- ¿Qué sucede si no se puede abrir el fichero?
- ¿Qué sucede si no se puede determinar la longitud del fichero?
- ¿Qué sucede si no hay suficiente memoria libre?
- ¿Qué sucede si la lectura falla?
- ¿Qué sucede si no se puede cerrar el fichero?

Para responder a estas cuestiones dentro de la función, tendríamos que añadir mucho código para la detección y el manejo de errores. El aspecto final de la función se parecería esto:

```
codigodeError leerFichero {  
    inicializar codigodeError = 0;  
    abrir el fichero;  
    if (ficheroAbierto) {  
        determinar la longitud del fichero;  
        if (obtenerLongitudDelFichero) {  
            asignar suficiente memoria;  
            if (obtenerSuficienteMemoria) {  
                leer el fichero a memoria;  
                if (falloDeLectura) {  
                    codigodeError = -1;  
                }  
            } else {  
                codigodeError = -2;  
            }  
        } else {  
            codigodeError = -3;  
        }  
        cerrar el fichero;  
        if (ficheroNoCerrado && codigodeError == 0) {
```

```

        codigodeError = -4;
    } else {
        codigodeError = codigodeError and -4;
    }
} else {
    codigodeError = -5;
}
return codigodeError;
}

```

Con la detección de errores, las 7 líneas originales (en **negrita**) se han convertido en 29 líneas de código-- a aumentado casi un 400 %. Lo peor, existe tanta detección y manejo de errores y de retorno que en las 7 líneas originales y el código está totalmente atestado. Y aún peor, el flujo lógico del código también se pierde, haciendo difícil poder decir si el código hace lo correcto (si ¿se cierra el fichero realmente si falla la asignación de memoria?) e incluso es difícil asegurar que el código continúe haciendo las cosas correctas cuando se modifique la función tres meses después de haberla escrito. Muchos programadores "resuelven" este problema ignorándolo-- se informa de los errores cuando el programa no funciona.

Java proporciona una solución elegante al problema del tratamiento de errores: las excepciones. Las excepciones le permiten escribir el flujo principal de su código y tratar los casos excepcionales en otro lugar. Si la función leerFichero utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto:

```

leerFichero {
    try {
        abrir el fichero;
        determinar su tamaño;
        asignar suficiente memoria;
        leer el fichero a la memoria;
        cerrar el fichero;
    } catch (falloAbrirFichero) {
        hacerAlgo;
    } catch (falloDeterminacionTamaño) {
        hacerAlgo;
    } catch (falloAsignaciondeMemoria) {
        hacerAlgo;
    } catch (falloLectura) {
        hacerAlgo;
    } catch (falloCerrarFichero) {
        hacerAlgo;
    }
}

```

Observa que las excepciones no evitan el esfuerzo de hacer el trabajo de detectar, informar y manejar errores. Lo que proporcionan las excepciones es la posibilidad de separar los detalles oscuros de qué hacer cuando ocurre algo fuera de la normal.

Además, el factor de aumento de código de este programa es de un 250% -- comparado con el 400% del ejemplo anterior.

Ventaja 2: Propagar los Errores sobre la Pila de Llamadas

Una segunda ventaja de las excepciones es la posibilidad de propagar el error encontrado sobre la pila de llamadas a métodos. Supongamos que el método leerFichero es el cuarto método en una serie de llamadas a métodos anidadas realizadas por un programa principal: metodo1 llama a metodo2, que llama a metodo3, que finalmente llama a leerFichero.

```
metodo1 {  
    call metodo2;  
}  
metodo2 {  
    call metodo3;  
}  
metodo3 {  
    call leerFichero;  
}
```

Supongamos también que metodo1 es el único método interesado en el error que ocurre dentro de leerFichero. Tradicionalmente las técnicas de notificación del error forzarían a metodo2 y metodo3 a propagar el código de error devuelto por leerFichero sobre la pila de llamadas hasta que el código de error llegue finalmente a metodo1 -- el único método que está interesado en él.

```
metodo1 {  
    codigodeErrorType error;  
    error = call metodo2;  
    if (error)  
        procesodelError;  
    else  
        proceder;  
}  
codigodeErrorType metodo2 {  
    codigodeErrorType error;  
    error = call metodo3;  
    if (error)  
        return error;  
    else  
        proceder;
```

```

}
codigodeErrorType metodo3 {
    codigodeErrorType error;
    error = call leerFichero;
    if (error)
        return error;
    else
        proceder;
}

```

Como se aprendió anteriormente, el sistema de ejecución Java busca hacia atrás en la pila de llamadas para encontrar cualquier método que esté interesado en manejar una excepción particular. Un método Java puede "esquivar" cualquier excepción lanzada dentro de él, por lo tanto permite a los métodos que están por encima de él en la pila de llamadas poder capturarlo. Sólo los métodos interesados en el error deben preocuparse de detectarlo.

```

metodo1 {
    try {
        call metodo2;
    } catch (excepcion) {
        procesodelError;
    }
}
metodo2 throws excepcion {
    call metodo3;
}
metodo3 throws excepcion {
    call leerFichero;
}

```

Sin embargo, como se puede ver desde este pseudo-código, requiere cierto esfuerzo por parte de los métodos centrales. Cualquier excepción chequeada que pueda ser lanzada dentro de un método forma parte del interface de programación público del método y debe ser especificado en la clausula throws del método. Así el método informa a su llamador sobre las excepciones que puede lanzar, para que el llamador pueda decidir concienzuda e inteligentemente qué hacer con esa excepción.

Observa de nuevo la diferencia del factor de aumento de código y el factor de ofuscación entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

Ventaja 3: Agrupar Errores y Diferenciación

Frecuentemente las excepciones se dividen en categorías o grupos. Por ejemplo, podríamos imaginar un grupo de excepciones, cada una de las

cuales representara un tipo de error específico que pudiera ocurrir durante la manipulación de un array: el índice está fuera del rango del tamaño del array, el elemento que se quiere insertar en el array no es del tipo correcto, o el elemento que se está buscando no está en el array. Además, podemos imaginar que algunos métodos querrían manejar todas las excepciones de esa categoria (todas las excepciones de array), y otros métodos podría manejar sólo algunas excepciones específicas (como la excepción de índice no válido).

Como todas las excepciones lanzadas dentro de los programas Java son objetos de primera clase, agrupar o categorizar las excepciones es una salida natural de las clases y las superclases. Las excepciones Java deben ser ejemplares de la clase Throwable, o de cualquier descendiente de ésta. Como de las otras clases Java, se pueden crear subclases de la clase Throwable y subclases de estas subclases. Cada clase 'hoja' (una clase sin subclases) representa un tipo específico de excepción y cada clase 'nodo' (una clase con una o más subclases) representa un grupo de excepciones relacionadas.

InvalidIndexException, ElementTypeException, y NoSuchElementException son todas clases hojas. Cada una representa un tipo específico de error que puede ocurrir cuando se manipula un array. Un método puede capturar una excepción basada en su tipo específico (su clase inmediata o interface). Por ejemplo, un manejador de excepción que sólo controle la excepción de índice no válido, tiene una sentencia catch como esta:

```
catch (InvalidIndexException e) {  
    . . .  
}
```

ArrayException es una clase nodo y representa cualquier error que pueda ocurrir durante la manipulación de un objeto array, incluyendo aquellos errores representados específicamente por una de sus subclases. Un método puede capturar una excepción basada en este grupo o tipo general especificando cualquiera de las superclases de la excepción en la sentencia catch. Por ejemplo, para capturar todas las excepciones de array, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento ArrayException:

```
catch (ArrayException e) {  
    . . .  
}
```

Este manejador podría capturar todas las excepciones de array, incluyendo InvalidIndexException, ElementTypeException, y NoSuchElementException. Se puede descubrir el tipo de excepción preciso que ha ocurrido comprobando el parámetro del manejador e. Incluso podríamos seleccionar un manejador de excepciones que

controlara cualquier excepción con este manejador:

```
catch (Exception e) {  
    . . .  
}
```

Los manejadores de excepciones que son demasiado generales, como el mostrado aquí, pueden hacer que el código sea propenso a errores mediante la captura y manejo de excepciones que no se hubieran anticipado y por lo tanto no son manejadas correctamente dentro de manejador. Como regla no se recomienda escribir manejadores de excepciones generales.

Como has visto, se pueden crear grupos de excepciones y manejarlas de una forma general, o se puede especificar un tipo de excepción específico para diferenciar excepciones y manejarlas de un modo exacto.

¿ Y ahora qué?

Ahora que has entendido qué son las excepciones y las ventajas de su utilización en los programas Java, es hora de aprender cómo utilizarlas.

Primer Encuentro con las Excepciones Java

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException
must be caught, or it must be declared in throws clause of this method.
    fis = new FileInputStream(filename);
           ^
```

El mensaje de error anterior es uno de los dos mensajes similares que verás si intentas compilar la clase [InputFile](#), porque la clase `InputFile` contiene llamadas a métodos que lanzan excepciones cuando se produce un error. El lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas desde dentro del ámbito de ese método. (Los detalles sobre lo que ocurre los puedes ver en la próxima página [Requerimientos de Java para Capturar o Especificar](#).)

Si el compilador detecta un método, como los de `InputFile`, que no cumplen este requerimiento, muestra un error como el anterior y no compila el programa.

Echemos un vistazo a `InputFile` en más detalle y veamos que sucede.

La clase `InputFile` envuelve un canal `FileInputStream` y proporciona un método, `getLine()`, para leer una línea en la posición actual del canal de entrada.

```
// Nota: Esta clase no se compila por diseño!
import java.io.*;

class InputFile {

    FileInputStream fis;

    InputFile(String filename) {
        fis = new FileInputStream(filename);
    }

    String getLine() {
        int c;
        StringBuffer buf = new StringBuffer();

        do {
            c = fis.read();
            if (c == '\n')                // nueva línea en UNIX
                return buf.toString();
            else if (c == '\r') {         // nueva línea en Windows 95/NT
                c = fis.read();
                if (c == '\n')
                    return buf.toString();
                else {
                    buf.append((char)'\r');
                    buf.append((char)c);
                }
            }
        } else
```

```

        buf.append((char)c);
    } while (c != -1);

    return null;
}
}

```

El compilador dará el primer error en la primera línea que está en **negrita**. Esta línea crea un objeto `FileInputStream` y lo utiliza para abrir un fichero (cuyo nombre se pasa dentro del constructor del `FileInputStream`).

Entonces, ¿Qué debe hacer el `FileInputStream` si el fichero no existe? Bien, eso depende de lo que quiera hacer el programa que utiliza el `FileInputStream`. Los implementadores de `FileInputStream` no tenían ni idea de lo que quiere hacer la clase `InputFile` si no existe el fichero. ¿Debe `FileInputStream` terminar el programa? ¿Debe intentar un nombre alternativo? o ¿debería crear un fichero con el nombre indicado? No existe una forma posible de que los implementadores de `FileInputStream` pudieran elegir una solución que sirviera para todos los usuarios de `FileInputStream`. Por eso ellos lanzaron una excepción. Esto es, si el fichero nombrado en el argumento del constructor de `FileInputStream` no existe, el constructor lanza una excepción `java.io.FileNotFoundException`. Mediante el lanzamiento de esta excepción, `FileInputStream` permite que el método llamador maneje ese error de la forma que considere más apropiada.

Como puedes ver en el listado, la clase `InputFile` ignora completamente el hecho de que el constructor de `FileInputStream` puede lanzar una excepción. Sin embargo, El lenguaje Java requiere que un método o bien lance o especifique todas las excepciones chequeadas que pueden ser lanzadas desde dentro de su ámbito. Como la Clase `InputFile` no hace ninguna de las dos cosas, el compilador rehusa su compilación e imprime el mensaje de error.

Además del primer error mostrado arriba, se podrá ver el siguiente mensaje de error cuando se compile la clase `InputFile`:

```

InputFile.java:15: Warning: Exception java.io.IOException must be caught,
or it must be declared in throws clause of this method.
    while ((c = fis.read()) != -1) {
                ^

```

El método `getLine()` de la clase `InputFile` lee una línea desde el `FileInputStream` que fue abierto por el constructor de `InputFile`. El método `read()` de `FileInputStream` lanza la excepción `java.io.IOException` si por alguna razón no pudiera leer el fichero. De nuevo, la clase `InputFile` no hace ningún intento por capturar o especificar esta excepción lo que se convierte en el segundo mensaje de error.

En este punto, tenemos dos opciones. Se puede capturar las excepciones con los métodos apropiados en la clase `InputFile`, o se puede esquivarlas y permitir que otros métodos anteriores en la pila de llamadas las capturen. De cualquier forma, los métodos de `InputFile` deben hacer algo, o capturar o especificar las excepciones, antes de poder compilar la clase `InputFile`. Aquí tiene la clase [InputFileDeclared](#), que corrige los errores de `InputFile` mediante la especificación de las excepciones.

La siguiente página le describe con más detalles los [Requerimientos de Java para Capturar o Especificar](#). Las páginas siguientes le enseñarán cómo cumplir estos requerimientos.

Ozito

Requerimientos de Java para Capturar o Especificar Excepciones

Como se mencionó anteriormente, Java requiere que un método o capture o especifique todas las excepciones chequeadas que se pueden lanzar dentro de su ámbito. Este requerimiento tiene varios componentes que necesitan una mayor descripción:

Capturar

Un método puede capturar una excepción proporcionando un manejador para ese tipo de excepción. La página siguiente, [Tratar con Excepciones](#), introduce un programa de ejemplo, le explica cómo capturar excepciones, y le muestra cómo escribir un manejador de excepciones para el programa de ejemplo.

Especificar

Si un método decide no capturar una excepción, debe especificar que puede lanzar esa excepción. ¿Por qué hicieron este requerimiento los diseñadores de Java? Porque una excepción que puede ser lanzada por un método es realmente una parte del interface de programación público del método: los llamadores de un método deben conocer las excepciones que ese método puede lanzar para poder decidir inteligente y concienzudamente qué hacer son esas excepciones. Así, en la firma del método debe especificar las excepciones que el método puede lanzar.

La siguiente página, [Tratar con Excepciones](#), le explica la especificación de excepciones que un método puede lanzar y le muestra cómo hacerlo.

Excepciones Chequeadas

Java tiene diferentes tipos de excepciones, incluyendo las excepciones de I/O, las excepciones en tiempo de ejecución, y las de su propia creación. Las que nos interesan a nosotros para esta explicación son las excepciones en tiempo de ejecución. Estas excepciones son aquellas que ocurren dentro del sistema de ejecución de Java. Esto incluye las excepciones aritméticas (como dividir por cero), excepciones de puntero (como intentar acceder a un objeto con una referencia nula), y excepciones de indexación (como intentar acceder a un elemento de un array con un índice que es muy grande o muy pequeño).

Las excepciones en tiempo de ejecución pueden ocurrir en cualquier parte de un programa y en un programa típico pueden ser muy numerosas. Muchas veces, el costo de chequear todas las excepciones

en tiempo de ejecución excede de los beneficios de capturarlas o especificarlas. Así el compilador no requiere que se capturen o especifiquen estas excepciones, pero se puede hacer.

Las excepciones chequeadas son excepciones que no son excepciones en tiempo de ejecución y que son chequeadas por el compilador (esto es, el compilador comprueba que esas excepciones son capturadas o especificadas).

Algunas veces esto se considera como un bucle cerrado en el mecanismo de manejo de excepciones de Java y los programadores se ven tentados a convertir todas las excepciones en excepciones en tiempo de ejecución. En general, esto no está recomendado. [La controversia -- Excepciones en Tiempo de Ejecución](#) contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.

Excepciones que pueden ser lanzadas desde el ámbito de un método

Esta sentencia podría parecer obvia a primera vista: sólo hay que fijarse en la sentencia `throw`. Sin embargo, esta sentencia incluye algo más no sólo las excepciones que pueden ser lanzadas directamente por el método: la clave está en la frase `dentro del ámbito de`. Esta frase incluye cualquier excepción que pueda ser lanzada mientras el flujo de control permanezca dentro del método. Así, esta sentencia incluye:

- excepciones que son lanzadas directamente por el método con la sentencia `throw` de Java, y
- las excepciones que son lanzadas por el método indirectamente a través de llamadas a otros métodos.

Tratar con Excepciones

[Primer Encuentro con las Excepciones de Java](#) describió brevemente cómo fue introducido en las excepciones Java: con un error del compilador indicando que las excepciones deben ser capturadas o especificadas. Luego [Requerimientos de Java para la Captura o Especificación](#) explicó qué significan exactamente los mensajes de error y por qué los diseñadores de Java decidieron hacer estos requerimientos. Ahora vamos a ver cómo capturar una excepción y cómo especificar otra.

El ejemplo: ListOfNumbers

Las secciones posteriores, sobre como capturar y especificar excepciones, utilizan el mismo ejemplo. Este ejemplo define e implementa un clase llamada ListOfNumbers. Esta clase llama a dos clases de los paquetes de Java que pueden lanzar excepciones. [Capturar y Manejar Excepciones](#) mostrará cómo escribir manejadores de excepciones para las dos excepciones, y [Especificar las Excepciones Lanzadas por un Método](#) mostrará cómo especificar esas excepciones en lugar de capturarlas.

Capturar y Manejar Excepciones

Una vez que te has familiarizado con la clase ListOfNumbers y con las excepciones que pueden ser lanzadas, puedes aprender cómo escribir manejadores de excepción que puedan capturar y manejar esas excepciones.

Esta sección cubre los tres componentes de un manejador de excepción -- los bloques try, catch, y finally -- y muestra cómo utilizarlos para escribir un manejador de excepción para el método writeList() de la clase ListOfNumbers. Además, esta sección contiene una página que pasea a lo largo del método writeList() y analiza lo que ocurre dentro del método en varios escenarios.

Especificar las Excepciones que pueden ser Lanzadas por un Método

Si no es apropiado que un método capture y maneje una excepción lanzada por un método que él ha llamado, o si el método lanza su propia excepción, debe especificar en la firma del método que éste puede lanzar una excepción. Utilizando la clase ListOfNumbers, esta sección le muestra cómo especificar las excepciones lanzadas por un método.

El ejemplo: ListOfNumbers

Las dos secciones siguientes que cubren la captura y especificación de excepciones utilizan este ejemplo:

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }

    public void writeList() {
        PrintStream pStr = null;

        System.out.println("Entering try statement");
        int i;
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));

        pStr.close();
    }
}
```

Este ejemplo define e implementa una clase llamada `ListOfNumbers`. Sobre su construcción, esta clase crea un `Vector` que contiene diez elementos enteros con valores secuenciales del 0 al 9. Esta clase también define un método llamado `writeList()` que escribe los números de la lista en un fichero llamado "OutFile.txt".

El método `writeList()` llama a dos métodos que pueden lanzar excepciones. Primero la siguiente línea invoca al constructor de `FileOutputStream`, que lanza una excepción `IOException` si el fichero no puede ser abierto por cualquier razón:

```
pStr = new PrintStream(new BufferedOutputStream(new
FileOutputStream("OutFile.txt")));
```

Segundo, el método `elementAt()` de la clase `Vector` lanza una excepción `ArrayIndexOutOfBoundsException` si se le pasa un índice cuyo valor sea demasiado pequeño (un número negativo) o demasiado grande (mayor que el número de elementos que contiene realmente el `Vector`). Aquí está cómo `ListOfNumbers` invoca a

elementAt():

```
pStr.println("Value at: " + i + " = " + victor.elementAt(i));
```

Si se intenta compilar la clase ListOfNumbers, el compilador dará un mensaje de error sobre la excepción lanzada por el constructor de FileOutputStream, pero no muestra ningún error sobre la excepción lanzada por elementAt().

Esto es porque la excepción lanzada por FileOutputStream, es una excepción chequeada y la lanzada por elementAt() es una ejecución de tiempo de ejecución. Java sólo requiere que se especifiquen o capturen las excepciones chequeadas. Para más información, puedes ver [Requerimientos de Java para Capturar o Especificar](#).

La siguiente sección, [Captura y Manejo de Excepciones](#), le mostrará cómo escribir un manejador de excepción para el método writeList() de ListOfNumbers.

Después de esto, una sección llamada [Especificar las Excepciones Lanzadas por un Método](#), mostrará cómo especificar que el método writeList() lanza excepciones en lugar de capturarlas.

Capturar y Manejar Excepciones

Las siguientes páginas muestran cómo construir un manejador de excepciones para el método `writeList()` descrito en [El ejemplo: ListOfNumbers](#). Las tres primeras páginas listadas abajo describen tres componentes diferentes de un manejador de excepciones y le muestran cómo pueden utilizarse esos componentes en el método `writeList()`. La cuarta página trata sobre el método `writeList()` resultante y analiza lo que ocurre dentro del código de ejemplo a través de varios escenarios.

El Bloque try

El primer paso en la escritura de un manejador de excepciones es poner la sentencia Java dentro de la cual se puede producir la excepción dentro de un bloque `try`. Se dice que el bloque `try` gobierna las sentencias encerradas dentro de él y define el ámbito de cualquier manejador de excepciones (establecido por el bloque `catch` subsecuente) asociado con él.

Los bloques catch

Después se debe asociar un manejador de excepciones con un bloque `try` proporcionándole uno o más bloques `catch` directamente después del bloque `try`.

El bloque finally

El bloque `finally` de Java proporciona un mecanismo que permite a sus métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque `try`. Se utiliza el bloque `finally` para cerrar ficheros o liberar otros recursos del sistema.

Poniéndolo Todo Junto

Las secciones anteriores describen cómo construir los bloques de código `try`, `catch`, y `finally` para el ejemplo `writeList()`. Ahora, pasaremos sobre el código para investigar que sucede en varios escenarios.

El bloque try

El primer paso en la construcción de un manejador de excepciones es encerrar las sentencias que podrían lanzar una excepción dentro de un bloque try. En general, este bloque se parece a esto:

```
try {  
    sentencias Java  
}
```

El segmento de código etiquetado sentencias java está compuesto por una o más sentencias legales de Java que podrían lanzar una excepción.

Para construir un manejador de excepción para el método writeList() de la clase ListOfNumbers, se necesita encerrar la sentencia que lanza la excepción en el método writeList() dentro de un bloque try.

Existe más de una forma de realizar esta tarea. Podríamos poner cada una de las sentencias que potencialmente pudieran lanzar una excepción dentro de su propio bloque try, y proporcionar manejadores de excepciones separados para cada uno de los bloques try. O podríamos poner todas las sentencias de writeList() dentro de un sólo bloque try y asociar varios manejadores con él. El siguiente listado utiliza un sólo bloque try para todo el método porque el código tiende a ser más fácil de leer.

```
PrintStream pstr;  
  
try {  
    int i;  
  
    System.out.println("Entering try statement");  
    pStr = new PrintStream(  
        new BufferedOutputStream(  
            new FileOutputStream("OutFile.txt")));  
  
    for (i = 0; i < size; i++)  
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));  
}
```

Se dice que el bloque try gobierna las sentencias encerradas dentro del él y define el ámbito de cualquier manejador de excepción (establecido por su subsecuente bloque catch) asociado con él. En otras palabras, si ocurre una excepción dentro del bloque try, esta excepción será manejada por el manejador de excepción asociado con esta sentencia try.

Una sentencia try debe ir acompañada de al menos un bloque catch o un bloque finally.

Los bloques catch

Como se aprendió en la [página anterior](#), la sentencia try define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia try proporcionando uno o más bloques catch directamente después del bloque try:

```
try {  
    .  
    .  
    .  
} catch ( . . . ) {  
    .  
    .  
    .  
} catch ( . . . ) {  
    .  
    .  
    .  
} . . .
```

No puede haber ningún código entre el final de la sentencia try y el principio de la primera sentencia catch. La forma general de una sentencia catch en Java es esta:

```
catch (AlgunObjetoThrowable nombreVariable) {  
    Sentencias Java  
}
```

Como puedes ver, la sentencia catch requiere un sólo argumento formal. Este argumento parece un argumento de una declaración de método. El tipo del argumento AlgunObjetoThrowable declara el tipo de excepción que el manejador puede manejar y debe ser el nombre de una clase heredada de la clase [Throwable](#) definida en el paquete [java.lang](#). (Cuando los programas Java lanzan una excepción realmente están lanzando un objeto, sólo pueden lanzarse los objetos derivados de la clase Throwable. Aprenderás cómo lanzar excepciones en la lección [¿Cómo Lanzar Excepciones?.](#))

nombreVariable es el nombre por el que el manejador puede referirse a la excepción capturada. Por ejemplo, los manejadores de excepciones para el método writeList() (mostrados más adelante) llaman al método getMessage() de la excepción utilizando el nombre de excepción declarado e:

```
e.getMessage()
```

Se puede acceder a las variables y métodos de las excepciones en la misma forma que accede a los de cualquier otro objeto. getMessage() es un método proporcionado por la clase Throwable que imprime información adicional sobre el error ocurrido. La clase Throwable también implementa dos métodos para rellenar e imprimir el contenido de la pila de ejecución cuando ocurre la excepción. Las subclases de Throwable pueden añadir otros métodos o variables de ejemplar. Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases antecesoras.

El bloque catch contiene una serie de sentencias Java legales. Estas sentencias se ejecutan cuando se llama al manejador de excepción. El sistema de ejecución llama al manejador de excepción cuando el manejador es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

El método writeList() de la clase de ejemplo ListOfNumbers utiliza dos manejadores de excepción para su sentencia try, con un manejador para cada uno de los tipos de excepciones que pueden lanzarse dentro del bloque try -- ArrayIndexOutOfBoundsException y IOException.

```
try {  
    .  
    .  
    .  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

}

Ocurre una IOException

Supongamos que ocurre una excepción IOException dentro del bloque try. El sistema de ejecución inmediatamente toma posesión e intenta localizar el manejador de excepción adecuado. El sistema de ejecución empieza buscando al principio de la pila de llamadas. Sin embargo, el constructor de FileOutputStream no tiene un manejador de excepción apropiado por eso el sistema de ejecución comprueba el siguiente método en la pila de llamadas -- el método writeList(). Este método tiene dos manejadores de excepciones: uno para ArrayIndexOutOfBoundsException y otro para IOException.

El sistema de ejecución comprueba los manejadores de writeList() por el orden en el que aparecen después del bloque try. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una ArrayIndexOutOfBoundsException, pero la excepción que se ha lanzado era una IOException. Una excepción IOException no puede asignarse legalmente a una ArrayIndexOutOfBoundsException, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de writeList() es una IOException. La excepción lanzada por el constructor de FileOutputStream también es una IOException y por eso puede ser asignada al argumento del manejador de excepciones de IOException. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia:

```
Caught IOException: OutFile.txt
```

El sistema de ejecución sigue un proceso similar si ocurre una excepción ArrayIndexOutOfBoundsException. Para más detalles puedes ver: [Poniéndolo todo Junto](#) que te lleva a través de método writeList() después de haberlo completado (queda un paso más) e investiga lo que sucede en varios escenarios.

Capturar Varios Tipos de Excepciones con Un Manejador

Los dos manejadores de excepción utilizados por el método writeList() son muy especializados. Cada uno sólo maneja un tipo de excepción. El lenguaje Java permite escribir manejadores de excepciones generales que pueden manejar varios tipos de excepciones.

Como ya sabes, las excepciones Java son objetos de la clase Throwable (son ejemplares de la clase Throwable a de alguna de sus subclases). Los paquetes Java contienen numerosas clases derivadas de la clase Throwable y así construyen un árbol de clases Throwable.

El manejador de excepción puede ser escrito para manejar cualquier clase heredada de Throwable. Si se escribe un manejador para una clase 'hoja' (una clase que no tiene subclases), se habrá escrito un manejador especializado: sólo maneja excepciones de un tipo específico. Si se escribe un manejador para una clase 'nodo' (una clase que tiene subclases), se habrá escrito un manejador general: se podrá manejar cualquier excepción cuyo tipo sea el de la clase nodo o de cualquiera de sus subclases.

Módifiquemos de nuevo el método writeList(). Sólo esta vez, escribamoslo para que maneje las dos excepciones IOExceptions y ArrayIndexOutOfBoundsExceptions. El antecesor más cercano de estas dos excepciones es la clase Exception. Así un manejador de excepción que quisiera manejar los dos tipos se parecería a esto:

```
try {  
    . . .  
} catch (Exception e) {  
    System.err.println("Exception caught: " + e.getMessage());  
}
```

La clase `Exception` está bastante arriba en el árbol de herencias de la clase `Throwable`. Por eso, además de capturar los tipos de `IOException` y `ArrayIndexOutOfBoundsException` este manejador de excepciones, puede capturar otros muchos tipos. Generalmente hablando, los manejadores de excepción deben ser más especializados.

Los manejadores que pueden capturar la mayoría o todas las excepciones son menos utilizados para la recuperación de errores porque el manejador tiene que determinar qué tipo de excepción ha ocurrido de todas formas (para determinar la mejor estrategia de recuperación). Los manejadores de excepciones que son demasiado generales pueden hacer el código más propenso a errores mediante la captura y manejo de excepciones que no fueron anticipadas por el programador y para las que el manejador no está diseñado.

El bloque finally

El paso final en la creación de un manejador de excepción es proporcionar un mecanismo que limpie el estado del método antes (posiblemente) de permitir que el control pase a otra parte diferente del programa. Se puede hacer esto encerrando el código de limpieza dentro de un bloque finally.

El bloque try del método writeList() ha estado trabajando con un PrintStream abierto. El programa debería cerrar ese canal antes de permitir que el control salga del método writeList(). Esto plantea un problema complicado, ya que el bloque try del writeList() tiene tres posibles salidas:

1. La sentencia new FileOutputStream falla y lanza una IOException.
2. La sentencia victor.elementAt(i) falla y lanza una ArrayIndexOutOfBoundsException.
3. Todo tiene éxito y el bloque try sale normalmente.

El sistema de ejecución siempre ejecuta las sentencias que hay dentro del bloque finally sin importar lo que suceda dentro del bloque try. Esto es, sin importar la forma de salida del bloque try del método writeList() debido a los escenarios 1, 2 ó 3 listados arriba, el código que hay dentro del bloque finally será ejecutado de todas formas.

Este es el bloque finally para el método writeList(). Limpia y cierra el canal PrintStream.

```
finally {
    if (pStr != null) {
        System.out.println("Closing PrintStream");
        pStr.close();
    } else {
        System.out.println("PrintStream not open");
    }
}
```

¿Es realmente necesaria la sentencia finally?

La primera necesidad de la sentencia finally podría no aparecer de forma inmediata. Los programadores se preguntan frecuentemente "¿Es realmente necesaria la sentencia finally o es sólo azúcar para mi Java?" En particular los programadores de C++ dudan de la necesidad de esta sentencia porque C++ no la tiene.

Esta necesidad de la sentencia finally no aparece hasta que se considera lo siguiente: ¿cómo se podría cerrar el PrintStream en el método writeList() si no se proporcionara un manejador de excepción para la ArrayIndexOutOfBoundsException y ocurre una ArrayIndexOutOfBoundsException? (sería sencillo y legal omitir un manejador de excepción para ArrayIndexOutOfBoundsException porque es una excepción en tiempo de ejecución y el compilador no alerta de que writeList() contiene una llamada a un método que puede lanzar una).

La respuesta es que el PrintStream no se cerraría si ocurriera una excepción ArrayIndexOutOfBoundsException y writeList() no proporcionara un manejador para ella -- a menos que writeList() proporcionara una sentencia finally.

Existen otros beneficios de la utilización de la sentencia finally. En el ejemplo de writeList() es posible proporcionar un código de limpieza sin la intervención de una sentencia finally. Por ejemplo, podríamos poner el código para cerrar el PrintStream al final del bloque try y de nuevo dentro del manejador de excepción para ArrayIndexOutOfBoundsException, como se muestra aquí:

```
try {
    . . .
    pStr.close();           // No haga esto, duplica el código
} catch (ArrayIndexOutOfBoundsException e) {
    pStr.close();           // No haga esto, duplica el código
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
}
```



```
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

Sin embargo, esto duplica el código, haciéndolo difícil de leer y propenso a errores si se modifica más tarde. Por ejemplo, si se añade código al bloque try que pudiera lanzar otro tipo de excepción, se tendría que recordar el cerrar el PrintStream dentro del nuevo manejador de excepción (lo que se olvidará seguro si se parece a mí).

Poniéndolo Todo Junto

Cuando se juntan todos los componentes, el método `writeList()` se parece a esto:

```
public void writeList() {
    PrintStream pStr = null;

    try {
        int i;

        System.out.println("Entrando en la Sentencia try");
        pStr = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: " +
e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (pStr != null) {
            System.out.println("Cerrando PrintStream");
            pStr.close();
        } else {
            System.out.println("PrintStream no está abierto");
        }
    }
}
```

El bloque `try` de este método tiene tres posibilidades de salida diferentes:

1. La sentencia `new FileOutputStream` falla y lanza una `IOException`.
2. La sentencia `victor.elementAt(i)` falla y lanza una `ArrayIndexOutOfBoundsException`.
3. Todo tiene éxito y la sentencia `try` sale normalmente.

Esta página investiga en detalle lo que sucede en el método `writeList` durante cada una de esas posibilidades de salida.

Escenario 1:Ocurre una excepción `IOException`

La sentencia `new FileOutputStream("OutFile.txt")` puede fallar por varias razones: el usuario no tiene permiso de escritura sobre el fichero o el directorio, el sistema de ficheros está lleno, o no existe el directorio. Si cualquiera de estas situaciones es verdadera el constructor de `FileOutputStream` lanza una excepción `IOException`.

Cuando se lanza una excepción `IOException`, el sistema de ejecución para inmediatamente la ejecución del bloque `try`. Y luego intenta localizar un manejador de excepción apropiado para manejar una `IOException`.

El sistema de ejecución comienza su búsqueda al principio de la pila de llamadas. Cuando ocurrió la excepción, el constructor de `FileOutputStream` estaba al principio de la pila de llamadas. Sin embargo, este constructor no tiene un manejador de excepción apropiado por lo que el sistema comprueba el siguiente método que hay en la pila de llamadas -- el método `writeList()`. Este método tiene dos manejadores de excepciones: uno para `ArrayIndexOutOfBoundsException` y otro para `IOException`.

El sistema de ejecución comprueba los manejadores de `writeList()` por el orden en el que aparecen después del bloque `try`. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una `ArrayIndexOutOfBoundsException`, pero la excepción que se ha lanzado era una `IOException`. Una excepción `IOException` no puede asignarse legalmente a una `ArrayIndexOutOfBoundsException`, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de `writeList()` es una `IOException`. La excepción lanzada por el constructor de `FileOutputStream` también es una `IOException` y por eso puede ser asignada al argumento del manejador de excepciones de `IOException`. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia:

```
Caught IOException: OutFile.txt
```

Después de que se haya ejecutado el manejador de excepción, el sistema pasa el control al bloque `finally`. En este escenario particular, el canal `PrintStream` nunca se ha abierto, así el `pStr` es `null` y no se cierra. Después de que se haya completado la ejecución del bloque `finally`, el programa continúa con la primera sentencia después de este bloque.

La salida completa que se podrá ver desde el programa `ListOfNumbers` cuando se lanza una excepción `IOException` es esta:

```
Entrando en la sentecia try
Caught IOException: OutFile.txt
PrintStream no está abierto
```

Escenario 2: Ocurre una excepción `ArrayIndexOutOfBoundsException`

Este escenario es el mismo que el primero excepto que ocurre un error diferente dentro del bloque `try`. En este escenario, el argumento pasado al método `elementAt()` de `Vector` está fuera de límites. Esto es, el argumento es menor que cero o mayor que el tamaño del array. (De la forma en que

está escrito el código, esto es realmente imposible, pero supongamos que se ha introducido un error cuando alguien lo ha modificado).

Como en el escenario 1, cuando ocurre una excepción el sistema de ejecución para la ejecución del bloque try e intenta localizar un manejador de excepción apropiado para `ArrayIndexOutOfBoundsException`. El sistema busca como lo hizo anteriormente. Llega a la sentencia catch en el método `writeList()` que maneja excepciones del tipo `ArrayIndexOutOfBoundsException`. Como el tipo de la excepción corresponde con el de el manejador, el sistema ejecuta el manejador de excepción.

Después de haber ejecutado el manejador de excepción, el sistema pasa el control al bloque finally. En este escenario particular, el canal `PrintStream` si que se ha abierto, así que el bloque finally lo cerrará. Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida completa que dará el programa `ListOfNumbers` si ocurre una excepción `ArrayIndexOutOfBoundsException`:

```
Entrando en la sentencia try  
Caught ArrayIndexOutOfBoundsException: 10 >= 10  
Cerrando PrintStream
```

Escenario 3: El bloque try sale normalmente

En este escenario, todas las sentencias dentro del ámbito del bloque try se ejecutan de forma satisfactoria y no lanzan excepciones. La ejecución cae al final del bloque try y el sistema pasa el control al bloque finally. Como todo ha salido satisfactorio, el `PrintStream` abierto se cierra cuando el bloque finally consigue el control. De nuevo, Después de que el bloque finally haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida cuando el programa `ListOfNumbers` cuando no se lanzan excepciones:

```
Entrando en la sentencia try  
Cerrando PrintStream
```

Especificar las Excepciones Lanzadas por un Método

Le sección anterior mostraba como escribir un manejador de excepción para el método `writeList()` de la clase `ListOfNumbers`. Algunas veces, es apropiado capturar las excepciones que ocurren pero en otras ocasiones, sin embargo, es mejor dejar que un método superior en la pila de llamadas maneje la excepción. Por ejemplo, si se está utilizando la clase `ListOfNumbers` como parte de un paquete de clases, probablemente no se querrá anticipar las necesidades de todos los usuarios de su paquete. En este caso, es mejor no capturar las excepciones y permitir que alguien la capture más arriba en la pila de llamadas.

Si el método `writeList()` no captura las excepciones que pueden ocurrir dentro de él, debe especificar que puede lanzar excepciones. Modifiquemos el método `writeList()` para especificar que puede lanzar excepciones. Como recordatorio, aquí tienes la versión original del método `writeList()`:

```
public void writeList() {
    System.out.println("Entrando en la sentencia try");
    int i;
    pStr = new PrintStream(
        new BufferedOutputStream(
            new FileOutputStream("OutFile.txt")));

    for (i = 0; i < size; i++)
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

Como recordarás, la sentencia `new FileOutputStream("OutFile.txt")` podría lanzar un excepción `IOException` (que no es una excepción en tiempo de ejecución). La sentencia `victor.elementAt(i)` puede lanzar una excepción `ArrayIndexOutOfBoundsException` (que es una subclase de la clase `RuntimeException`, y es una excepción en tiempo de ejecución).

Para especificar que `writeList()` lanza estas dos excepciones, se añade la clausula `throws` a la firma del método de `writeList()`. La clausula `throws` está compuesta por la palabra clave `throws` seguida por una lista separada por comas de todas las excepciones lanzadas por el método. Esta clausula va después del nombre del método y antes de la llave abierta que define el ámbito del método. Aquí tienes un ejemplo:

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

Recuerda que la excepción `ArrayIndexOutOfBoundsException` es una excepción en tiempo de ejecución, por eso no tiene porque especificarse en la sentencia `throws` pero puede hacerse si se quiere.

La Sentencia throw

Todos los métodos Java utilizan la sentencia throw para lanzar una excepción. Esta sentencia requiere un sólo argumento, un objeto Throwable. En el sistema Java, los objetos lanzables son ejemplares de la clase [Throwable](#) definida en el paquete [java.lang](#). Aquí tienes un ejemplo de la sentencia throw:

```
throw algunObjetoThrowable;
```

Si se intenta lanzar un objeto que no es 'lanzable', el compilador rehusa la compilación del programa y muestra un mensaje de error similar a éste:

```
testing.java:10: Cannot throw class java.lang.Integer; it must be a subclass
of class java.lang.Throwable.
        throw new Integer(4);
        ^
```

La página siguiente, [La clase Throwable y sus Subclases](#), cuentan más cosas sobre la clase Throwable.

Echemos un vistazo a la sentencia throw en su contexto. El siguiente método está tomado de una clase que implementa un objeto pila normal. El método pop() saca el elemento superior de la pila y lo devuelve:

```
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

El método pop() comprueba si hay algún elemento en la pila. Si la pila está vacía (su tamaño es igual a cero), ejemplariza un nuevo objeto de la clase EmptyStackException y lo lanza. Esta clase está definida en el paquete java.util. En páginas posteriores podrás ver cómo crear tus propias clases de excepciones. Por ahora, todo lo que necesitas recordar es que se pueden lanzar objetos heredados desde la clase Throwable.

La clausula throws

Habrás observado que la declaración del método pop() contiene esta clausula:

```
throws EmptyStackException
```

La clausula throws especifica que el método puede lanzar una excepción EmptyStackException. Como ya sabes, el lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito. Se puede hacer esto con la clausula throws de la declaración del método. Para más información sobre estos requerimientos puedes ver [Requerimientos Java para Capturar o Especificar](#).

También puedes ver, [Especificar las Excepciones lanzadas por un Método](#) para

obtener más detalles sobre cómo un método puede lanzar excepciones.

Ozito

La clase Throwable y sus Subclases

Como se aprendió en la página anterior, sólo se pueden lanzar objetos que estén derivados de la clase Throwable. Esto incluye descendientes directos (esto es, objetos de la clase Throwable) y descendiente indirectos (objetos derivados de hijos o nietos de la clase Throwable).

Este diagrama ilustra el árbol de herencia de la clase Throwable y sus subclases más importantes:



Como se puede ver en el diagrama, la clase Throwable tiene dos descendientes directos: Error y Exception.

Error

Cuando falla un enlace dinámico, y hay algún fallo "hardware" en la máquina virtual, ésta lanza un error. Típicamente los programas Java no capturan los Errores. Pero siempre lanzarán errores.

Exception

La mayoría de los programas lanzan y capturan objetos derivados de la clase `Exception`. Una `Excepción` indica que ha ocurrido un problema pero que el problema no es demasiado serio. La mayoría de los programas que escribirás lanzarán y capturarán excepciones.

La clase `Exception` tiene muchos descendiente definidos en los paquetes Java. Estos descendientes indican varios tipos de excepciones que pueden ocurrir. Por ejemplo, `IllegalAccessException` señala que no se

puede encontrar un método particular, y `NegativeArraySizeException` indica que un programa intenta crear un array con tamaño negativo.

Una subclase de `Exception` tiene un significado especial en el lenguaje Java: `RuntimeException`.

Excepciones en Tiempo de Ejecución

La clase `RuntimeException` representa las excepciones que ocurren dentro de la máquina virtual Java (durante el tiempo de ejecución). Un ejemplo de estas excepciones es `NullPointerException`, que ocurre cuando un método intenta acceder a un miembro de un objeto a través de una referencia nula. Esta excepción puede ocurrir en cualquier lugar en que un programa intente desreferenciar una referencia a un objeto. Frecuentemente el coste de chequear estas excepciones sobrepasa los beneficios de capturarlas.

Como las excepciones en tiempo de ejecución están omnipresentes e intentar capturar o especificarlas todas en todo momento podría ser un ejercicio infructuoso (y un código infructuoso, imposible de leer y de mantener), el compilador permite que estas excepciones no se capturen ni se especifiquen.

Los paquetes Java definen varias clases `RuntimeException`. Se pueden capturar estas excepciones al igual que las otras. Sin embargo, no se requiere que un método especifique que lanza excepciones en tiempo de ejecución. Además puedes crear sus propias subclases de `RuntimeException`. [Excepciones en Tiempo de Ejecución -- La Controversia](#) contiene una explicación detallada sobre cómo utilizar las excepciones en tiempo de ejecución.

Crear Clases de Excepciones

Cuando diseñes un paquete de clases java que colabore para proporcionar alguna función útil a sus usuarios, deberás trabajar duro para asegurarte de que las clases interactúan correctamente y que sus interfaces son fáciles de entender y utilizar. Deberías estar mucho tiempo pensando sobre ello y diseñar las excepciones que esas clases pueden lanzar.

Supon que estás escribiendo una clase con una lista enlazada que estás pensando en distribuir como freeware. Entre otros métodos la clase debería soportar estos:

`objectAt(int n)`

Devuelve el objeto en la posición `n` de la lista.

`firstObject()`

Devuelve el primer objeto de la lista.

`indexOf(Object o)`

Busca el Objeto especificado en la lista y devuelve su posición en ella.

¿Qué puede ir mal?

Como muchos programadores utilizarán tu clase de lista enlazada, puedes estar seguro de que muchos de ellos la utilizarán mal o abusarán de los métodos de la clase. También, alguna llamada legítima a los métodos de la clase podría dar algún resultado indefinido. No importa, con respecto a los errores, querrá que tu clase sea lo más robusta posible, para hacer algo razonable con los errores, y comunicar los errores al programa llamador. Sin embargo, no puedes anticipar como quiere cada usuario de tus clases enlazadas que se comporten sus objetos ante la adversidad. Por eso, lo mejor que puedes hacer cuando ocurre un error es lanzar una excepción.

Cada uno de los métodos soportados por la lista enlazada podría lanzar una excepción bajo ciertas condiciones, y cada uno podría lanzar un tipo diferente de excepción. Por ejemplo:

`objectAt()`

lanzará una excepción si se pasa un entero al método que sea menor que 0 o mayor que el número de objetos que hay realmente en la lista.

`firstObject()`

lanzará una excepción si la lista no contiene objetos.

`indexOf()`

lanzará una excepción si el objeto pasado al método no está en la lista.

Pero ¿qué tipo de excepción debería lanzar cada método? ¿Debería ser una excepción proporcionada por el entorno de desarrollo de Java? O ¿Deberían ser excepciones propias?

Elegir el Tipo de Excepción Lanzada

Tratándose de la elección del tipo de excepción a lanzar, tienes dos opciones:

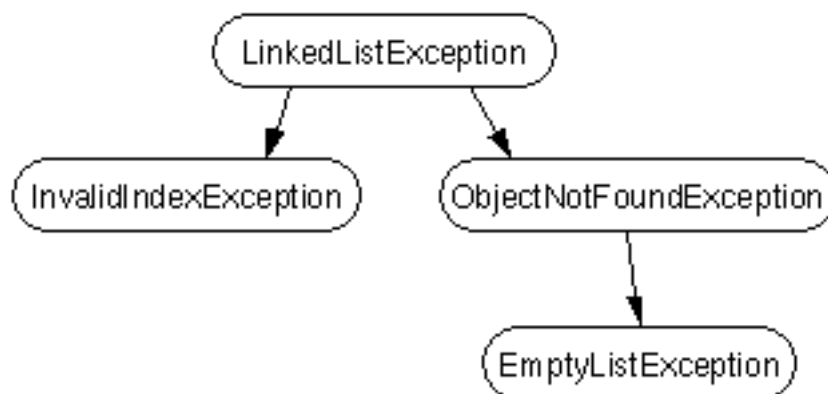
1. Utilizar una escrita por otra persona. Por ejemplo, el entorno de desarrollo de Java proporciona muchas clases de excepciones que podrías utilizar.
2. Escribirlas tu mismo.

Necesitarás escribir tus propias clases de excepciones si respondes "Si" a alguna de las siguientes preguntas. Si no es así, probablemente podrás utilizar alguna excepción ya escrita:

- ¿Necesitas un tipo de excepción que no está representada por lo existentes en el entorno de desarrollo de Java?
- ¿Ayudaría a sus usuarios si pudieran diferenciar sus excepciones de las otras lanzadas por clases escritas por otros vendedores?
- ¿Lanza el código más una excepción relacionada?
- Si utilizas excepciones de otros, ¿Podrán sus usuarios tener acceso a estas excepciones? Una pregunta similar es "¿Debería tu paquete ser independiente y auto-contenedor?"

La clase de lista enlazada puede lanzar varias excepciones, y sería conveniente poder capturar todas las excepciones lanzadas por la lista enlazada con un manejador. Si planeas distribuir la lista enlazada en un paquete, todo el código relacionado debe empaquetarse junto. Así para la lista enlazada, deberías crear tu propio árbol de clases de excepciones.

El siguiente diagrama ilustra una posibilidad del árbol de clases para su lista enlazada:



LinkedListException es la clase padre de todas las posibles excepciones

que pueden ser lanzadas por la clase de la lista enlazada, Los usuarios de esta clase pueden escribir un sólo manejador de excepciones para manejarlas todas con una sentencia catch como esta:

```
catch (LinkedListException) {  
    . . .  
}
```

O, podrñia escribir manejadores más especializados para cada una de las subclases de LinkedListException.

Elegir una Superclase

El diagrama anterior no indica la superclase de la clase LinkedListException. Como ya sabes, las excepciones de Java deben ser ejemplares de la clase Throwable o de sus subclases. Por eso podría tentarte hacer LinkedListException como una subclase de la clase Throwable. Sin embargo, el paquete java.lang proporciona dos clases Throwable que dividen los tipos de problemas que pueden ocurrir en un programa java: Errores y Excepción. La mayoría de los applets y de las aplicaciones que escribes lanzan objetos que son Excepciones. (Los errores están reservados para problemas más serios que pueden ocurrir en el sistema.)

Teóricamente, cualquier subclase de Exception podría ser utilizada como padre de la clase LinkedListException. Sin embargo, un rápido examen de esas clases muestra que o son demasiado especializadas o no están relacionadas con LinkedListException para ser apropiadas. Asi que el padre de la clase LinkedListException debería ser Exception.

Como las excepciones en tiempo de ejecución no tienen por qué ser especificadas en la clausula throws de un método, muchos desarrolladores de paquetes se preguntan: "¿No es más sencillo hacer que todas mis excepciones sean heredadas de Runtime Exception?" La respuesta a esta pregunta con más detalle en [Excepciones en Tiempo de Ejecución -- La Controversia](#). La línea inferior dice que no deberías utilizar subclases de RuntimeException en tus clases a menos que tus excepciones sean realmente en tiempo de ejecución! Para la mayoría de nosotros, esto significa "NO, tus excepciones no deben descender de la clase RuntimeException."

Convenciones de Nombres

Es una buena práctica añadir la palabra "Exception" al final del nombre de todas las clases heredadas (directa o indirectamente) de la clase Exception. De forma similar, los nombres de las clases que se hereden desde la clase Error deberían terminar con la palabra "Error".

Excepciones en Tiempo de Ejecución -- La Controversia

Como el lenguaje Java no requiere que los métodos capturen o especifiquen las excepciones en tiempo de ejecución, es una tentación para los programadores escribir código que lance sólo excepciones de tiempo de ejecución o hacer que todas sus subclasses de excepciones hereden de la clase `RuntimeException`. Estos atajos de programación permiten a los programadores escribir código Java sin preocuparse por los consiguientes:

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught,
or it must be declared in throws clause of this method.
    fis = new FileInputStream(filename);
           ^
```

errores del compilador y sin preocuparse por especificar o capturar ninguna excepción. Mientras esta forma parece conveniente para los programadores, esquiva los requerimientos de Java de capturar o especificar y pueden causar problemas a los programadores que utilicen tus clases.

¿Por qué decidieron los diseñadores de Java forzar a un método a especificar todas las excepciones chequeadas no capturadas que pueden ser lanzadas dentro de su ámbito? Como cualquier excepción que pueda ser lanzada por un método es realmente una parte del interface de programación público del método: los llamadores de un método deben conocer las excepciones que el método puede lanzar para poder decidir concienzuda e inteligentemente qué hacer con estas excepciones. Las excepciones que un método puede lanzar son como una parte del interface de programación del método como sus parámetros y devuelven un valor.

La siguiente pregunta podría ser: " Bien ¿Si es bueno de documentar el API de un método incluyendo las excepciones que pueda lanzar, por qué no especificar también las excepciones de tiempo de ejecución?".

Las excepciones de tiempo de ejecución representan problemas que son detectados por el sistema de ejecución. Esto incluye excepciones aritméticas (como la división por cero), excepciones de punteros (como intentar acceder a un objeto con un referencia nula), y las excepciones de indexación (como intentar acceder a un elemento de un array a través de un índice demasiado grande o demasiado pequeño).

Las excepciones de tiempo de ejecución pueden ocurrir en cualquier lugar del programa y en un programa típico pueden ser muy numerosas. Típicamente, el coste del chequeo de las excepciones de tiempo de ejecución excede de los beneficios de capturarlas o especificarlas. Así el compilador no requiere que se capturen o especifiquen las excepciones de tiempo de ejecución, pero se puede hacer.

Las excepciones chequeadas representan información útil sobre la operación legalmente especificada sobre la que el llamador podría no tener control y el llamador necesita estar informado sobre ella -- por ejemplo, el sistema de ficheros está lleno, o el ordenador remoto ha cerrado la conexión, o los permisos de acceso no permiten esta acción.

¿Qué se consigue si se lanza una excepción `RuntimeException` o se crea una subclase de `RuntimeException` sólo porque no se quiere especificarla? Simplemente, se obtiene la posibilidad de lanzar una excepción sin especificar lo que se está haciendo. En otras palabras, es una forma de evitar la documentación de las excepciones que puede lanzar un método. ¿Cuándo es bueno esto? Bien, ¿cuándo es bueno evitar la documentación sobre el comportamiento de los métodos? La respuesta es "NUNCA".

Reglas del Pulgar:

- Se puede detectar y lanzar una excepción de tiempo de ejecución cuando se encuentra un error en la máquina virtual, sin embargo, es más sencillo dejar que la máquina virtual lo detecte y lo lance. Normalmente, los métodos que escribas lanzarán excepciones del tipo

Exception, no del tipo RuntimeException.

- De forma similar, puedes crear una subclase de RuntimeException cuando estas creando un error en la máquina virtual (que probablemente no lo hará), De otro modo utilizará la clase Exception.
 - No lances una excepción en tiempo de ejecución o crees una subclase de RuntimeException simplemente porque no quieres preocuparte de especificarla.
-

Cambios del API que Afectan al Manejo de Errores Utilizando Excepciones

[Manejo de Errores Utilizando Excepciones](#)

Aunque el mecanismo básico del manejo de excepciones no ha cambiado en el JDK 1.1, se han añadido muchas nuevas clases de excepciones y errores. Comprueba la [Documentación online del API](#).

[El ejemplo ListOfNumbers](#)

El ejemplo ListOfNumbers utiliza métodos desfasados. Puedes ver [Cambios en el JDK 1.1: El ejemplo ListOfNumbers](#)

[La clase Throwable y sus subclases](#)

El JDK 1.1 requiere algún cambio menor en la clase Throwable debido a la internacionalización. Puedes ver: [Cambios en el JDK 1.1: la Clase Throwable](#)

Threads de Control

Abajo tienes tres copias de un applet que anima diferentes algoritmos de ordenación. No, esta lección no trata de los algoritmos de ordenación. Pero estos applets proporcionan una forma visual de entender una poderosa capacidad del lenguaje Java -- los threads.

Burbuja Doble Burbuja Rápido

Ahora arranca cada uno de estos applets, uno por uno, pulsando con el ratón dentro de ellos. ¿Observas algo? Si! Los applets se están ejecutando a la vez. ¿Observas algo más? Si! Puedes moverte por esta página o traer uno de los paneles del navegador mientras estos tres applets están ordenando sus datos. Todo esto se debe al poder de los 'hilos' threads.

[Ozito](#)

¿Qué es un Thread?

Todos los programadores están familiarizados con la escritura de programas secuenciales. Tú probablemente hayas escrito un programa que muestre "Hello World!", o que ordene una lista de nombres, o que calcule la lista de números primos. Estos son programas secuenciales: cada uno tiene un principio, una secuencia de ejecución y un final. En un momento dado durante la ejecución del programa hay un sólo punto de ejecución.

Un Thread es similar a los programas secuenciales descritos arriba: un sólo thread también tiene un principio, un final, una secuencia, y en un momento dado durante el tiempo de ejecución del thread sólo hay un punto de ejecución. Sin embargo, un thread por si mismo no es un programa. No puede ejecutarse por sí mismo, pero si con un programa.

Definición: Un thread es un flujo secuencial de control dentro de un programa.

No hay nada nuevo en el concepto de un sólo thread. Pero el juego real alrededor de los threads no está sobre los threads secuenciales solitarios, sino sobre la posibilidad de que un solo programa ejecute varios threads a la vez y que realicen diferentes tareas.

El navegador HotJava es un ejemplo de una aplicación multi-thread. Dentro del navegador HotJava puedes moverte por la página mientras bajas un applet o una imagen, se ejecuta una animación o escuchas un sonido, imprimes la página en segundo plano mientras descargas una nueva página, o ves cómo los tres algoritmos de ordenación alcanzan la meta.

Algunos textos utilizan el nombre proceso de poco peso en lugar de thread. Un thread es similar a un proceso real en el que un thread y un programa en ejecución son un sólo flujo secuencial de control. Sin embargo, un thread se considera un proceso de poco peso porque se ejecuta dentro del contexto de un programa completo y se aprovecha de los recursos asignados por ese programa y del entorno de éste.

Como un flujo secuencial de control, un thread debe conseguir algunos de sus propios recursos dentro de un programa en ejecución. (Debe tener su propia pila de ejecución y contador de programa, por ejemplo). El código que se ejecuta dentro de un Thread trabaja sólo en éste contexto. Así, algunos texto utilizan el término contexto de ejecución como un sinónimo para los threads.

Un sencillo Thread de Ejemplo

Este ejemplo define dos clases: SimpleThread y TwoThreadsTest. Empecemos nuestra exploración de la aplicación con la clase SimpleThread -- una subclase de la clase Thread, que es proporcionada por el paquete java.lang:

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("HECHO! " + getName());
    }
}
```

El primer método de esta clase es un constructor que toma una cadena como su único argumento. Este constructor está implementado mediante una llamada al constructor de la superclase y es interesante para nosotros sólo porque selecciona el nombre del Thread, que se usará más adelante en el programa.

El siguiente método es el método run(). Este método es el corazón de cualquier Thread y donde tiene lugar la acción del Thread. El método run() de la clase SimpleThread contiene un bucle for que itera diez veces. En cada iteración el método muestra el número de iteración y el nombre del Thread, luego espera durante un intervalo aleatorio de hasta 1 segundo. Después de haber terminado el bucle, el método run() imprime "HECHO!" con el nombre del Thread.

La clase TwoThreads proporciona un método main() que crea dos threads SimpleThread: uno llamado "Jamaica" y otro llamado "Fiji". (Si no quieres decidir donde ir de vacaciones puedes utilizar este programa para ayudarte a elegir -- ve a la isla cuyo thread imprima "HECHO!" primero).

```
class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

El método main() también arranca cada uno de los threads inmediatamente después siguiendo su construcción con una llamada al método start(). El programa daría una salida parecida a esta:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
HECHO! Fiji
9 Jamaica
HECHO! Jamaica
```

Observa cómo la salida de cada uno de los threads se mezcla con la salida del otro. Esto es porque los dos threads SimpleThread se están ejecutando de forma concurrente. Así, los dos métodos run() se stán ejecutando al mismo tiempo y cada thread está mostrándo su salida al mismo tiempo que el otro.

Prueba esto: Modifica el programa principal y crea un tercer Thread llamado "Bora Bora". Compila el programa y ejecútalo de nuevo. ¿Ha cambiado la isla de destino de sus vacaciones?

Ozito

Atributos de un Thread

Por ahora, te has familiarizado con los threads y has visto un sencillo programa Java que ejecuta dos thread concurrentemente. Esta página presenta varias características específicas de los threads Java y proporciona enlaces a las páginas que explican cada característica con más detalle.

Los threads java están implementados por la clase Thread, que es una parte del paquete java.lang. Esta clase implementa una definición de threads independiente del sistema. Pero bajo la campana, la implementación real de la operación concurrente la proporciona una implementación específica del sistema. Para la mayoría de las aplicaciones, la implementación básica no importa. Se puede ignorar la implementación básica y programar el API de los thread descrito en estas lecciones y en otra documentación proporcionada con el sistema Java.

[Cuerpo del Thread](#)

Toda la acción tiene lugar en el cuerpo del thread -- el método run(). Se puede proporcionar el cuerpo de un Thread de una de estas dos formas: subclasificando la clase Thread y sobrescribiendo su método run(), o creando un thread con un objeto de la clase Runnable y su target.

[Estado de un Thread](#)

A lo largo de su vida, un thread tiene uno o varios estados. El estado de un thread indica qué está haciendo el Thread y lo que es capaz de hacer durante su tiempo de vida: ¿se está ejecutando?, ¿está esperando? ¿o está muerto?

[La prioridad de un Thread](#)

Una prioridad del Thread le dice al temporizador de threads de Java cuando se debe ejecutar este thread en relación con los otros.

[Threads Daemon](#)

Estos threads son aquellos que proporcionan un servicio para otros threads del sistema. Cualquier thread Java puede ser un thread daemon.

[Grupos de Threads](#)

Todos los threads pertenecen a un grupo. La clase ThreadGroup, perteneciente al paquete java.lang define e implementa las capacidades de un grupo de thread relacionados.

El Cuerpo de un Thread

Toda la acción tiene lugar en el cuerpo del thread, que es el método `run()` del thread. Después de crear e inicializar un thread, el sistema de ejecución llama a su método `run()`. El código de este método implementa el comportamiento para el que fue creado el thread. Es la razón de existir del thread.

Frecuentemente, el método `run()` de un thread es un bucle. Por ejemplo, un thread de animación podría iterar a través de un bucle y mostrar una serie de imágenes. Algunas veces un método `run()` realiza una operación que tarda mucho tiempo, como descargar y ejecutar un sonido o una película JPEG.

Puedes elegir una de estas dos formas para proporcionar un método `run()` a un thread Java:

1. Crear una subclase de la clase `Thread` definida en el paquete `java.lang` y sobrescribir el método `run()`.
Ejemplo: La clase `SimpleThread` descrita en [En ejemplo sencillo de Thread](#).
2. Proporcionar una clase que implemente el interface `Runnable`, también definido en el paquete `java.lang`. Ahora, cuando se ejemplarice un thread (bien directamente desde la clase `Thread` o desde una de sus subclases), dale al nuevo thread un manejador a un ejemplar de la clase `Runnable`. Este objeto `Runnable` proporciona el método `run()` para el thread.
Ejemplo: El applet de un reloj que verás en la página siguiente.

Existen varias buenas razones para elegir uno de estas dos opciones. Sin embargo, para la mayoría de los casos, la mejor opción será seguir esta regla del pulgar:

Regla del Pulgar: si tu clase debe ser una subclase de otra clase (el ejemplo más común son los applets), deberás utilizar `Runnable` descrito en la sección N° 2.

Un Applet de un Reloj Digital

Este applet muestra la hora actual y la actualiza cada segundo. Puedes moverte por la página o realizar cualquier otra tarea mientras el reloj continua actualizandose porque el código que actualiza el reloj y lo muestra se ejecuta dentro de un thread.

Esta sección explica el [código fuente](#) de este applet. En particular, esta página describe los segmentos de código que implementa el comportamiento del thread del reloj; no describe el código que está relacionado con el ciclo de vida del applet. Si no has escrito sus propios applets o si no estás familiarizado con el ciclo de vida de una applet, podrías encontrar las respuestas en [El ciclo de vida de un Applet](#) antes de proceder con esta página.

Decidir Utilizar el Interface Runnable

El applet del reloj utiliza el interface Runnable para proporcionar el método run() para su thread. Para ejecutarse dentro de un navegador compatible con Java, la clase Clock debe derivar de la clase Applet. Sin embargo este applet también necesita un thread para poder actualizar continuamente la pantalla sin tomar posesión del proceso en el que se está ejecutando. (Algunos navegadores, pero no todos, crean un nuevo thread para cada applet para impedir que un applet descortés tome posesión del thread principal del navegador. Sin embargo, no deberías contar con esto cuando escribas tus applets; los applets deben crear sus propios threads cuando hagan un trabajo de calculo intensivo). Como el lenguaje Java no soporta la herencia múltiple, la clase Class no puede heredarse desde la clase Thread y de la clase Applet a la vez. Por eso, la clase Clock debe utilizar el interface Runnable para proporcionar el comportamiento de sus thread.

Los applets no son threads, ni ningún navegador existente -- compatibles con Java o visualizadores de applets crean threads automáticamente en el que ejecutar applets. Por lo tanto, si un applet, necesita un thread debe creárselo el mismo. El applet del reloj necesita un thread en el que realizar las actualizaciones de pantalla porque se actualiza de forma frecuente y el usuario necesita poder realizar otras tareas a la vez que se ejecuta el reloj (como ir a otra página o moverse por ésta).

El Interface Runnable

El applet del reloj proporciona un método run() para su thread mediante el interface Runnable. La definición de la clase Clock indica que es una subclase de la clase Applet y que implementa el interface Runnable. Si no estás familiarizado con los interfaces puedes revisar la información de la lección "Objetos, Clases, e Interfaces"

```
class Clock extends Applet implements Runnable {
```

El interface Runnable define un sólo método llamado run() que no acepta ningún argumento y que no devuelve ningún valor. Como la clase Clock implementa el interface Runnable, debe proporcionar una implementación para el método run() como está definido en el interface. Sin embargo, antes de explicar el método run() de la clase Clock, echemos un vistazo a los otros elementos del código de esta clase:

Crear el Thread

La aplicación en la que se ejecuta el applet llama al método start() del applet cuando el usuario visita la página del applet. El applet del reloj crea un Thread, clockThread, en su método start() y arranca el thread.

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

Primero el método `start()` comprueba si `clockThread` es nulo. Si lo es, significa que el applet acaba de ser cargado o que ha sido parado anteriormente y se debe crear un nuevo Thread. De otro modo, significa que el applet ya se está ejecutando. El applet crea un nuevo Thread con esta llamada:

```
clockThread = new Thread(this, "Clock");
```

Observa que `this` -- el applet `Clock` -- es el primer argumento del constructor del thread. El primer argumento de este constructor Thread debe implementar el interface `Runnable` y se convierte en el origen del thread. Cuando se construye de esta forma, el thread del reloj obtiene su método `run()` desde el objeto `Runnable` origen -- en este caso el applet `Clock`.

El segundo argumento es sólo el nombre del thread.

Parar el Thread

Cuando abandones la página que muestra el Reloj, la aplicación en la que el applet se está ejecutando llama al método `stop()` del applet. El método `stop()` del applet `Clock` pone `clockThread` a nulo. Esto le dice al bucle principal en el método `run()` que termine (observa la siguiente sección), la actualización del reloj resultando eventualmente en la parada del thread y la recolección de basura.

```
public void stop() {  
    clockThread = null;  
}
```

Podrías utilizar `clockThread.stop()` en su lugar, lo que pararía inmediatamente el thread del reloj. Sin embargo, el método `stop()` de la clase `Thread` tiene un efecto súbito, lo que significa que el método `run()` podría estar en medio de una operación crítica cuando se pare el thread. Para los métodos `run()` más complejos, utilizar el método `stop()` de la clase `Thread` podría dejar el programa en un estado inconsistente. Por esta razón, es mejor evitar el uso del método `stop()` de la clase `Thread` cuando sea posible.

Si revisita la página de nuevo, se llama otra vez al método `start()` y el reloj arranca de nuevo con un nuevo thread.

El Método Run

Y finalmente, el método `run()` del applet `Clock` implementa el corazón de este applet y se parece a esto:

```
public void run() {  
    // El bucle termina cuando clockThread se pone a null en stop()  
    while (Thread.currentThread() == clockThread) {  
        repaint();  
        try {  
            clockThread.sleep(1000);  
        } catch (InterruptedException e){  
        }  
    }  
}
```

Como se vió en la sección anterior, cuando se le pide al applet que se pare, este selecciona `clockThread` a `null`; esto permite que el método `run()` sepa cuando debe parar. Así, la primera línea del método `run()` tiene un bucle hasta que `clockThread` sea nulo. Dentro del bucle, el applet se repinta a sí mismo y le dice al Thread que espere durante 1 segundo (1000 milisegundos). Luego el método `repaint()` del applet llama al método `paint()` del applet, que actualiza el área de pantalla del applet. El método `paint()` de nuestro applet `Clock` obtiene la hora actual y la muestra en la pantalla:

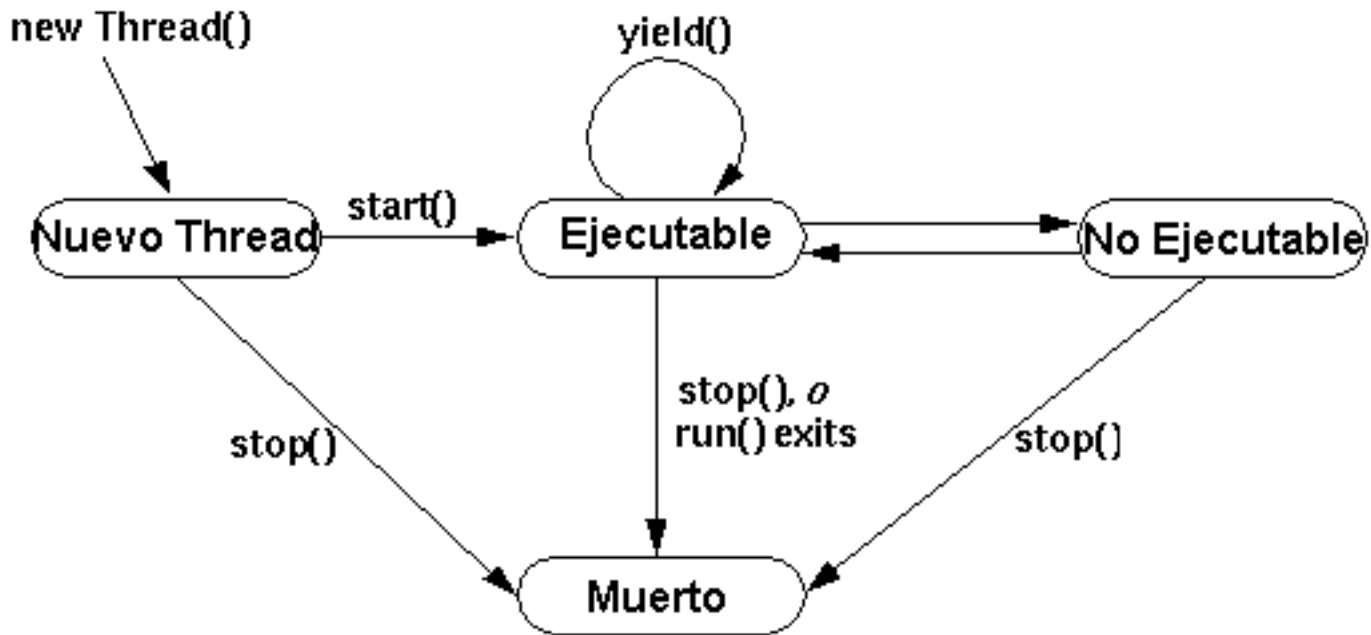
```
public void paint(Graphics g) {  
    Date now = new Date();  
    g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5,  
10);
```


}

Ozito

El Estado del Thread

El siguiente diagrama ilustra los distintos estados que puede tener un Thread Java en cualquier momento de su vida. También ilustra las llamadas a métodos que provocan las transiciones de un estado a otro. Este no es un diagrama de estado finito pero da una idea general de las facetas más interesantes y comunes en la vida de un thread. El resto de esta página explica el ciclo de vida de un thread, basándose en sus estados.



Un "Nuevo Thread"

La siguiente sentencia crea un nuevo thread pero no lo arranca, por lo tanto deja el thread en el estado : "New Thread" = "Nuevo Thread".

```
Thread miThread = new MiClaseThread();
```

Cuando un thread está en este estado, es sólo un objeto Thread vacío. No se han asignado recursos del sistema todavía para el thread. Así, cuando un thread está en este estado, lo único que se puede hacer es arrancarlo o pararlo. Llamar a otros métodos distintos de `start()` o `stop()` no tiene sentido y causa una excepción del tipo [IllegalThreadStateException](#).

Ejecutable

Ahora consideremos estas dos líneas de código:

```
Thread miThread = new MiClaseThread();
miThread.start();
```

Cuando el método `start()` crea los recursos del sistema necesarios para ejecutar el thread, programa el thread para ejecutarse, y llama al

método `run()` del thread. En este punto el thread está en el estado "Ejecutable". Este estado se llama "Ejecutable" mejor que "Ejecutando" ya que el thread todavía no ha empezado a ejecutarse cuando está en este estado. Muchos procesadores tienen un sólo procesador, haciendo posible que todos los threads sean "Ejecutables" al mismo tiempo. Por eso, el sistema de ejecución de Java debe implementar un esquema de programación para compartir el procesador entre todos los threads "Ejecutables". (Puedes ver la página [Prioridad de un Thread](#) para obtener más información sobre la programación.) Sin embargo, para la mayoría de los propósitos puedes pensar en "Ejecutable" como un sencillo "Ejecutando". Cuando un thread se está ejecutando -- está "Ejecutable" y es el thread actual -- las instrucciones de su método `run()` se ejecutan de forma secuencial.

No Ejecutable

Un thread entra en el estado "No Ejecutable" cuando ocurre uno de estos cuatro eventos:

- Alguien llama a su método `sleep()`.
- Alguien llama a su método `suspend()`.
- El thread utiliza su método `wait()` para esperar una condición variable.
- El thread está bloqueado durante la I/O.

Por ejemplo, la línea en negrita del siguiente fragmento de código pone a dormir `miThread` durante 10 segundos (10.000 milisegundos):

```
Thread miThread = new MiClaseThread();
miThread.start();
try {
    miThread.sleep(10000);
} catch (InterruptedException e){
}
```

Durante los 10 segundos que `miThread` está dormido, incluso si el proceso se vuelve disponible `miThread` no se ejecuta. Después de 10 segundos, `miThread` se convierte en "Ejecutable" de nuevo y, si el procesador está disponible se ejecuta.

Para cada entrada en el estado "No Ejecutable" mostrado en figura, existe una ruta de escape distinta y específica que devuelve el thread al estado "Ejecutable". Una ruta de escape sólo trabaja para su entrada correspondiente. Por ejemplo, si un thread ha sido puesto a dormir durante un cierto número de milisegundos deben pasar esos milisegundos antes de volverse "Ejecutable" de nuevo. Llamar al método `resume()` en un thread dormido no tiene efecto.

Esta lista indica la ruta de escape para cada entrada en el estado "No Ejecutable":

- Si se ha puesto a dormir un thread, deben pasar el número de milisegundos especificados.
- Si se ha suspendido un thread, alguien debe llamar a su método `resume()`.
- Si un thread está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante `notify()` o `notifyAll()`.
- Si un thread está bloqueado durante la I/O, cuando se complete la I/O.

Muerto

Un thread puede morir de dos formas: por causas naturales o siendo asesinado (parado). Una muerte natural se produce cuando su método `run()` sale normalmente. Por ejemplo, el bucle `while` en este método es un bucle finito -- itera 100 veces y luego sale.

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

Un thread con este método `run()` moriría naturalmente después de que el bucle y el método `run()` se hubieran completado.

También puede matar un thread en cualquier momento llamando a su método `stop()`. El siguiente código crea y arranca `miThread` luego lo pone a dormir durante 10 segundos. Cuando el thread actual se despierta, la línea en negrita mata `miThread`.

```
Thread miThread = new MiClaseThread();
miThread.start();
try {
    Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
miThread.stop();
```

El método `stop()` lanza un objeto `ThreadDeath` hacia al thread a eliminar. Así, cuando se mata al thread de esta forma, muere de forma asíncrona. El thread moriría cuando reciba realmente la excepción `ThreadDeath`.

El método `stop()` provoca una terminación súbita del método `run()` del

thread. Si el método `run()` estuviera realizando cálculos sensibles, `stop()` podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método `stop()` pero si se debería proporcionar una terminación educada como la selección de una bandera que indique que el método `run()` debería salir.

La Excepción `IllegalThreadStateException`

El sistema de ejecución lanza una excepción `IllegalThreadStateException` cuando llama a un método en un thread y el estado del thread no permite esa llamada a método. Por ejemplo, esta excepción se lanza cuando se llama a `suspend()` en un thread que no está "Ejecutable".

Como se ha mostrado en varios ejemplos de threads en esta lección, cuando se llame a un método de un thread que pueda lanzar una excepción, se debe capturar y manejar la excepción, o especificar al método llamador que se lanza la excepción no capturada. Puedes ver la información sobre el manejo de excepciones en Java en [Excepciones](#)

El Método `isAlive()`

Una última palabra sobre el estado del thread: el interface de programación de la clase `Thread` incluye un método llamado `isAlive()`. Este método devuelve `true` si el thread ha sido arrancado y no ha parado. Así, si el método `isAlive()` devuelve `false` sabrás que se trata de un "Nuevo thread" o de un thread "Muerto". Por el contrario si devuelve `true` sabrás que el thread es "Ejecutable" o "No Ejecutable". No se puede diferenciar entre un "Nuevo thread" y un thread "Muerto", como tampoco se puede hacer entre un thread "Ejecutable" y otro "No Ejecutable".

La Prioridad de un Thread

Anteriormente en esta lección , hemos reclamado que los applets se ejecuten de forma concurrente. Mientras conceptualmente esto es cierto, en la práctica no lo es. La mayoría de las configuraciones de ordenadores sólo tienen una CPU, por eso los threads realmente se ejecutan de uno en uno de forma que proporcionan una ilusión de concurrencia. La ejecución de varios threads en una sola CPU, en algunos órdenes, es llamada programación. El sistema de ejecución de Java soporta un algoritmo de programación determinístico muy sencillo conocido como programación de prioridad fija. Este algoritmo programa los threads basándose en su prioridad relativa a otros threads ["Ejecutables"](#).

Cuando se crea un thread Java, hereda su prioridad desde el thread que lo ha creado. También se puede modificar la prioridad de un thread en cualquier momento después de su creación utilizando el método `setPriority()` . Las prioridades de un thread son un rango de enteros entre `MIN_PRIORITY` y `MAX_PRIORITY` (constantes definidas en la clase `Thread`). El entero más alto, es la prioridad más alta. En un momento dado, cuando varios threads están listos para ser ejecutados, el sistema de ejecución elige aquellos thread "Ejecutables" con la prioridad más alta para su ejecución. Sólo cuando el thread se para, abandona o se convierte en ["No Ejecutable"](#) por alguna razón empezará su ejecución un thread con prioridad inferior. Si dos threads con la misma prioridad están esperando por la CPU, el programador elige uno de ellos en una forma de competición. El thread elegido se ejecutará hasta que ocurra alguna de las siguientes condiciones:

- Un thread con prioridad superior se vuelve "Ejecutable".
- Abandona, o su método `run()` sale.
- En sistemas que soportan tiempo-compartido, su tiempo ha expirado.

Luego el segundo thread tiene una oportunidad para ejecutarse, y así continuamente hasta que el interprete abandone.

El algoritmo de programación de threads del sistema de ejecución de Java también es preemptivo. Si en cualquier momento un thread con prioridad superior que todos los demás se vuelve "Ejecutable", el sistema elige el nuevo thread con prioridad más alta. Se dice que el thread con prioridad superior prevalece sobre los otros threads.

Regla del Pulgar: En un momento dado, el thread con prioridad superior se está ejecutando. Sin embargo, este no es una garantía. El programador de threads podría elegir otro thread con prioridad inferior para evitar el hambre. Por esta razón, el uso de las prioridades sólo afecta a la política del programador para propósitos de eficiencia. No dependas de la prioridad de los threads para algoritmos incorrectos.

La carrera de Threads

Este [código fuente Java](#) implementa un applet que anima una carrera entre dos threads "corredores" con diferentes prioridades. Cuando pulses con el ratón sobre el applet, arrancan los dos corredores. El corredor superior ,

llamado "2", tiene una prioridad 2. El segundo corredor, llamado "3", tiene una prioridad 3.

Prueba esto: Pulsa sobre el applet inferior para iniciar la carrera.

Este es el método run() para los dos [corredores](#).

```
public int tick = 1;
public void run() {
    while (tick < 400000) {
        tick++;
    }
}
```

Este método sólo cuenta desde 1 hasta 400.000. La variable tick es pública porque la utiliza el applet para determinar cuanto ha progresado el corredor (cómo de larga es su línea).

Además de los dos threads corredores, el applet tiene un tercer thread que controla el dibujo. El método run() de este thread contiene un bucle infinito; durante cada iteración del bucle dibuja una línea para cada corredor (cuya longitud se calcula mediante la variable tick), y luego duerme durante 10 milisegundos. Este thread tiene una prioridad de 4 -- superior que la de los corredores. Por eso, siempre que se despierte cada 10 milisegundos, se convierte en el thread de mayor prioridad, prevalece sobre el thread que se está ejecutando, y dibuja las líneas. Se puede ver cómo las líneas van atravesando la página.

Como puedes ver, esto no es una carrera justa porque un corredor tiene más prioridad que el otro. Cada vez que el thread que dibuja abandona la CPU para irse a dormir durante 10 milisegundos, el programador elige el thread ejecutable con una prioridad superior; en este caso, siempre será el corredor llamado "3". Aquí tienes otra versión del applet que implementa una carrera justa, esto es, los dos corredores tienen la misma prioridad y tienen las mismas posibilidades para ser elegidos.

Prueba esto: Pulsa sobre el Applet para iniciar la carrera.

En esta carrera, cada vez que el thread de dibujo abandona la CPU, hay dos threads ejecutables con igual prioridad -- los corredores -- esperando por la CPU; el programador debe elegir uno de los threads. En esta situación, el programador elige el siguiente thread en una especie de competición deportiva.

Threads Egoistas

La clase Runner utilizada en las carreras anteriores realmente implementa un comportamiendo "socialmente-perjudicioso". Recuerda el método run() de la clase Runner utilizado en las carreras:

```
public int tick = 1;
public void run() {
```

```

        while (tick < 400000) {
            tick++;
        }
    }
}

```

El bucle while del método run() está en un método ajustado. Esto es, una vez que el programador elige un thread con este cuerpo de thread para su ejecución, el thread nunca abandona voluntariamente el control de la CPU -- el thread se continúa ejecutando hasta que el bucle while termina naturalmente o hasta que el thread es superado por un thread con prioridad superior.

En algunas situaciones, tener threads "egoistas" no causa ningún problema porque prevalecen los threads con prioridad superior (como el thread del dibujo prevalece sobre los threads egoistas de los corredores. Sin embargo, en otras situaciones, los threads con métodos run() avariciosos de CPU, como los de la clase Runner, pueden tomar posesión de la CPU haciendo que otros threads esperen por mucho tiempo antes de obtener una oportunidad para ejecutarse.

Tiempo-Compartido

En sistemas, como Windows 95, la lucha contra el comportamiento egoista de los threads tiene una estrategia conocida como tiempo-compartido. Esta estrategia entra en juego cuando existen varios threads "Ejecutables" con igual prioridad y estos threads son los que tienen una prioridad mayor de los que están compitiendo por la CPU. Por ejemplo, este [programa Java](#) (que está basado en la carrera de Applets anterior) crea dos [threads egoistas](#) con la misma prioridad que tienen el siguiente método run():

```

public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
        }
    }
}

```

Este método contiene un bucle ajustado que incrementa el entero tick y cada 50.000 ticks imprime el identificador del thread y su contador tick.

Cuando se ejecuta el programa en un sistema con tiempo-compartido, verás los mensajes de los dos threads, intermitentemente uno y otro. Como esto:

```

Thread #1, tick = 50000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 150000

```



```
Thread #0, tick = 200000
Thread #1, tick = 250000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000
```

Esto es porque un sistema de tiempo compartido divide la CPU en espacios de tiempo e iterativamente le da a cada thread con prioridad superior un espacio de tiempo para ejecutarse. El sistema de tiempo compartido itera a través de los threads con la misma prioridad superior otorgándoles un pequeño espacio de tiempo para que se ejecuten, hasta que uno o más de estos threads finalizan, o hasta que aparezca un thread con prioridad superior. Observa que el tiempo compartido no ofrece garantías sobre la frecuencia y el orden en que se van a ejecutar los threads.

Cuando ejecutes este programa en un sistema sin tiempo compartido, sin embargo, veras que los mensajes de un thread terminan de imprimirse antes de que el otro tenga una oportunidad de mostrar un sólo mensaje. Como esto:

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

Esto es porque el sistema sin tiempo compartido elige uno de los threads con igual prioridad para ejecutarlo y le permite ejecutarse hasta que abandone la CPU o hasta que aparezca un thread con prioridad superior.

Nota: El sistema de ejecución Java no implementa (y por lo tanto no garantiza) el tiempo compartido. Sin embargo, algunos sistemas en los que se puede ejecutar Java sí soportan el tiempo compartido. Los programas Java no deberían ser relativos al tiempo compartido ya que podrían producir resultados diferentes en distintos sistemas.

Prueba esto: Compila y ejecuta las clases [RaceTest](#) y [SelfishRunner](#) en tu ordenador. ¿Puedes decir si su sistema tiene tiempo compartido?

Como te puedes imaginar, escribir código que haga un uso intensivo de la CPU puede tener repercusiones negativas en otros threads que se ejecutan en el mismo proceso. En general, se debería intentar escribir threads con "buen comportamiento" que abandonen voluntariamente la CPU de forma periódica y le den una oportunidad a otros threads para que se ejecuten. En particular, no escribas nunca código Java que trate con tiempo compartido-- esto garantiza prácticamente que tu programa dará diferentes resultados en distintos sistemas de ordenador.

Un thread puede abandonar la CPU (sin ir a dormir o algún otro método drástico) con una llamada al método `yield()`. Este método da una oportunidad a otros threads con la misma prioridad. Si no existen otros threads con la misma prioridad en el estado "ejecutable", este método será ignorado.

Prueba esto: Reescribe la clase `SelfishRunner` para que sea un [PoliteRunner](#) "Corredor Educado" mediante una llamada al método `yield()` desde el método `run()`. Asegúrese de modificar el [programa principal](#) para crear `PoliteRunners` en vez de `SelfishRunners`. Compila y ejecuta las nuevas clases en tu ordenador. ¿No está mejor ahora?

Sumario

- La mayoría de los ordenadores sólo tienen una CPU, los threads deben compartir la CPU con otros threads. La ejecución de varios threads en un sólo CPU, en cualquier orden, se llama programación. El sistema de ejecución Java soporta un algoritmo de programación determinístico que es conocido como programación de prioridad fija.
- A cada thread Java se le da una prioridad numérica entre `MIN_PRIORITY` y `MAX_PRIORITY` (constantes definidas en la clase `Thread`). En un momento dado, cuando varios threads están listos para ejecutarse, el thread con prioridad superior será el elegido para su ejecución. Sólo cuando el thread para o se suspende por alguna razón, se empezará a ejecutar un thread con prioridad inferior.
- La programación de la CPU es totalmente preemptiva. Si un thread con prioridad superior que el que se está ejecutando actualmente necesita ejecutarse, toma inmediatamente posesión del control sobre la CPU.
- El sistema de ejecución de Java no hace abandonar a un thread el control de la CPU por otro thread con la misma prioridad. En otras palabras, el sistema de ejecución de Java no comparte el tiempo. Sin embargo, algunos sistemas si lo soportan por lo que no se debe escribir código que esté relacionado con el tiempo compartido.
- Además, un thread cualquiera, en cualquier momento, puede ceder el control de la CPU llamando al método `yield()`. Los threads sólo pueden 'prestar' la CPU a otros threads con la misma prioridad que él -- intentar cederle la CPU a un thread con prioridad inferior no tendrá ningún efecto.

- Cuando todos los threads "ejecutables" del sistema tienen la misma prioridad, el programador elige a uno de ellos en una especie de orden de competición.
-

Threads Servidores

Cualquier thread Java puede ser un thread daemon "Servidor". Los threads daemon proporcionan servicios para otros threads que se están ejecutando en el mismo proceso que él. Por ejemplo, el navegador HotJava utiliza cuatro threads daemon llamados "Image Fetcher" para buscar imágenes en el sistema de ficheros en la red para los threads que las necesiten. El método `run()` de un thread daemon normalmente es un bucle infinito que espera una petición de servicio.

Cuando el único thread en un proceso es un thread daemon, el interprete sale. Esto tiene sentido porque al permanecer sólo el thread daemon, no existe ningún otro thread al que poder proporcionale un servicio.

Para especificar que un thread es un thread daemon, se llama al método `setDaemon()` con el argumento `true`. Para determinar si un thread es un thread daemon se utiliza el método accesor `isDaemon()`.

Grupos de Threads

Cada thread de Java es miembro de un grupo de threads. Los grupos proporcionan un mecanismo para la colección de varios threads dentro de un sólo objeto y la manipulación de esos threads de una vez, mejor que de forma individual. Por ejemplo, se puede arrancar o suspender todos los threads de un grupo con una sólo llamada a un método. Los grupos de threads de Java están implementados por la clase `ThreadGroup` del paquete `java.lang`.

El sistema de ejecución pone un thread dentro de un grupo durante su construcción. Cuando se crea un thread, también se puede permitir que el sistema de ejecución ponga el nuevo thread en algún grupo por defecto razonable o se puede especificar explícitamente el grupo del nuevo thread. El thread es un miembro permanente del grupo al que se unió durante su creación -- no se puede mover un thread a otro grupo después de haber sido creado.

El Grupo de Threads por Defecto

Si se crea un nuevo `Thread` sin especificar su grupo en el constructor, el sistema de ejecución automáticamente sitúa el nuevo thread en el mismo grupo que el thread que lo creó (conocido como el grupo de threads actual y el thread actual, respectivamente). Entonces, si se deja sin especificar el grupo de threads cuando se crea un thread, ¿qué grupo contiene el thread?

Cuando se arranca por primera vez una aplicación Java, el sistema de ejecución crea un `ThreadGroup` llamado "main". Entonces, a menos que se especifique otra cosa, todos los nuevos threads que se creen se convierten en miembros del grupo de threads "main".

Nota: Si se crea un thread dentro de un applet, el grupo del nuevo thread podría ser distinto de "main" -- depende del navegador o del visualizador donde se esté ejecutando al applet. Puedes referirte a [Threads en Applets](#) para obtener más información sobre los grupos de threads en los applets.

Muchos programadores java ignoran por completo los grupos de threads y permiten al sistema de ejecución que maneje todos los detalles con respecto a los grupos de threads. Sin embargo, si tu programa crea muchos threads que deben ser manipulados como un grupo, o si estás implementando un Controlador de Seguridad de cliente, probablemente querrás más control sobre los grupos de threads. Continúe leyendo para más detalles!

Crear un Thread en un Grupo Específico

Como se mencionó anteriormente, un thread es un miembro permanente del grupo al que se unió cuando fue creado -- no se puede mover un thread a otro grupo después de haber sido creado. Así, si se desea poner si nuevo thread en un grupo distinto al de defecto, se debe especificar explícitamente el grupo en que crear el thread. La clase `Thread` tiene tres constructores que permiten

seleccionar un nuevo grupo de threads:

```
public Thread(ThreadGroup grupo, Runnable fuente)
public Thread(ThreadGroup grupo, String nombre)
public Thread(ThreadGroup grupo, Runnable fuente, String nombre)
```

Cada uno de estos constructores crea un nuevo thread, lo inicializa basandose en los parámetros Runnable y String, y lo hace miembro del grupo especificado. Por ejemplo, el siguiente ejemplo crea un grupo de threads (miGrupoDeThread) y luego crea un thread (miThread) en ese grupo.

```
ThreadGroup miGrupoDeThread = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miGrupoDeThread, "un thread de mi grupo");
```

El ThreadGroup pasado al constructor del Thread no tiene que ser necesariamente un grupo que hayas creado -- puede ser un grupo creado por el sistema de ejecución Java, o un grupo creado por la aplicación donde se está ejecutando el applet.

Obtener el Grupo de un Thread

Para encontrar el grupo en el que está un thread, puede llamar a su método `getThreadGroup()`.

```
theGroup = miThread.getThreadGroup();
```

La Clase ThreadGroup

Una vez que obtenido el Grupo de un thread, puedes pedir más información sobre el grupo, como cuántos threads más hay en el grupo. También se pueden modificar todos los threads del grupo (como suspenderlos, pararlos, etc...) con una sola llamada a un método.

La Clase ThreadGroup

La clase ThreadGroup maneja grupos de threads para las aplicaciones Java. Un ThreadGroup puede contener cualquier número de threads. Los threads de un grupo generalmente están; relacionados de alguna forma, como por quién fueron creados, qué función realizan o cuándo deben arrancar o parar.

Un threadGroup no sólo puede contener threads, también puede contener otros ThreadGroups. El grupo principal en una aplicación Java es el grupo de threads llamado "main". Se pueden crear threads y grupos de threads dentro del grupo "main". También se pueden crear threads y grupos de threads dentro de subgrupos de "main" y así sucesivamente. Esto resulta en una herencia del tipo raíz de los threads y los grupos de threads.

La clase ThreadGroup tiene métodos que pueden ser categorizados de la siguiente forma:

Métodos de Manejo de la Colección

El ThreadGroup proporciona un juego de métodos que maneja los threads y los subgrupos dentro de un grupo y permite que otros objetos le pregunten a ThreadGroup sobre su contenido. Por ejemplo, se puede llamar al método activeCount() para encontrar el número de threads activos actualmente dentro del grupo. Este método se utiliza frecuentemente con el método enumerate() para obtener un array con las referencias de todos los threads activos en el ThreadGroup. Por ejemplo, el método listCurrentThreads() del siguiente ejemplo rellena un array con todos los threads activos en el grupo actual e impime sus nombres:

```
class EnumerateTest {
    void listCurrentThreads() {
        ThreadGroup currentGroup = Thread.currentThread().getThreadGroup();
        int numThreads;
        Thread[] listOfThreads;

        numThreads = currentGroup.activeCount();
        listOfThreads = new Thread[numThreads];
        currentGroup.enumerate(listOfThreads);
        for (int i = 0; i < numThreads; i++) {
            System.out.println("Thread #" + i + " = " + listOfThreads[i].getName());
        }
    }
}
```

Otros métodos de manejo de la colección proporcionados por la clase ThreadGroup incluyen activeGroupCount() y list().

Métodos que Operan sobre el Grupo

La clase ThreadGroup soporta varios atributos que son seleccionados y recuperados para el grupo en su totalidad. Estos atributos incluyen la prioridad máxima que un thread puede tener dentro del grupo, si el grupo es un grupo "daemon", el nombre del grupo, y el nombre del padre del grupo.

Los métodos que seleccionan y obtienen estos atributos operan a nivel de grupo. Esto es, pueden inspeccionar el atributo del objeto ThreadGroup, pero no afectan a los threads que hay dentro del grupo. El siguiente listado muestra los métodos de ThreadGroup que operan a nivel de grupo:

- getMaxPriority(), y setMaxPriority()
- getDaemon(), y setDaemon()
- getName()
- getParent(), y parentOf()

- toString()

Así, por ejemplo, cuando se utiliza `setMaxPriority()` para cambiar la prioridad máxima del grupo, sólo está cambiando el atributo en el grupo, no está cambiando la prioridad de ninguno de los thread del grupo. Consideremos este pequeño programa que crea un grupo y un thread dentro de él:

```
class MaxPriorityTest {
    public static void main(String[] args) {

        ThreadGroup groupNORM = new ThreadGroup(
            "Un grupo con prioridad normal");
        Thread priorityMAX = new Thread(groupNORM,
            "Un thread con prioridad máxima");

        // Selecciona la prioridad del thread al máximo (10)
        priorityMAX.setPriority(Thread.MAX_PRIORITY);

        // Selecciona la prioridad del grupo a normal (5)
        groupNORM.setMaxPriority(Thread.NORM_PRIORITY);

        System.out.println("Máxima prioridad del grupo = " +
            groupNORM.getMaxPriority());
        System.out.println("Prioridad del Thread = " +
            priorityMAX.getPriority());
    }
}
```

Cuando se crea el grupo `groupNORM` hereda su atributo de prioridad máxima desde su grupo padre. En este caso, la prioridad del grupo padre es la máxima (`MAX_PRIORITY`) permitida por el sistema de ejecución de Java. Luego el programa selecciona la prioridad del thread `priorityMAX` al máximo permitido por el sistema Java. Luego el programa baja la prioridad del grupo a normal (`NORM_PRIORITY`). El método `setMaxPriority()` no afecta a la prioridad del thread `priorityMAX`, por eso en este punto, el thread `priorityMAX` tienen una prioridad de 10 que es mayor que la prioridad máxima de su grupo `groupNORM`. Esta es la salida del programa:

```
Prioridad máxima del Grupo = 5
Prioridad del Thread = 10
```

Como puedes ver un thread puede tener una prioridad superior que el máximo permitido por su grupo siempre que la prioridad del thread se haya seleccionado antes de haber bajado la prioridad máxima del grupo. La prioridad máxima de un grupo de threads se utiliza para limitar la prioridad de los threads cuando son creados dentro de un grupo o cuando se utiliza `setPriority()` para cambiar la prioridad del thread. Observa que `setMaxPriority()` también cambia la prioridad máxima de todos sus subgrupos.

Sin embargo, el estado `daemon` de un grupo sólo se aplica al grupo. Cambiar el estado `daemon` de un grupo no afecta al estado `daemon` de los threads que hay dentro del grupo. Además, el estado `daemon` de un grupo no implica de ninguna forma el estado `daemon` de sus threads -- se puede poner cualquier thread dentro de un grupo de threads `daemon`. El `daemon` de un grupo de threads sólo implica que el grupo puede ser destruido cuando todos los threads se han terminado.

Métodos que Operan con Todos los Threads de un Grupo.

La clase `ThreadGroup` tiene tres métodos que le permiten modificar el estado actual de todos los threads de un grupo.

- `resume()`
- `stop()`
- `suspend()`

Estos métodos aplican el cambio de estado apropiado a todos los threads del grupo y sus subgrupos.

Métodos de Restricción de Acceso

La clase ThreadGroup por si misma no impone ninguna restricción de acceso, como permitir que los threads de un grupo puedan inspeccionar o modificar threads de un grupo diferente. Mas bien las clases Thread y ThreadGroup cooperan con los manejadores de seguridad (subclases de la clase java.lang.SecurityManager), que puede imponer restricciones de acceso basándose en los miembros de un grupo de threads.

Las clases Thread y threadGroup tienen un método, checkAccess(), que llama al método checkAccess() del controlador de seguridad actual. El controlador de seguridad decide si permite el acceso basándose en los miembros del grupo de threads involucrado. Si el acceso no está permitido, el método checkAccess() lanza una excepción SecurityException. De otro modo el método checkAccess() simplemente retorna.

La siguiente lista muestra varios métodos de ThreadGroup que llaman a checkAccess() antes de realizar la acción del método. Esto se conoce como acceso regulado, esto es, accesos que deben ser aprobados por el controlador de seguridad antes de poder ser completados.

- ThreadGroup(ThreadGroup **padre**, String **nombre**)
- setDaemon(boolean **isDaemon**)
- setMaxPriority(int **maxPriority**)
- stop()
- suspend()
- resume()
- destroy()

Esta es una lista de métodos de la clase Thread que llaman a checkAccess() antes de proceder:

- Constructores que especifican un grupo de threads.
- stop()
- suspend()
- resume()
- setPriority(int **priority**)
- setName(String **name**)
- setDaemon(boolean **isDaemon**)

Una aplicación Java solitaria no tiene un controlador de seguridad por defecto. Esto es, por defecto, no se imponen restricciones a ningún thread para que pueda inspeccionar o modificar cualquier otro thread, sin importar el grupo en el que se encuentra. Se puede definir e implementar propias restricciones de acceso para los grupos de threads mediante la subclasificación de la clase SecurityManager, sobrescribiendo los métodos apropiados, e instalándolo como el controlador de seguridad para su aplicación.

El navegador HotJava es un ejemplo de aplicación que implementa su propio controlador de seguridad. HotJava necesita asegurarse de que los applets tengan un buen comportamiento y no hagan cosas sucias a otros applets que se están ejecutando al mismo tiempo (como bajar la prioridad de otros threads de otros applets). El controlador de seguridad de HotJava no permite que un thread modifique threads de otro grupo. Por favor, observa que las restricciones de acceso basadas en los grupos de threads pueden variar de un navegador a otro y por eso tus applets pueden tener comportamientos diferentes en diferentes navegadores.

Programas con Varios Threads

Sincronización de Threads

Frecuentemente, los threads necesitan compartir datos. Por ejemplo, supongamos que existe un thread que escribe datos en un fichero mientras, al mismo tiempo, otro thread está leyendo el mismo fichero. Cuando los threads comparten información necesitan sincronizarse para obtener los resultados deseados.

Imparcialidad, Hambre y Punto Muerto

Si se escribe un programa en el que varios threads concurrentes deben competir por los recursos, se debe tomar las precauciones necesarias para asegurarse la justicia. Un sistema es justo cuando cada thread obtiene suficiente acceso a los recursos limitados como para tener un progreso razonable. Un sistema justo previene el hambre y el punto muerto. El hambre ocurre cuando uno o más threads de un programa están bloqueados por ganar el acceso a un recurso y así no pueden progresar. El punto muerto es la última forma de hambre; ocurre cuando dos o más threads están esperando una condición que no puede ser satisfecha. El punto muerto ocurre muy frecuentemente cuando dos (o más) threads están esperando a que el otro u otros haga algo.

Volatile

Los programas pueden modificar variables miembros fuera de la protección de un método o un bloque sincronizados y puede declarar que la variable miembro es volatile.

Si una variable miembro es declarada como volatile, el sistema de ejecución Java utiliza esta información para asegurarse que la variable sea cargada desde la memoria antes de cada uso, y almacenada en la memoria después de utilizarla. Esto asegura que el valor de la variable es consistente y coherente a lo largo del programa.

Sincronización de Threads

Las lecciones anteriores contenían ejemplos con threads asíncronos e independientes. Esto es, cada thread contenía todos los datos y métodos necesarios y no requerían recursos externos. Además, los threads de esos ejemplos se ejecutaban en su propio espacio sin preocuparse sobre el estado o actividad de otros threads que se ejecutaban de forma concurrente.

Sin embargo, existen muchas situaciones interesantes donde ejecutar threads concurrentes que compartan datos y deban considerar el estado y actividad de otros threads. Este conjunto de situaciones de programación son conocidos como escenarios 'productor/consumidor'; donde el productor genera un canal de datos que es consumido por el consumidor.

Por ejemplo, puedes imaginar una aplicación Java donde un thread (el productor) escribe datos en un fichero mientras que un segundo thread (el consumidor) lee los datos del mismo fichero. O si tecleas caracteres en el teclado, el thread productor sitúa las pulsaciones en una pila de eventos y el thread consumidor lee los eventos de la misma pila. Estos dos ejemplos utilizan threads concurrentes que comparten un recurso común; el primero comparte un fichero y el segundo una pila de eventos. Como los threads comparten un recurso común, deben sincronizarse de alguna forma.

Esta lección enseña la sincronización de threads Java mediante un sencillo ejemplo de productor/consumidor.

El Ejemplo Productor/Consumidor

El [Productor](#) genera un entero entre 0 y 9 (inclusive), lo almacena en un objeto "CubbyHole", e imprime el número generado. Para hacer más interesante el problema de la sincronización, el productor duerme durante un tiempo aleatorio entre 0 y 100 milisegundos antes de repetir el ciclo de generación de números:

```
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Productor #" + this.number + " pone: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

El [Consumidor](#), estando hambriento, consume todos los enteros de CubbyHole (exactamente el mismo objeto en que el productor puso los enteros en primer lugar) tan rápidamente como estén disponibles.

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
    }
}
```

```

        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumidor #" + this.number + " obtiene: " + value);
        }
    }
}

```

En este ejemplo el Productor y el Consumidor comparten datos a través de un objeto CubbyHole común. Observarás que ninguno de los dos hace ningún esfuerzo sea el que sea para asegurarse de que el consumidor obtiene cada valor producido una y sólo una vez. La sincronización entre estos dos threads realmente ocurre a un nivel inferior, dentro de los métodos `get()` y `put()` del objeto `CubbyHole`. Sin embargo, asumamos por un momento que estos dos threads no están sincronizados y veamos los problemas potenciales que podría provocar esta situación.

Un problema sería cuando el Productor fuera más rápido que el Consumidor y generara dos números antes de que el Consumidor tuviera una posibilidad de consumir el primer número. Así el Consumidor se saltaría un número. Parte de la salida se podría parecer a esto:

```

. . .
Consumidor #1 obtiene: 3
Productor #1 pone: 4
Productor #1 pone: 5
Consumidor #1 obtiene: 5
. . .

```

Otro problema podría aparecer si el consumidor fuera más rápido que el Productor y consumiera el mismo valor dos o más veces. En esta situación el Consumidor imprimirá el mismo valor dos veces y podría producir una salida como esta:

```

. . .
Productor #1 pone: 4
Consumidor #1 obtiene: 4
Consumidor #1 obtiene: 4
Productor #1 pone: 5
. . .

```

De cualquier forma, el resultado es erróneo. Se quiere que el consumidor obtenga cada entero producido por el Productor y sólo una vez. Los problemas como los escritos anteriormente, se llaman condiciones de carrera. Se alcanzan cuando varios threads ejecutados asincrónicamente intentan acceder a un mismo objeto al mismo tiempo y obtienen resultados erróneos.

Para prevenir estas condiciones en nuestro ejemplo Productor/Consumidor, el almacenamiento de un nuevo entero en `CubbyHole` por el Productor debe estar sincronizado con la recuperación del entero por parte del Consumidor. El Consumidor debe consumir cada entero exactamente una vez. El programa Productor/Consumidor utiliza dos mecanismos diferentes para sincronizar los threads `Producer` y `Consumer`; los monitores, y los métodos `notify()` y `wait()`.

Monitores

Los objetos, como el `CubbyHole` que son compartidos entre dos threads y cuyo acceso debe ser sincronizado son llamados condiciones variables. El lenguaje Java permite sincronizar threads alrededor de una condición variable mediante el uso de monitores. Los monitores previenen que dos threads accedan simultáneamente a la misma variable.

Los métodos notify() y wait()

En un nivel superior, el ejemplo Productor/Consumidor utiliza los métodos notify() y wait() del objeto para coordinar la actividad de los dos threads. El objeto CubbyHole utiliza notify() y wait() para asegurarse de que cada valor situado en él por el Productor es recuperado una vez y sólo una por el Consumidor.

El programa Principal

Aquí tienes una pequeña [aplicación Java](#) que crea un objeto [CubbyHole](#), un Producer, un Consumer y arranca los dos threads.

```
class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

La Salida

Aquí tienes la salida del programa ProducerConsumerTest.

```
Producer #1 pone: 0
Consumidor #1 obtiene: 0
Productor #1 pone: 1
Consumidor #1 obtiene: 1
Productor #1 pone: 2
Consumidor #1 obtiene: 2
Productor #1 pone: 3
Consumidor #1 obtiene: 3
Productor #1 pone: 4
Consumidor #1 obtiene: 4
Productor #1 pone: 5
Consumidor #1 obtiene: 5
Productor #1 pone: 6
Consumidor #1 obtiene: 6
Productor #1 pone: 7
Consumidor #1 obtiene: 7
Productor #1 pone: 8
Consumidor #1 obtiene: 8
Productor #1 pone: 9
Consumidor #1 obtiene: 9
```

Monitores Java

El lenguaje Java y el sistema de ejecución soportan la sincronización de threads mediante el uso de monitores. En general, un monitor está asociado con un objeto específico (una condición variable) y funciona como un bloqueo para ese dato. Cuando un thread mantiene el monitor para algún dato del objeto, los otros threads están bloqueados y no pueden ni inspeccionar ni modificar el dato.

Los segmentos de código dentro de programa que acceden al mismo dato dentro de threads concurrentes separados son conocidos como secciones críticas. En el lenguaje Java, se pueden marcar las secciones críticas del programa con la palabra clave `synchronized`.

Nota: Generalmente, la sección críticas en los programas Java son métodos. Se pueden marcar segmentos pequeños de código como sincronizados. Sin embargo, esto viola los paradigmas de la programación orientada a objetos y produce un código que es difícil de leer y de mantener. Para la mayoría de los propósitos de programación en Java, es mejor utilizar `synchronized` sólo a nivel de métodos.

En el lenguaje Java se asocia un único monitor con cada objeto que tiene un método sincronizado. La clase [CubbyHole](#) del ejemplo Producer/Consumer de la [página anterior](#) tiene dos métodos sincronizados: el método `put()`, que se utiliza para cambiar el valor de `CubbyHole`, y el método `get()`, que se utiliza para el recuperar el valor actual. Así el sistema asocia un único monitor con cada ejemplar de `CubbyHole`.

Aquí tienes el código fuente del objeto `CubbyHole`. Las líneas en negrita proporcionan la sincronización de los threads:

```
class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        notify();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
```

```

        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    notify();
}
}

```

La clase CubbyHole tiene dos variables privadas: contents, que es el contenido actual de CubbyHole, y la variable booleana available, que indica si se puede recuperar el contenido de CubbyHole. Cuando available es verdadera indica que el Productor ha puesto un nuevo valor en CubbyHole y que el Consumidor todavía no la ha consumido. El Consumidor sólo puede consumir el valor de CubbyHole cuando available es verdadera.

Como CubbyHole tiene dos métodos sincronizados, java proporciona un único monitor para cada ejemplar de CubbyHole (incluyendo el compartido por el Productor y el Consumidor). Siempre que el control entra en un método sincronizado, el thread que ha llamado el método adquiere el monitor del objeto cuyo método ha sido llamado. Otros threads no pueden llamar a un método sincronizado del mismo objeto hasta que el monitor sea liberado.

Nota: [Los Monitores Java son Re-entrantes](#). Esto es, el mismo thread puede llamar a un método sincronizado de un objeto para el que ya tiene el monitor, es decir, puede re-adquirir el monitor.

Así, siempre que el Productor llama al método put() de CubbyHole, adquiere el monitor del objeto CubbyHole, y así evita que el consumidor pueda llamar al método get() de CubbyHole. (El método wait() libera temporalmente el monitor como se verá más adelante).

```

public synchronized void put(int value) {
    // El productor adquiere el monitor
    while (available == true) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    notify();
    // El productor libera el monitor
}

```

Cuando el método `put()` retorna, el Productor libera el monitor y por lo tanto desbloquea el objeto `CubbyHole`.

Siempre que el Consumidor llama al método `get()` de `CubbyHole`, adquiere el monitor de ese objeto y por lo tanto evita que el productor pueda llamar al método `put()`.

```
public synchronized int get() {  
    // El consumidor adquiere el monitor  
    while (available == false) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    available = false;  
    notify();  
    return contents;  
    // el Consumidor libera el monitor  
}
```

La adquisición y liberación del monitor la hace automáticamente el sistema de ejecución de Java. Esto asegura que no puedan ocurrir condiciones de competición en la implementación de los threads, asegurando la integridad de los datos.

Prueba esto: Elimina las líneas que están en **negrita** en el listado de la clase `CubbyHole` [mostrada arriba](#). Recompila el programa y ejecutalo de nuevo. ¿Qué sucede? Como no se ha realizado ningún esfuerzo explícito para sincronizar los threads, el Consumidor consume con un abandono temerario y obtiene sólo una ristra de ceros, en lugar de obtener los enteros entre 0 y 9 exactamente una vez cada uno.

Los Monitores Java son Re-entrantes

El sistema de ejecución de Java permite que un thread re-adquiera el monitor que ya posee realmente porque los monitores Java son re-entrantes. Los monitores re-entrantes son importantes porque eliminan la posibilidad de que un sólo thread ponga en punto muerto un monitor que ya posee.

Consideremos esta clase:

```
class Reentrant {
    public synchronized void a() {
        b();
        System.out.println("Estoy aquí, en a()");
    }
    public synchronized void b() {
        System.out.println("Estoy aquí, en b()");
    }
}
```

Esta clase contiene dos métodos sincronizados: a() y b(). El primer método sincronizado, llama al otro método sincronizado.

Cuando el control entra en el método a(), el thread actual adquiere el monitor del objeto Reentrant. Ahora, a() llama a b() y como también está sincronizado el thread también intenta adquirir el monitor de nuevo. Como Java soporta los monitores re-entrantes, esto si funciona. El thread actual puede adquirir de nuevo el monitor del objeto Reentrant y se ejecutan los dos métodos a() y b(), como evidencia la salida del programa:

```
Estoy aquí, en b()
Estoy aquí, en a()
```

En sistemas que no soportan monitores re-entrantes, esta secuencia de llamadas a métodos causaría un punto muerto.

Los Métodos notify() y wait()

Los métodos `get()` y `put()` del objeto `CubbyHole` hacen uso de los métodos `notify()` y `wait()` para coordinar la obtención y puesta de valores dentro de `CubbyHole`. Los métodos `notify()` y `wait()` son miembros de la clase `java.lang.Object`.

Nota: Los métodos `notify()` y `wait()` pueden ser invocados sólo desde dentro de un método sincronizado o dentro de un bloque o una sentencia sincronizada.

Investiguemos el uso del método `notify()` en `CubbyHole` mirando el método `get()`.

El método notify()

El método `get()` llama al método `notify()` como lo último que hace (junto retornar). El método `notify()` elige un thread que está esperando el monitor poseído por el thread actual y lo despierta. Normalmente, el thread que espera capturará el monitor y procederá.

El caso del ejemplo Productor/Consumidor, el thread Consumidor llama al método `get()`, por lo que el método Consumidor posee el monitor de `CubbyHole` durante la ejecución del método `get()`. Al final del método `get()`, la llamada al método `notify()` despierta al thread Productor que obtiene el monitor de `CubbyHole` y procede.

```
public synchronized int get() {
    while (available == false) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    available = false;
    notify();           // lo notifica al Productor
    return contents;
}
```

Si existen varios threads esperando por un monitor, el sistema de ejecución Java elige uno de esos threads, sin ningún compromiso ni garantía sobre el thread que será elegido.

El método `put()` trabaja de una forma similar a `get()`, despertando al thread consumidor que está esperando que el Productor libere el monitor.

La clase `Object` tiene otro método --`notifyAll()`-- que despierta todos los threads que están esperando al mismo monitor. En esta Situación, los threads despertados compiten por el monitor. Uno de ellos obtiene el monitor y los otros vuelven a esperar.

El método wait()

El método `wait()` hace que el thread actual espere (posiblemente para siempre) hasta que otro thread se lo notifique o a que cambie una condición. Se utiliza el método `wait()` en conjunción con el método `notify()` para coordinar la actividad de varios threads que utilizan los mismos recursos.

El método `get()` contiene una sentencia `while` que hace un bucle hasta que `available` se convierte en `true`. Si `available` es `false` -- el Productor todavía no ha producido un nuevo número y el consumidor debe esperar -- el método `get()` llama a `wait()`.

El bucle `while` contiene la llamada a `wait()`. El método `wait()` espera indefinidamente

hasta que llegue una notificación del thread Productor. Cuando el método `put()` llama a `notify()`, el Consumidor despierta del estado de espera y continúa con el bucle. Presumiblemente, el Productor ya ha generado un nuevo número y el método `get()` cae al final del bucle y procede. Si el Productor no ha generado un nuevo número, `get()` vuelve al principio del bucle y continua esperando hasta que el Productor genere un nuevo número y llame a `notify()`.

```
public synchronized int get() {
    while (available == false) {
        try {
            wait();           // espera una llamada a notify() desde el Productor
        } catch (InterruptedException e) {
        }
    }
    available = false;
    notify();
    return contents;
}
```

El método `put()` trabaja de un forma similar, esperando a que el thread Consumidor consuma el valor actual antes de permitir que el Productor genere uno nuevo.

Junto a la versión utilizada en el ejemplo de Productor/Consumidor, que espera indefinidamente una notificación, la clase `Object` contiene otras dos versiones del método `wait()`:

`wait(long timeout)`

Espera una notificación o hasta que haya pasado el tiempo de espera --`timeout` se mide en milisegundos.

`wait(long timeout, int nanos)`

Espera una notificación o hasta que hayan pasado **timeout** milisegundos mas **nanos** nanosegundos.

Los Monitores y los Métodos `notify()` y `wait()`

Habras observado un problema potencial en los métodos `put()` y `get()` de `CubbyHole`. Al principio del método `get()`, si el valor de `CubbyHole` no está disponible (esto es, el Productor no ha generado un nuevo número desde la última vez que el Consumidor lo consumió), luego el Consumidor espera a que el Productor ponga un nuevo número en `CubbyHole`. Aquí está la cuestión -- ¿cómo puede el Productor poner un nuevo valor dentro de `CubbyHole` si el Consumidor tiene el monitor? (El Consumidor posee el monitor de `CubbyHole` porque está dentro del método `get()` que está sincronizado).

Similarmente, al principio del método `put()`, si todavía no se ha consumido el valor, el Productor espera a que el Consumidor consuma el valor del `CubbyHole`. Y de nuevo llegamos a la cuestión -- ¿Cómo puede el consumidor obtener el valor de `CubbyHole`, si el Productor posee el monitor? (El productor posee el monitor de `CubbyHole` porque está dentro dentro del método `put()` que está sincronizado).

Bien, los diseñadores del lenguaje Java también pensaron en esto. Cuando el thread entra en el método `wait()`, lo que sucede al principio de los métodos `put()` y `get`, el monitor es liberado automáticamente, y cuando el thread sale del método `wait()`, se adquiere de nuevo el monitor. Esto le da una oportunidad al objeto que está esperando de adquirir el monitor y, dependiendo, de quién está esperando, consume el valor de `CubbyHole` o produce un nuevo valor para el `CubbyHole`.

Cambios en el JDK 1.1: Que afectan a los Threads de Control

[Los Applets de Ordenación](#)

Los applets de ordeanción utilizan métodos del AWT caducados. Puedes ver [Cambios en el JDK 1.1: Applet de Ordenación](#).

[El Applet del Reloj](#)

El ejemplo del Reloj utiliza métodos caducados de la clase Date. Puedes ver [Cambios en el JDK 1.1: El Applet del Reloj](#).

[La carrea de Threads](#)

El ejemplo RaceAppler utiliza métodos caducados del AWT. Puedes ver [Cambios en el JDK 1.1: Carrera de Applets](#).

[Grupos de Threads](#)

Se ha añadido un método a la clases ThreadGroup:
AllowThreadSuspension. Puedes ver [Cambios en el JDK 1.1: La clase ThreadGroup](#).

Los Canales de Entrada y Salida

Muchos programas Java necesitan leer o escribir datos.

Definición: Un stream (o canal) es un flujo secuencial de caracteres.

Un programa puede leer la entrada desde una fuente de datos leyendo una secuencia de caracteres desde un canal agregado a la fuente. Un programa puede producir una salida escribiendo una secuencia de caracteres en un canal agregado al destino. El entorno de desarrollo de Java incluye un paquete, `java.io`, que contiene un juego de canales de entrada y salida que los programas pueden utilizar para leer y escribir datos. Las clases `InputStream` y `OutputStream` del paquete `java.io` son superclases abstractas que definen el comportamiento de los canales de I/O secuenciales de Java. `java.io` también incluye muchas subclases de `InputStream` y `OutputStream` que implementan tipos específicos de canales de I/O. Esta lección explica que hace cada una de las clases del paquete `java.io`, cómo decidir cual utilizar, cómo utilizarlas, y cómo subclasificarlas para escribir sus propias clases de canales.

Ya que esta lección no tiene un ejemplo para cada tipo de canal de I/O disponible en el paquete `java.io`, proporciona muchos ejemplos prácticos de cómo utilizar las clases más populares de este paquete.

Consideraciones de Seguridad: La entrada y salida en el sistema local de ficheros está sujeta a la aprobación del controlador actual de seguridad. Los programas de ejemplo contenidos en estas lecciones son aplicaciones solitarias, que por defecto, no tienen controlador de seguridad. Si intentas ejecutar este código en applets podría no funcionar, dependiendo del navegador en que se esté ejecutando. Puede ver [Entender las Capacidades y Restricciones de los Applets](#) para obtener información sobre las restricciones de seguridad en los applets.

Primer Encuentro con la I/O en Java

Si has estado utilizando los canales de entrada y salida estandar, entonces has utilizado, quizás sin saberlo, los canales de I/O del paquete java.io.

Este programa de ejemplo utiliza `System.out.println()` para imprimir el texto "Hola Mundo!" en la pantalla.

```
class HelloWorldApp {
    public static void main (String[] args) {
        System.out.println("Hola Mundo!");
    }
}
```

`System.out` se refiere a un canal de salida manejado por la clase `System` que implementa el canal de salida estandar. `System.out` es un ejemplar de la clase [PrintStream](#) definida en el paquete java.io. La clase `PrintStream` es un `OutputStream` muy sencillo de utilizar. Simplemente llama a uno de sus métodos `print()`, `println()`, o `write()` para escribir varios tipos de datos en el canal.

`PrintStream` pertenece al conjunto de canales conocidos como canales filtrados que se cubren [más adelante en esta lección](#).

Similarmente, este programa de ejemplo está estructurado utilizando `System.in.read()` para leer los caracteres tecleados por el usuario.

```
class Count {
    public static void main(String[] args)
        throws java.io.IOException
    {
        int count = 0;

        while (System.in.read() != -1)
            count++;
        System.out.println("La entrada tiene " + count + " caracteres.");
    }
}
```

`System.in` se refiere a un canal de entrada manejado por la clase `System` que implementa el canal de entrada estandar. `System.in` es un objeto [InputStream](#). `InputStream` es una clase abstracta definida en el paquete java.io que define el comportamiento de los canales secuenciales de entrada de Java. Todos los canales de entrada definidos en el paquete java.io son subclases de `InputStream`. `InputStream` define un interface de programación para los canales de entrada que incluye métodos para leer datos desde el canal, marcar una posición dentro del canal, saltar a una marca, y cerrar el canal.

Como has visto, ya estás familiarizado con algunos canales de I/O del paquete java.io. El resto de esta lección introduce los canales del paquete java.io (incluyendo los canales mencionados en esta página: `PrintStream`, `OutputStream` e `InputStream`) y muestra cómo utilizarlos.

Introducción a los Canales de I/O

El paquete `java.io` contiene dos clases, `InputStream` y `OutputStream`, de las que derivan la mayoría de las clases de este paquete.

La clase `InputStream` es una superclase abstracta que proporciona un interface de programación mínimo y una implementación parcial del canal de entrada. La clase `InputStream` define métodos para leer bytes o arrays de bytes, marcar posiciones en el canal, saltar bytes de la entrada, conocer el número de bytes disponibles para ser leídos, y resetear la posición actual dentro del canal. Un canal de entrada se abre automáticamente cuando se crea. Se puede cerrar un canal explícitamente con el método `close()`, o puede dejarse que se cierre implícitamente cuando se recolecta la basura, lo que ocurre cuando el objeto deja de referenciarse.

La clase `OutputStream` es una superclase abstracta que proporciona un interface de programación mínimo y una implementación parcial de los canales de salida. Un canal de salida se abre automáticamente cuando se crea. Se puede cerrar un canal explícitamente con el método `close()`, o se puede dejar que se cierre implícitamente cuando se recolecta la basura, lo que ocurre cuando el objeto deja de referenciarse.

El paquete `java.io` contiene muchas subclases de `InputStream` y `OutputStream` que implementan funciones específicas de entrada y salida. Por ejemplo, `FileInputStream` y `FileOutputStream` son canales de de entrada y salida que operan con ficheros en el sistema de ficheros nativo.

Canales Simples de I/O

Lo siguiente es una introducción a las clases no abstractas que descienden directamente desde `InputStream` y `OutputStream`:

`FileInputStream` y `FileOutputStream`

Leen o escriben datos en un fichero del sistema de ficheros nativo.

`PipedInputStream` y `PipedOutputStream`

Implementan los componentes de entrada y salida de una tubería. Las tuberías se utilizan para canalizar la salida de un programa hacia la entrada de otro. Un `PipedInputStream` debe ser conectado a un `PipedOutputStream` y un `PipedOutputStream` debe ser conectado a un `PipedInputStream`.

`ByteArrayInputStream` y `ByteArrayOutputStream`

Leen o escriben datos en un array de la memoria.

`SequenceInputStream`

Concatena varios canales de entrada dentro de un sólo canal de entrada.

`StringBufferInputStream`

Permite a los programas leer desde un StringBuffer como si fuera un canal de entrada.

Canales Filtrados

FilterInputStream y FilterOutputStream son subclases de InputStream y OutputStream, respectivamente, y también son clases abstractas. Estas clases definen el interface para los canales filtrados que procesan los datos que están siendo leído o escritos. Por ejemplo, los canales filtrados BufferedInputStream y BufferedOutputStream almacenan datos mientras los leen o escriben para aumentar su velocidad.

DataInputStream y DataOutputStream

Lee o escribe datos primitivos de Java en un máquina independiente del formato.

BufferedInputStream y BufferedOutputStream

Almacena datos mientras los lee o escribe para reducir el número de accesos requeridos a la fuente original. Los canales con buffer son más eficientes que los canales similares sin buffer.

LineNumberInputStream

Tiene en cuenta los números de línea mientras lee.

PushbackInputStream

Un canal de entrada con un buffer de un byte hacia atrás. Algunas veces cuando se leen bytes desde un canal es útil chequear el siguiente carácter para poder decir lo que hacer luego. Si se chequea un carácter del canal, se necesitará volver a ponerlo en su sitio para que pueda ser leído y procesado normalmente.

PrintStream

Un canal de salida con los métodos de impresión convenientes.

Y el Resto...

Además de la clases streams, el paquete java.io contiene estas otras clases:

File

Representa un fichero en el sistema de ficheros nativo. Se puede crear un objeto File para un fichero en el sistema de ficheros nativo y luego pedirle al objeto la información que se necesite sobre el fichero (como su path completo).

FileDescriptor

Representa un manejador de fichero (o descriptor) para abrir un fichero o una conexión.

RandomAccessFile

Representa un fichero de acceso aleatorio.

StreamTokenizer

Divide el contenido de un canal en Token (partes). Los Tokens son la unidad más pequeña reconocida por el algoritmo de análisis de texto (como son palabras, símbolos, etc...). Un Objeto StreamTokenizer puede ser utilizado para analizar cualquier fichero de texto. Por ejemplo, podrías utilizarlo para dividir un fichero fuente de Java en nombres de variables, etc.. o un fichero HTML en etiquetas de HTML.

Y finalmente el paquete java.io define tres interfaces:

DataInput y DataOutput

Estos dos interfaces describen canales que pueden leer o escribir datos de tipos primitivos de Java en máquinas independientes del formato. DataInputStream, DataOutputStream, y RandomAccessFile implementan estos interfaces.

FilenameFilter

El método list() de la clase File utiliza un objeto FilenameFilter para determinar los ficheros a listar en un directorio. Este interface acepta ficheros basándose en sus nombres. Se podría utilizar FilenameFilter para implementar una sencilla expresión de búsqueda al estilo de los comodines como fichero.* , etc...

Utilizar Canales para Implementar Tuberías

El paquete `java.io` contiene dos clases, [PipedInputStream](#) y [PipedOutputStream](#), que implementan los componentes de entrada y salida para una tubería. Las tuberías se utilizan para canalizar la salida de un programa (o un thread) hacia la entrada de otro.

Los canales de entrada y salida para tuberías son convenientes para métodos que producen una salida para ser utilizada como entrada de otro método. Por ejemplo, supongamos que estamos escribiendo una clase que implementa varias utilidades de texto, como la ordenación y la inversión del texto. Sería bonito que la salida de uno de esos métodos pudiera utilizarse como la entrada de otro. Así podríamos concatenar una serie de esos métodos para realizar alguna función. La tubería utilizada aquí utiliza los métodos `reverse`, `sort` y `reverse` sobre una lista de palabras para ordenarla de forma rítmica:

Sin los canales de tuberías, tendríamos que crear un fichero temporal entre cada uno de los pasos. Echemos un vistazo al programa que implementa los métodos `reverse` y `sort` descritos arriba utilizando los canales de tuberías, y luego utiliza estas métodos para mostrar la tubería anterior para generar una lista de palabras rítmicas.

Primero, la clase [RhymingWords](#) contiene tres métodos: `main()`, `reverse()`, y `sort()`. El método `main()` proporciona el código para el programa principal, el cual abre un fichero de entrada, utiliza los otros dos métodos para invertir, ordenar y volver a invertir las palabras del fichero de entrada, y escribe la salida en el canal de salida estándar.

`reverse()` y `sort()` están diseñados para ser utilizados en una tubería. Estos dos métodos leen datos desde un `InputStream`, los procesan (invirtiendo las cadenas u ordenandolas), y producen un `PipedInputStream` adecuado para que otro método pueda leerlo. Echemos un vistazo en más detalle al método `reverse()`; `sort` es muy similar al anterior, por eso no merece la pena explicarlo.

```
public static InputStream reverse(InputStream source) {
    PipedOutputStream pos = null;
    PipedInputStream pis = null;

    try {
        DataInputStream dis = new DataInputStream(source);
        String input;

        pos = new PipedOutputStream();
        pis = new PipedInputStream(pos);
        PrintStream ps = new PrintStream(pos);

        new WriteReversedThread(ps, dis).start();
```

```

    } catch (Exception e) {
        System.out.println("RhymingWords reverse: " + e);
    }
    return pis;
}

```

El método `reverse()` toma un `InputStream` llamado `source` que contiene la lista de cadenas a invertir. `reverse()` proyecta un `DataInputStream` sobre el `InputStream source` para que pueda utilizar el método `readLine()` de `DataInputStream` para leer una a una las líneas del fichero. (`DataInputStream` es un canal filtrado que debe ser añadido a un `InputStream` (o proyectado sobre) cuyos datos se deben filtrar mientras son leídos. [Trabajar con Canales Filtrados](#) explica más cosas sobre esto.)

Luego `reverse()` crea un `PipedOutputStream` y le conecta un `PipedInputStream`. Recuerda que un `PipedOutputStream` debe estar conectado a un `PipedInputStream`. Después, `reverse()` proyecta un `PrintStream` sobre el `PipedOutputStream` para que pueda utilizar el método `println()` de `PrintStream` para escribir las cadenas en el `PipedOutputStream`.

Ahora `reverse()` crea un objeto thread [WriteReversedThread](#), que cuelga de dos canales -- el `PrintStream` añadido a `PipedOutputStream` y el `DataInputStream` añadido a `source`-- y lo arranca. El objeto `WriteReversedThread` lee las palabras desde el `DataInputStream`, las invierte y escribe la salida en el `PrintStream` (por lo tanto, escribe la salida en una tubería). El objeto thread permite que cada final de la tubería se ejecute independientemente y evita que el método `main()` se bloquee si una de las tuberías se bloquea mientras espera a que se complete una llamada a I/O.

Aquí tienes el método `run` de `WriteReversedThread`:

```

public void run() {
    if (ps != null && dis != null) {
        try {
            String input;
            while ((input = dis.readLine()) != null) {
                ps.println(reverseIt(input));
                ps.flush();
            }
            ps.close();
        } catch (IOException e) {
            System.out.println("WriteReversedThread run: " + e);
        }
    }
}

```

Como el `PipedOutputStream` está conectado al `PipedInputStream`, todos los datos escritos en el `PipedOutputStream` caen dentro del `PipedInputStream`. Los datos

pueden ser leídos desde `PipedInputStream` por cualquier otro programa o thread. `reverse()` devuelve el `PipedInputStream` para que lo utilice el programa llamador.

El método `sort()` sigue el siguiente patrón:

- Abre un canal de salida de tubería.
- Conecta un canal de entrada de tubería al canal de entrada.
- Utiliza un objeto [SortThread](#), para leer la entrada de la tubería y escribirla en el canal de salida, que es el canal de entrada para algún otro.
- Cuelga el canal de salida, ahora lleno, hacia algún otro para que lo lea.

Las llamadas a los métodos `reverse()` y `sort()` pueden situarse en cascada para que la salida de un método pueda ser la entrada del siguiente método. De hecho, el método `main()` lo hace. Realiza llamadas en cascada a `reverse()`, `sort()`, y luego a `reverse()` para generar una lista de palabras rítmicas:

```
InputStream rhymedWords = reverse(sort(reverse(words)));
```

Cuando ejecutes `RhymingWords` con [este fichero de texto](#) verás la siguiente salida:

```
Java
interface
image
language
communicate
integrate
native
string
network
stream
program
application
animation
exception
primer
container
user
graphics
threads
tools
class
bolts
nuts
object
applet
environment
development
argument
component
```

input
output
anatomy
security

Si miras detenidamente puedes ver que las palabras rítmicas como environment, development, argument, y component están agrupadas juntas.

Ozito

Utilizar Canales para Leer y Escribir Ficheros

Los canales de Ficheros son quizás los canales más fáciles de entender. Las clases [FileInputStream](#) y [FileOutputStream](#) representan una canal de entrada (o salida) sobre un fichero que reside en el sistema de ficheros nativo. Se puede crear un canal de fichero desde un nombre de fichero, un objeto [File](#) o un objeto [FileDescriptor](#). Utiliza los canales de ficheros para leer o escribir datos de un fichero del sistema de ficheros.

Este pequeño ejemplo utiliza los canales de ficheros para copiar el contenido de un fichero en otro:

```
import java.io.*;

class FileStreamsTest {
    public static void main(String[] args) {
        try {
            File inputFile = new File("farrago.txt");
            File outputFile = new File("outagain.txt");

            FileInputStream fis = new FileInputStream(inputFile);
            FileOutputStream fos = new FileOutputStream(outputFile);
            int c;

            while ((c = fis.read()) != -1) {
                fos.write(c);
            }

            fis.close();
            fos.close();
        } catch (FileNotFoundException e) {
            System.err.println("FileStreamsTest: " + e);
        } catch (IOException e) {
            System.err.println("FileStreamsTest: " + e);
        }
    }
}
```

Aquí tiene el contenido del fichero de entrada [farrago.txt](#):

```
So she went into the garden to cut a cabbage-leaf, to
make an apple-pie; and at the same time a great
she-bear, coming up the street, pops its head into the
shop. 'What! no soap?' So he died, and she very
imprudently married the barber; and there were
present the Picninnies, and the Joblillies, and the
Garyalies, and the grand Panjandrum himself, with the
```


little round button at top, and they all fell to playing the game of catch as catch can, till the gun powder ran out at the heels of their boots.

Samuel Foote 1720-1777

Este programa crea un FileInputStream desde un objeto File con este código:

```
File inputFile = new File("farrago.txt");
FileInputStream fis = new FileInputStream(inputFile);
```

Observa el uso del objeto File inputFile en el constructor. inputFile representa el fichero llamado farrago.txt, del sistema de ficheros nativo. Este programa sólo utiliza inputFile para crear un FileInputStream sobre farrago.txt. Sin embargo, el programa podría utilizar inputFile para obtener información sobre farrago.txt como su path completo, por ejemplo.

Ozito

Utilizar los Canales para Leer o Escribir Posiciones de Memoria

Se utiliza [ByteArrayInputStream](#) y [ByteArrayOutputStream](#) para escribir datos de 8 bits. Se pueden crear estos canales sobre un array de bytes existente y luego se pueden utilizar los métodos `read()` y `write()` para leer y escribir datos en el array de memoria.

Se utiliza `StringBufferInputStream` para leer datos desde un `StringBuffer`. Se puede crear un `StringBufferInputStream` sobre un objeto `StringBuffer` existente y luego utilizar el método `read()` para leer desde `StringBuffer` como si residiera en memoria. Esta canal es similar a `ByteArrayInputStream` que lee datos de 8 bits desde un array en la memoria, pero `StringBufferInputStream` lee datos de 16 bits en formato Unicode desde un buffer de string en la memoria. El paquete `java.io` no tiene ningún canal de salida compañero de `StringBufferInputStream` -- en su lugar puede utilizar directamente `StringBuffer`.

Utilizar Canales para Concatenar Ficheros

El canal [SequenceInputStream](#) crea un sólo canal de entrada para varias fuentes. Este programa de ejemplo, [Concatenate](#), utiliza SequenceInputStream para implementar una utilidad de concatenación que une ficheros secuencialmente, en el orden en que se han listado en la línea de comandos.

Esta es la clase controladora de la utilidad Concatenate:

```
import java.io.*;

class Concatenate {
    public static void main(String[] args) {
        ListOfFiles mylist = new ListOfFiles(args);

        try {
            SequenceInputStream s = new SequenceInputStream(mylist);
            int c;

            while ((c = s.read()) != -1) {
                System.out.write(c);
            }

            s.close();
        } catch (IOException e) {
            System.err.println("Concatenate: " + e);
        }
    }
}
```

Lo primero que hace esta clase es crear un objeto ListOfFiles llamado mylist que es inicializado desde los argumentos de la línea de comandos introducidos por el usuario. Los argumentos de la línea de comandos son una lista de ficheros para ser concatenados. mylist es utilizada para inicializar el SequenceInputStream el cual utiliza mylist para obtener un nuevo InputStream para cada fichero listado por el usuario. [ListOfFiles.java](#)

```
import java.util.*;
import java.io.*;

class ListOfFiles implements Enumeration {

    String[] listOfFiles;
    int current = 0;

    ListOfFiles(String[] listOfFiles) {
        this.listOfFiles = listOfFiles;
    }
}
```

```

    }

    public boolean hasMoreElements() {
        if (current < listOfFiles.length)
            return true;
        else
            return false;
    }

    public Object nextElement() {
        InputStream is = null;

        if (!hasMoreElements())
            throw new NoSuchElementException("No more files.");
        else {
            try {
                String nextElement = listOfFiles[current];
                current++;
                is = new FileInputStream(nextElement);
            } catch (FileNotFoundException e) {
                System.out.println("ListOfFiles: " + e);
            }
        }
        return is;
    }
}

```

ListOfFiles implementa el interface [Enumeration](#). Verás cómo esto entra en juego cuando pasemos por el resto del programa.

Después de que el método main() cree el SequenceInputStream, lo lee un byte cada vez. Cuando el SequenceInputStream necesite un InputStream para la nueva fuente (como para el primer byte leído o cuando se alcance el final del canal de entrada actual), llama a nextElement() en el objeto Enumeration para obtener el nuevo InputStream. ListOfFiles crea objetos FileInputStream perezosamente, lo que significa que si SequenceInputStream llama a nextElement(), ListOfFiles abre un FileInputStream con el siguiente fichero de la lista y devuelve el canal. Cuando el ListOfFiles alcanza el final de los ficheros (no hay más elementos), nextElement() devuelve nulo, y la llamada al método read() de SequenceInputStream devuelve -1 para indicar el final de la entrada.

La concatenación simplemente copia todos los datos leídos desde SequenceInputStream en la salida estandar.

Prueba esto: intenta concatenar los ficheros [farrago.txt](#) y [words.txt](#) que han sido utilizados como ficheros de entrada en esta lección.

Trabajar con Canales Filtrados

Se puede añadir un canal filtrado a otro canal para filtrar los datos que se se están leyendo o escribiendo en el canal original. El paquete `java.io` contiene estos canales filtrados que son subclases de `FilterInputStream` o de `FilterOutputStream`:

- `DataInputStream` y `DataOutputStream`
- `BufferedInputStream` y `BufferedOutputStream`
- `LineNumberInputStream`
- `PushbackInputStream`
- `PrintStream` (este es un canal de salida)

Esta sección muestra cómo utilizar los canales filtrados a través de un ejemplo que utiliza `DataInputStream` y `DataOutputStream`. Además, esta sección muestra cómo escribir nuestros propios canales filtrados.

Utilizar Canales Filtrados

Para utilizar un canal filtrado de entrada (o salida), se añade el canal filtrado a otro canal de entrada (o salida). Se puede añadir un canal filtrado a otro canal cuando se crea. Por ejemplo, se puede añadir un `DataInputStream` al canal de entrada estandar de la siguiente forma:

```
DataInputStream dis = new DataInputStream(System.in.read());
String input;
```

```
while ((input = dis.readLine()) != null) {
    . . . // Hacer algo interesante aquí
}
```

Podrías hacer esto para poder utilizar los métodos más convenientes `readXXX()`, como `readLine()`, implementado por `DataInputStream`.

Utilizar `DataInputStream` y `DataOutputStream`

Esta página proporciona una explicación de un ejemplo de utilización de `DataInputStream` y `DataOutputStream`, dos canales filtrados que pueden leer y escribir datos de tipos primitivos de Java.

Escribir Nuestros Propios Canales Filtrados

La mayoría de los programadores encuentran la necesidad de implementar sus propios canales que filtren o procesen los datos que se están leyendo o escribiendo desde un canal. Algunas veces el procesamiento es independiente del formado de los datos, como un contador de los ítems de una cadena, y otras veces el procesamiento está relacionado directamente con los propios datos o el formato de los

datos, como leer o escribir los datos contenidos en filas y columnas. Frecuentemente, estos programadores subclasifican `FilterOutputStream` y `FileInputStream` para conseguir sus objetivos.

Utilizar DataInputStream y DataOutputStream

Esta página muestra cómo utilizar las clases `DataInputStream` y `DataOutputStream` del paquete `java.io`. Utiliza un ejemplo, [DataIOTest](#), que lee y escribe datos tabulados (facturando productos Java). Los datos tabulados están formateados en columnas, donde cada columna está separada de la siguiente por un Tab. Las columnas contienen los precios de venta, el número de unidades pedidas, y una descripción del producto, de esta forma:

```
19.99    12      Java T-shirt
9.99     8       Java Mug
```

`DataOutputStream`, al igual que otros canales de salida filtrados, debe ser añadido a algún `OutputStream`. En este caso, se añade a un `FileOutputStream` que se ha seleccionado para escribir en un fichero llamado `invoice1.txt`.

```
DataOutputStream dos = new DataOutputStream(
    new FileOutputStream("invoice1.txt"));
```

Luego, `DataIOTest` utiliza el método especializado `writeXXX()` de `DataOutputStream` que escribe los datos de la factura (que están contenidos dentro de arrays en el programa) de acuerdo con el tipo de dato que se está escribiendo:

```
for (int i = 0; i < prices.length; i++) {
    dos.writeDouble(prices[i]);
    dos.writeChar('\t');
    dos.writeInt(units[i]);
    dos.writeChar('\t');
    dos.writeChars(descs[i]);
    dos.writeChar('\n');
}
dos.close();
```

Observa que este código cierra el canal de salida cuando termina.

Luego, `DataIOTest` abre un `DataInputStream` con el fichero que acaba de escribir:

```
DataInputStream dis = new DataInputStream(
    new FileInputStream("invoice1.txt"));
```

`DataInputStream`, también debe ser añadido a algún otro `InputStream`. En este caso, se ha añadido a un `FileInputStream` seleccionado para leer el fichero que se acaba de escribir `--invoice1.txt`. `DataIOTest` lee de nuevo los datos utilizando los métodos especializados `readXXX()` de `DataInputStream`.

```
try {
    while (true) {
        price = dis.readDouble();
        dis.readChar();          // Elimina el tab
        unit = dis.readInt();
        dis.readChar();          // Elimina el tab
        desc = dis.readLine();
        System.out.println("Ha pedido " + unit + " unidades de " + desc + " a " +
price);
        total = total + unit * price;
    }
} catch (EOFException e) {
}
System.out.println("Para un TOTAL de: $" + total);
```



```
dis.close();
```

Cuando se han leído todos los datos, DataIOTest muestra una sentencia que resume el pedido y cantidad total, y cierra el canal.

Observa el bucle que utiliza DataIOTest para leer los datos desde el DataInputStream. Normalmente, para la lectura verás un bucle como este:

```
while ((input = dis.readLine()) != null) {  
    . . .  
}
```

El método `readLine()` devuelve un valor, `null`, que indica que se ha alcanzado el final del fichero. Muchos de los métodos `readXXX()` de `DataInputStream` no pueden hacer esto porque cualquier valor devuelto para indicar el final del fichero también podría ser interpretado como un valor legítimo leído desde el canal. Por ejemplo, supongamos que has utilizado `-1` para indicar el final del fichero. Bien, no puedes hacerlo, porque `-1` es un valor legítimo que puede ser leído desde la entrada del canal utilizando `readDouble()`, `readInt()`, o cualquiera de los otros métodos que leen números. Entonces los métodos `readXXX()` de `DataInputStream` lanzan una `EOFException`. Cuando ocurre una `EOFException` el bucle `while(true)` termina.

Cuando se ejecute el programa `DataIoTest` verás la siguiente salida:

```
Ha pedido 12 unidades de Camiseta Java a $19.99  
Ha pedido 8 unidades de Mug Java a $9.99  
Ha pedido 13 unidades de Muñecos Duke a $15.99  
He pedido 29 unidades de Pin Java a $3.99  
Para un TOTAL de: $892.88
```

Ozito

Escribir Nuestros Propios Canales Filtrados

Lo siguiente es una lista de pasos a seguir cuando se escriban canales de I/O filtrados propios:

- Crear una subclase de `FilterInputStream` y `FilterOutputStream`. Los canales de entrada y salida vienen normalmente en parejas, entonces podríamos tener que crear dos versiones de los canales filtrados una de entrada y otra de salida.
- Sobreescribir los métodos `read()` y `write()`.
- Sobreescribir todos los métodos que pudiera necesitar.
- Asegurarse de que los canales de entrada y salida trabajan conjuntamente.

Esta página le muestra cómo implementar sus propios canales filtrados a través de un ejemplo que impletan un par de canales de entrada y salida.

Los dos canales utilizan una clase `Checksum` para calcular la suma de los datos escritos o leídos en un canal. La suma puede ser utilizada para determinar si los datos leídos por la entrada corresponden con los escritos en la salida.

Este programa lo componen cuatro clases y un interface:

- Las subclases para los canales filtrados de entrada y salida -- `CheckedOutputStream` y `CheckedInputStream`.
- El interface `Checksum` y la clase `Adler32` calculan la suma de los canales.
- La clase `CheckedIOClass` define el método principal del programa.

La Clase `CheckedOutputStream`

La clase [CheckedOutputStream](#) es una subclase de `FilterOutputStream` que calcula la suma de los datos que se están escribiendo en un canal. Cuando se cree un `CheckedOutputStream` se debe utilizar su único constructor:

```
public CheckedExceptionStream(OutputStream out, Checksum cksum) {
    super(out);
    this.cksum = cksum;
}
```

Este constructor toma dos argumentos: un `OutputStream` y un `Checksum`. El argumento `OutputStream` es el canal de salida que `CheckedOutputStream` debería filtrar. El objeto `Checksum` es un objeto que puede calcular una suma. `CheckedOutputStream` se inicializa a sí mismo llamando al constructor de su superclase e inicializa una variable privada, `cksum`, con el objeto `Checksum`. El `CheckedOutputStream` utiliza `cksum` que actualiza la suma en el momento en que se están escribiendo los datos en el canal.

`CheckedOutputStream` necesita sobreescribir los métodos `write()` de `FilterOutputStream` para que cada vez que se llame a un método `write()` la suma se actualice. `FilterOutputStream` define tres versiones del método `write()`:

1. `write(int i)`
2. `write(byte[] b)`
3. `write(byte[] b, int offset, int length)`

`CheckedOutputStream` sobreescribe los tres métodos:

```
public void write(int b) throws IOException {
    out.write(b);
    cksum.update(b);
}

public void write(byte[] b) throws IOException {
    out.write(b, 0, b.length);
    cksum.update(b, 0, b.length);
}
```

```

public void write(byte[] b, int off, int len) throws IOException {
    out.write(b, off, len);
    cksum.update(b, off, len);
}

```

La implementación de estos tres métodos `write()` es sencilla: escriben los datos en el canal de salida al que este canal filtrado ha sido añadido, y luego actualizan la suma.

La Clase `CheckedInputStream`

La clase [CheckedInputStream](#) es muy similar a la clase `CheckedOutputStream`.

`CheckedInputStream` es una subclase de `FilterInputStream` que calcula la suma de los datos que están siendo leídos desde un canal. Cuando se cree un `CheckedInputStream`, debe utilizar su único constructor:

```

public CheckedInputStream(InputStream in, Checksum cksum) {
    super(in);
    this.cksum = cksum;
}

```

Este constructor es similar al constructor de `CheckedOutputStream`.

Al igual que `CheckedOutputStream` se necesitará sobrescribir los métodos `wite()` de `FilterOutputStream`, `CheckedInputStream` necesita sobrescribir los métodos `read()` de `FilterInputStream`, para que cada vez que se llame a un método `read()` se actualice la suma. `FilterInputStream` también define tres versiones del método `read()` y `CheckedInputStream` los sobrescribe:

```

public int read() throws IOException {
    int b = in.read();
    if (b != -1) {
        cksum.update(b);
    }
    return b;
}

```

```

public int read(byte[] b) throws IOException {
    int len;
    len = in.read(b, 0, b.length);
    if (len != -1) {
        cksum.update(b, 0, b.length);
    }
    return len;
}

```

```

public int read(byte[] b, int off, int len) throws IOException {
    len = in.read(b, off, len);
    if (len != -1) {
        cksum.update(b, off, len);
    }
    return len;
}

```

La implementación de estos tres métodos `read()` es sencilla: leen los datos del canal de entrada al que este; canal filtrado ha sido añadido, y luego si se ha leído algún dato, actualizan la suma.

El Interface `Checksum` y la Clase `Adler32`

El interface [Checksum](#) define cuatro métodos para sumar objetos, estos métodos resetean, actualizan y devuelve el valor de la suma. Se podría escribir una clase `Checksum` que calcule la suma de un tipo de dato específico como la suma de CRC-32. Observa que la herencia en la

suma es la noción de estado. El objeto checksum no sólo calcula una suma de una forma. En su lugar, la suma es actualizada cada vez que se lee o escribe información en el canal sobre el que este objeto está calculando la suma. Si quieres reutilizar un objeto Checksum, debes resetearlo.

Para este ejemplo, implementamos el objeto checksum [Adler32](#), que es casi una versión de un objeto checksum CRC- pero que puede ser calculado mucho más rápidamente.

Un Programa Principal para Probarlo

La última clase de este ejemplo, [CheckedIOTest](#), contiene el método main() para el programa:

```
import java.io.*;

class CheckedIOTest {
    public static void main(String[] args) {

        Adler32 inChecker = new Adler32();
        Adler32 outChecker = new Adler32();
        CheckedInputStream cis = null;
        CheckedOutputStream cos = null;

        try {
            cis = new CheckedInputStream(new FileInputStream("farrago.txt"),
inChecker);
            cos = new CheckedOutputStream(new FileOutputStream("outagain.txt"),
outChecker);
        } catch (FileNotFoundException e) {
            System.err.println("CheckedIOTest: " + e);
            System.exit(-1);
        } catch (IOException e) {
            System.err.println("CheckedIOTest: " + e);
            System.exit(-1);
        }

        try {
            int c;

            while ((c = cis.read()) != -1) {
                cos.write(c);
            }

            System.out.println("Suma del Canal de Entrada: " + inChecker.getValue());
            System.out.println("Suma del Canal de Salida: " + outChecker.getValue());

            cis.close();
            cos.close();
        } catch (IOException e) {
            System.err.println("CheckedIOTest: " + e);
        }
    }
}
```

El método main() crea dos objetos checksum Adler32, uno para CheckedOutputStream y otro para CheckedInputStream. El ejemplo requiere dos objetos checksum porque se actualizan durante las llamadas a read() y write() y estas llamadas ocurren de forma concurrente.

Luego, main() abre un CheckedInputStream sobre un pequeño fichero de texto [farrago.txt](#), y un CheckedOutputStream sobre un fichero de salida llamado outagain.txt (que no existirá hasta que se ejecute el programa por primera vez).

El método `main()` lee el texto desde `CheckedInputStream` y lo copia en `CeckedOutputStream`. Los métodos `read()` y `write()` utilizar el objeto checksum `Adler32` para calcular la suma durante la lectura y escritura. Después de que se haya leído completamente el fichero de entrada (y consecuentemente, se haya terminado de escribir el fichero de salida), el programa imprime la suma de los dos canales (que deberían corresponder) y luego cierra los dos canales.

Cuando se ejecute `CheckedIOTest` deberías ver esta salida:

```
Suma del Canal de Entrada: 736868089
```

```
Suma del sanal de Salida: 736868089
```

Filtrar Ficheros de Acceso Aleatorio

Los canales filtrados del paquete `java.io` descienden de `InputStream` o `OutputStream` que implementan los ficheros de acceso secuencial. Entonces, si se subclasifica `FilterInputStream` o `FilterOutputStream` los canales filtrados también serán ficheros de acceso secuencial. [Escribir filtros para ficheros de acceso aleatorio](#) más adelante es está lección le muestra cómo re-escribir este ejemplo para que trabaje en un `RandomAccessFile` así como con un `DataInputStream` o un `DataOutputStream`.

Trabajar con Ficheros de Acceso Aleatorio

Los canales de entrada y salida que has aprendido hasta ahora en esta lección han sido canales de acceso secuencial -- canales cuyo contenido debe ser leído o escrito secuencialmente. Esto es increíblemente útil, los ficheros de acceso secuencial son consecuencia de un medio secuencial como una cinta magnética. Por otro lado, los ficheros de acceso aleatorio, permiten un acceso no secuencial, o aleatorio al contenido de un fichero.

Veamos por qué podríamos necesitar ficheros de acceso aleatorio. Consideremos el formato de archivo conocido como "zip". Los archivos Zip contienen ficheros que normalmente están comprimidos para ahorrar espacio. Estos archivos también contienen una "entrada de directorio" que indica donde comienzan los distintos ficheros contenidos dentro del archibo Zip:

Ahora supongamos que queremos extraer un fichero específico de un archivo Zip. Si utilizáramos un canal de acceso secuencial, tendría que hacer lo siguiente:

- Abrir el archivo Zip.
- Buscar a través del archivo Zip hasta que se localice el fichero que quiere extraer.
- Extraer el Fichero.
- Cerar el archivo Zip.

Como mínimo, utilizando este argumento, habremos tenido que leer medio archivo Zip antes de encontrar el fichero que quiere extraer. Se puede extraer el mismo fichero de un archivo Zip de una forma más eficiente utilizando la característica de búsqueda de un fichero de acceso aleatorio:

- Abrir el archvivo Zip.
- Buscar en la entrada de directorio y localizar la entrada para el fichero que quiere extraer.
- Buscar (hacia atrás) dentro del archivo Zip hasta la posición del fichero a extraer.
- Extraer el fichero.
- Cerrar el archivo Zip.

Este algoritmo es mucho más eficiente porque sólo lee la entrada de directorio y el fichero que se quiere extraer.

La clase `RandomAccessFile` del paquete `java.io` implementa un fichero de acceso aleatorio.

Utilizar Ficheros de Acceso Aleatorio

Al contrario que las clases de canales de I/O de `java.io`, `RandomAccessFile` se utiliza tanto para leer como para escribir ficheros.

Se puede crear el `RandomAccessFile` con diferentes argumentos dependiendo de lo que se intente hacer, leer o escribir.

Escribir Filtros para Ficheros de Acceso Aleatorio

`RandomAccessFile` está de alguna forma desconectado de los canales de I/O de `java.io` -- no descende de las clases `InputStream` o `OutputStream`. Esto tiene alguna desventaja en que no se pueden aplicar los mismos filtros a `RandomAccessFiles` como se puede hacer con los otros canales. Sin embargo, `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput`, por lo que se podría de hecho diseñar un filtro que trabajando con `DataInput` o `DataOutput` podría trabajar (de alguna forma) con ficheros de acceso secuencial (los únicos que implementan `DataInput` o `DataOutput`) como con cualquier `RandomAccessFile`.

Utilizar Ficheros de Acceso Aleatorio

La clase `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput` y por lo tanto puede utilizarse tanto para leer como para escribir. `RandomAccessFile` es similar a `FileInputStream` y `FileOutputStream` en que se especifica un fichero del sistema de ficheros nativo para abrirlo cuando se crea. Se puede hacer esto con un nombre de fichero o un objeto [File](#). Cuando se crea un `RandomAccessFile`, se debe indicar si sólo se va a leer el fichero o también se va a escribir en él. La siguiente línea de código java crea un `RandomAccessFile` para leer el fichero llamado `farrago.txt`:

```
new RandomAccessFile("farrago.txt", "r");
```

Y esta otra abre el mismo fichero para leer y escribir:

```
new RandomAccessFile("farrago.txt", "rw");
```

Después de haber abierto el fichero, se pueden utilizar los métodos comunes `readXXX()` o `writeXXX()` para realizar la I/O sobre el fichero.

`RandomAccessFile` soporta la noción de puntero de fichero. El puntero de fichero indica la posición actual dentro del fichero. Cuando el fichero se crea por primera vez, el puntero de fichero es cero, indicando el principio del fichero. Las llamadas a los métodos `readXXX()` y `writeXXX()` ajustan el puntero de fichero el número de bytes leídos o escritos.

Además de los métodos normales de I/O que mueven implícitamente el puntero de fichero cuando ocurre una operación, `RandomAccessFile` contiene tres métodos que manipulan explícitamente el puntero de fichero.

`skipBytes()`

Mueve el puntero de fichero hacia adelante el número de bytes especificado.

`seek()`

Posiciona el puntero de fichero justo en el byte especificado.

`getFilePointer()`

Devuelve la posición actual del puntero de fichero.

Escribir Filtros para Ficheros de Acceso Aleatorio

Reescribamos el ejemplo de [Escribir nuestros propios Canales Filtrados](#) para que trabaje con un `RandomAccessFile`. Como `RandomAccessFile` implementa los interfaces `DataInput` y `DataOutput`, un beneficio lateral es que los canales filtrados también trabajan con otros canales `DataInput` y `DataOutput` incluyendo algunos canales de acceso secuencial (como `DataInputStream` y `DataOutputStream`).

El ejemplo [CheckedIOTest](#) de [Escribir nuestros propios Canales Filtrados](#) implementa dos canales filtrados [CheckedInputStream](#) y [CheckedOutputStream](#), que calculan la suma de los datos leídos o escritos en un canal.

En el nuevo ejemplo, [CheckedDataOutput](#) se ha reescrito `CheckedOutputStream`-- calcula la suma de los datos escritos en el canal -- pero opera sobre objetos `DataOutput` en vez de sobre objetos `OutputStream`. De forma similar, [CheckedDataInput](#) modifica `CheckedInputStream` para que ahora trabaje sobre objetos `DataInput` en vez de hacerlo sobre objetos `InputStream`.

CheckedDataOutput contra CheckedOutputStream

Echemos un vistazo a las diferencias entre `CheckedDataOutput` y `CheckedOutputStream`.

La primera diferencia es que `CheckedDataOutput` no descende de `FilterOutputStream`. En su lugar, implementa el interface `DataOutput`.

```
public class CheckedDataOutput implements DataOutput
```

Nota: Para intentar mantener el ejemplo lo más sencillo posible, la clase `CheckedDataOutput` realmente no está declarada para implementar el interface `DataInput`. Esto es así porque este interface implementa demasiados métodos. Sin embargo, la clase `CheckedDataOutput` como está implementada en el ejemplo, si que implementa varios métodos de `DataInput` para ilustrar cómo deberían trabajar.

Luego, `CheckedDataOutput` declara una variable privada para contener el objeto `DataOutput`.

```
private DataOutput out;
```

Este es el objeto donde se escribirán los datos.

El constructor de `CheckedDataOutput` se diferencia del de `CheckedOutputStream` en que el primero crea un objeto `DataOutput` en vez de un `OutputStream`.

```
public CheckedDataOutput(DataOutput out, Checksum cksum) {  
    this.cksum = cksum;  
    this.out = out;  
}
```

Observa que este constructor no llama a `super(out)` como lo hacía el constructor de `CheckedOutputStream`. Esto es así porque `CheckedDataOutput` descende de la clase `Object` en vez de una clase stream.

Estas han sido las únicas modificaciones hechas en `CheckedOutputStream` para crear un filtro que trabaje con objetos `DataOutput`.

CheckedDataInput contra CheckedInputStream

`CheckedDataInput` requiere los mismos cambios que `CheckedDataOutput`:

- `CheckedDataInput` no descende de `FilterInputStream` pero implementa el interface `DataInput` en su lugar.
-

Nota: Para intentar mantener el ejemplo lo más sencillo posible, la clase `CheckedDataInput` realmente no está declarada para implementar el interface `DataInput`. Esto es así porque este interface implementa demasiados métodos. Sin embargo, la clase

CheckedDataInput como está implementada en el ejemplo, si que implementa varios métodos de DataInput para ilustrar cómo deberían trabajar.

- CheckedDataInput declare una variable privada para contener un objeto DataInput.
- El constructor de CheckedDataInput requiere un objeto DataInput en vez de un InputStream.

Además de estos cambios también se han cambiado los métodos read().

CheckedInputStream del ejemplo original implementa dos métodos read(), uno para leer un sólo byte y otro para leer un array de bytes. El interface DataInput tiene métodos que implementan la misma funcionalidad, pero tienen diferentes nombres y diferentes firma de método. Así los métodos read() de la clase CheckedDataInput tienen nuevos nombres y firmas de método:

```
public byte readByte() throws IOException {
    byte b = in.readByte();
    cksum.update(b);
    return b;
}

public void readFully(byte[] b) throws IOException {
    in.readFully(b, 0, b.length);
    cksum.update(b, 0, b.length);
}

public void readFully(byte[] b, int off, int len) throws IOException {
    in.readFully(b, off, len);
    cksum.update(b, off, len);
}
```

Los Programas Principales

Finalmente, este ejemplo tiene dos programas principales para probar los nuevos filtros:

[CheckedDITest](#), que ejecuta los filtros en ficheros de acceso secuencial (objetos DataInputStream and DataOutputStream objects), y [CheckedRAFTest](#), que ejecuta los filtros en ficheros de acceso aleatorio (RandomAccessFiles).

Estos dos programas principales se diferencian sólo en el tipo de objeto que abre los filtros de suma. CheckedDITest crea un DataInputStream y un DataOutputStream y utiliza el filtro sumador sobre ellos. como esto:

```
cis = new CheckedDataInput(new DataInputStream(
    new FileInputStream("farrago.txt")), inChecker);
cos = new CheckedDataOutput(new DataOutputStream(
    new FileOutputStream("outagain.txt")), outChecker);
```

CheckedRAFTest crea crea dos RandomAccessFiles, uno para leer y otro para escribir, y utiliza el filtro sumador sobre ellos:

```
cis = new CheckedDataInput(new RandomAccessFile("farrago.txt", "r"), inChecker);
cos = new CheckedDataOutput(new RandomAccessFile("outagain.txt", "rw"), outChecker);
```

Cuando se ejecute cualquiera de estos programas debería ver la siguiente salida:

```
Suma del canal de Entrada: 736868089
Suma del canal de Salida: 736868089
```

Cambios en el JDK 1.1:

Que afectan a los Steams de I/O

El paquete java.io ha sido ampliado con streams de caracteres. que son como stream de bytes pero que contienen caracteres Unicode de 16-bits , en vez de vytes de 8-bits. Los streams de caracteres hacen sencillo escribir programas que no sean dependientes de una codificación específica de caracteres. y por lo tanto son sencillos de internacionalizar. Casi toda la funcionalidad existente en los streams de bytes también está disponible en los streams de caracteres. [Steams de I/O](#)

La mayoría de los programadores que utilicen JDK 1.1 deben utilizar el nuevo stream de caracteres para hacer I/O. Puedes ver [Cambios en el JDK 1.1: El Paquete java.io](#).

[Tu primer encuentro con la I/O en Java](#)

En el JDK 1.1. la salida de programas de texto deben escribirse mediante un PrintWriter. [Cambios en el JDK 1.1: Salida de Programas de Texto](#).

[Introducción a los Streams de I/O](#)

Se han añadido nuevas clases al paquete java.io para soportar la lectura-escritura de caracteres Unicode de 16-bits. Puedes ver [Cambios en el JDK 1.1: El Paquete java.io](#).

[Utilizar Streams para Leer y Escribir Ficheros](#)

La mayoría de los programadores del JDK 1.1 deberían utilizar las nuevas clases FileReader y FileWriter. Puedes ver [Cambios en el JDK 1.1: I/O de Ficheros](#).

[Utilizar Streams para Leer y Escribir Posiciones de Memoria](#)

La mayoría de los programadores del JDK 1.1 deberían utilizar las nuevas clases de streams de caracteres. Puedes ver [Cambios en el JDK 1.1: I/O de Memoria](#).

[Utilizar Streams para Concatenar Ficheros](#)

El JDK 1.1 no proporciona una alternativa de stream de caracteres para el stream de bytes SequenceInputStream.

[Trabajar con Ficheros Filtrados](#)

La mayoría de los programadores del JDK 1.1 deberían utilizar las nuevas clases de streams de caracteres. Puedes ver [Cambios en el JDK 1.1: Streams Filtrados](#).

[Utilizar DataInputStream y DataOutputStream](#)

El método DataInputStream.readLine ha caducado. Puedes ver [Cambios en el JDK 1.1: Data I/O](#).

[Escribir tus propios Streams Filtrados](#)

Como el ejemplo CheckSum utiliza Bytes, no debería ser modificado para utilizar las nuevas clases de streams de caracteres.

[Escribir Filtros para Ficheros de Acceso Aleatorio](#)

Como el ejemplo CheckSum utiliza Bytes, no debería ser modificado para utilizar las nuevas clases de streams de caracteres.

El Applet "Hola Mundo"

Siguiendo los pasos de esta página, podremos crear y utilizar un applet.

Crear un Fichero Fuente Java

Creamos un fichero llamado [HolaMundo.java](#) con el código Java del siguiente listado:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("HolaMundo!", 50, 25);
    }
}
```

Compilar el Fichero Fuente

Compilar un Fichero Fuente Java Utilizando el JDK: Detalles Específicos de la Plataforma:

UNIX:

```
javac HolaMundo.java
```

DOS shell (Windows 95/NT):

```
javac HolaMundo.java
```

MacOS:

Lleva el icono del fichero HolaMundo.java al icono del Compilador de Java.

Si la compilación tiene éxito, el compilador crea un fichero llamado Holamundo.class.

Si la compilación falla, asegurate de que has tecleado y llamado al programa exactamente como se mostró arriba.

Crear un Fichero HTML que incluya el Applet

Cree un fichero llamado [Hola.html](#) en el mismo directorio que contiene HolaMundo.class. Este fichero HTML debe contener el siguiente texto:

```
<HTML>
<HEAD>
```

```
<TITLE> A Programa Sencillo </TITLE>
</HEAD>
<BODY>
```

Aquí está la salida de mi programa:

```
<APPLET CODE="HolaMundo.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Cargar el Fichero HTML

Carga el fichero HTML en una aplicación que pueda ejecutar applets. Esta aplicación podría ser un navegador compatible con Java u otro programa que permite visualizar applets, como el AppletViewer del JDK. Para cargar el fichero HTML, normalmente sólo se necesita decirle a la aplicación la URL del fichero HTML que se ha creado. Por ejemplo, podríamos introducir algo como esto en el campo de localización de su navegador:

```
file:/paginas/web/HTML/Hola.html
```

Una vez que has completado estos pasos con éxito, podrás ver algo como esto en la ventana del navegador:

Aquí tienes la salida del programa:

La Anatomía de un Applet Java

Ahora que has visto un applet Java, probablemente querrás saber como trabaja. Recuerda que un applet Java es un programa que se adhiere a una serie de convenciones que le permiten ejecutarse dentro de navegadores compatibles con Java.

Aquí tienes el código para el applet "Hola Mundo".

```
import java.applet.Applet;
import java.awt.Graphics;

public class HolaMundo extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hola Mundo!", 50, 25);
    }
}
```

Importar Clases y Paquetes

El código anterior empieza con dos sentencias import. Mediante la importación de clases o paquetes, una clase puede referirse más fácilmente a otras clases en otros paquetes. En el lenguaje Java, los paquetes se utilizan para agrupar clases de forma similar a como las librerías C agrupan funciones.

Definir una Subclase Applet

Cada applet debe definir una subclase de la clase Applet. En el Applet "Hola Mundo", esta subclase se llama HolaMundo. Los applets heredan casi toda su funcionalidad de la clase Applet, desde la comunicación con el navegador hasta la posibilidad de presentar un interface gráfico de usuario (GUI).

Implementar Métodos Applet

El applet Hola Mundo sólo implementa un método, el método paint(). Cada applet debe implementar, al menos, uno de los siguientes métodos: init(), start(), o paint(). Al contrario que las aplicaciones Java, los applet no necesitan implementar el método main().

Ejecutar un Applet

Los applets están diseñados para incluirlos en páginas HTML. Utilizando la etiqueta <APPLET>, se puede especificar (como mínimo) la localización de

la subclase Applet y las dimensiones de la ventana del applet. Cuando el navegador compatible con Java encuentra la etiqueta <APPLET>, reserva un espacio en la pantalla para el applet, carga la subclase Applet dentro del ordenador donde se está ejecutando el navegador y crea un ejemplar de la subclase Applet. Luego el navegador llama a los métodos init() y start() del applet, y éste empieza su ejecución.

Importar Clases y Paquetes

Las dos primeras líneas del siguiente listado importan dos clases utilizadas en el applet: Applet y Graphics.

```
import java.applet.Applet;
import java.awt.Graphics;

public class HolaMundo extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hola Mundo!", 50, 25);
    }
}
```

Si se eliminan las dos primeras líneas, el applet todavía se compila y se ejecuta, pero sólo si se cambia el resto del código de la siguiente forma:

```
public class HolaMundo extends java.applet.Applet {
    public void paint(java.awt.Graphics g) {
        g.drawString("Hola Mundo!", 50, 25);
    }
}
```

Como puedes ver, la importación de las clases Applet y Graphics le permite al programa referirse a ellas sin prefijos. Los prefijos `java.applet.` y `java.awt.` le dicen al compilador los paquetes en los que debería buscar las clases Applet y Graphics. Los dos paquetes, `java.applet` y `java.awt` son parte del corazón del API de Java --API con el que cada programa Java puede contar dentro del entorno Java.

El paquete `java.applet` contiene clases que son esenciales para los applets Java. El paquete `java.awt` contiene las clases más utilizadas en la herramienta de Ventanas Abstractas (AWT) que proporciona el interface gráfico de usuario (GUI) de Java.

Además de importar clases individuales, también se puede importar paquetes completos. Aquí tienes un ejemplo:

```
import java.applet.*;
import java.awt.*;

public class HolaMundo extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hola Mundo!", 50, 25);
    }
}
```

En el lenguaje Java, cada clase es un paquete. Si el código fuente de una clase no tiene al principio la sentencia `package` declarando el paquete en el que se encuentra la clase, la clase está entonces en el paquete por defecto. Casi todas las clases de ejemplo utilizadas en este tutorial se encuentran en el paquete por

defecto.

Dentro de un paquete, todas las clases se pueden referir unas a otras sin prefijos. Por ejemplo, la clase Component de java.awt se refiere a la clase Graphics de java.awt sin prefijos, sin importar la clase Graphics.

Ozito

Definir una Subclase Applet

La primera línea en negrita del siguiente listado empieza un bloque que define la clase HolaMundo.

```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HolaMundo extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("Hola Mundo!", 50, 25);  
    }  
}
```

La palabra clave `extends` indica que `HolaMundo` es una subclase de la clase cuyo nombre viene a continuación: `Applet`.

De la clase `Applet`, los applets heredan un gran cantidad de funcionalidades. Quizás la más importante es la habilidad de responder a las peticiones del navegador. Por ejemplo, cuando un navegador compatible con Java carga una página que contiene un applet, el navegador envía una petición al applet, para que éste se inicialice y empiece su ejecución. Aprenderá más sobre lo que proporciona la clase `Applet` en la lección: [Descripción de un Applet](#)

Un applet no está restringido a definir sólo una clase. Junto con la necesaria subclase `Applet`, un applet puede definir clases de usuario adicionales. Cuando un applet intenta ejecutar una clase, la aplicación busca la clase en el ordenador local. Si la clase no está disponible localmente, la carga desde la posición donde fuera originaria la subclase `Applet`.

Implementar Métodos en un Applet

Las líneas en **negrita** del siguiente listado implementan el método `paint()`.

```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HolaMundo extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("Hola Mundo!", 50, 25);  
    }  
}
```

Todos los applets deben implementar uno o más de estos métodos: `init()`, `start()`, o `paint()`. Aprenderá más sobre estos métodos en la lección [Descripción de un Applet](#)

Junto con los métodos `init()`, `start()`, y `paint()`, un applet puede implementar dos métodos más que el navegador puede llamar cuando ocurre un evento principal (como salir de la página del applet): `stop()` y `destroy()`. Los applets pueden implementar cualquier número de métodos, así como métodos del cliente y métodos que sobrescriben los métodos implementados por la superclase.

Volviendo al código anterior, el objeto `Graphics` pasado dentro del método `paint()` representa el contexto de dibujo en la pantalla del applet. El primer argumento del método `drawString()` es la cadena que se muestra en la pantalla. El segundo argumento son las posiciones (x,y) de la esquina inferior izquierda del texto en la pantalla. Este applet muestra la cadena "Hola Mundo" en la posición (50,25). Las coordenadas de la ventana del applet empiezan en (0,0), que es la esquina superior izquierda de la ventana del applet.

Aprenderás más sobre el dibujo en la pantalla en la página: [Crear un Interface de Usuario](#)

Ejecutar un Applet

Las líneas en **negrita** en el siguiente listado comprenden la etiqueta `<APPLET>` que incluye el applet "Hola World" en una página HTML.

```
<HTML>
<HEAD>
<TITLE> Un Programa Sencillo </TITLE>
</HEAD>
<BODY>
```

Aquí está la salida de mi programa:

```
<APPLET code="HolaMundo.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

La etiqueta `<APPLET>` especifica que el navegador debe cargar la clase cuyo código compilado está en el fichero llamado `HolaMundo.class`. El navegador busca este fichero en el mismo directorio del documento HTML que contiene la etiqueta.

Cuando un navegador encuentra el fichero de la clase, la carga a través de la red, si es necesario, hasta el ordenador donde se está ejecutando el navegador. El navegador crea un ejemplar de la clase. Si se incluye un applet dos veces en una página, el navegador carga la clase una sola vez y crea dos ejemplares de la clase.

Los atributos `WIDTH` y `HEIGHT` son iguales que los de la etiqueta ``: Especifican el tamaño en pixels del área de la pantalla reservado para el applet. Muchos navegadores no permiten que el applet modifique su tamaño más allá de este área. Por ejemplo, cada bit que dibuja el applet "Hola Mundo" en su método `paint()` ocurre dentro de un área de pantalla de 150x25-pixels que reserva la etiqueta `<APPLET>`.

Para obtener más información sobre la etiqueta `<APPLET>`, puede ver [Añadir un Applet a una Página HTML](#)

Descripción de un Applet

Cada applet está implementado mediante una subclase de la clase Applet. La siguiente figura muestra las herencias de la clase Applet. Estas herencias determinan mucho de lo que un applet puede hacer y como lo hace, como verás en las siguientes páginas.

```
java.lang.Object
|
+----java.awt.Component
      |
      +----java.awt.Container
            |
            +----java.awt.Panel
                  |
                  +----java.applet.Applet
```

Un Applet Sencillo

Lo de abajo es el código fuente de un Applet llamado Simple. Este applet muestra una cadena descriptiva donde se encuentra el mayor evento en su vida, cuando el usuario visite por primera vez la página del applet. Las páginas que siguen utilizan el applet Simple y construyen sobre él para ilustrar conceptos que son comunes a muchos Applets.

```
import java.applet.Applet;
import java.awt.Graphics;

public class Simple extends Applet {
    StringBuffer buffer;
    public void init() {
        buffer = new StringBuffer();
        addItem("inicializando... ");
    }

    public void start() {
        addItem("arrancando... ");
    }

    public void stop() {
        addItem("parando... ");
    }

    public void destroy() {
        addItem("preparando para descargar...");
    }
}
```

```

void addItem(String newWord) {
    System.out.println(newWord);
    buffer.append(newWord);
    repaint();
}

public void paint(Graphics g) {
    //Dibuja un Rectangulo alrededor del area del Applet.
    g.drawRect(0, 0, size().width - 1, size().height - 1);

    //Dibuja la cadena actual dentro del Rectangulo.
    g.drawString(buffer.toString(), 5, 15);
}
}

```

El Ciclo de Vida de un Applet

Se puede utilizar el applet Simple para aprender sobre los eventos en cada una de las vidas de un applet.

Métodos para Milestones

La clase Applet proporciona una zona de trabajo para la ejecución del applet, definiendo métodos que el sistema llama cuando ocurre un milestone -- El mayor evento en el ciclo de vida de un applet --. La mayoría de los applets pasan por encima de estos métodos para responder apropiadamente al milestone.

Métodos para dibujar y para manejar eventos

Los Applets heredan los Métodos de dibujo y de manejo de eventos de la clase componente AWT. (AWT significa Abstract Windowing Toolkit ; los applets y las aplicaciones utilizan sus clases para producir Interfaces de usuario.) Dibujar se refiere a algo relacionado con la representación de un applet en la pantalla --dibujar imágenes, presentar componentes de un interface como botones, o utilizar gráficos rudimentarios. Manejo de Eventos se refiere a la detección y procesamiento de las entradas del usuario como pulsaciones del Ratón o del teclado, así como otros eventos abstractos como salvar ficheros o cerrar ventanas.

Métodos para añadir Componentes UI

Los Applets descienden de la clase contendora AWT. Esto significa que están diseñados para contener Componentes --Objetos de un interface

de usuario como botones, etiquetas, listas desplegadas y barras de desplazamiento. Al igual que otros contenedores los applets tienen manejadores para controlar el posicionamiento de los Componentes.

Tareas en un Applet

Como has aprendido en [Tareas de Control](#), una tarea --conocida tambien como un contexto de ejecución o un proceso ligero --es un sencillo flujo de control secuencial dentro de un proceso. Incluso el applet más sencillo se ejecuta en multiples tareas, aunque no siempre es obvio. Muchos applets crean y utilizan sus propias tareas, por eso las realizan correctamente sin afectar el rendimiento de la aplicación donde se encuentran o de otros applets.

Qué puede y qué no puede hacer un Applet

Por razones de seguridad, un applet que se ha cargado a través de la Red tiene muchas restricciones. Una de ellas es que normalmente no puede leer o escribir ficheros en el ordenador donde se está ejecutando. Otra es que un applet no puede hacer conexiones a través de la red excepto hacia el Host del que ha venido. A pesar de estas restricciones los applets pueden hacer algunas cosas que no se esperan. Por ejemplo, un applet puede llamar a los métodos públicos de otro applet que esté en la misma página.

Añadir un Applet a una página HTML

Una vez que has escrito un applet, necesitas añadirlo a una página HTML para que pueda ejecutarse.

El Ciclo de Vida de un Applet

Lo que ves aquí es el resultado del applet Simple.

Cargando el Applet

Arriba puedes ver "inicializando... arrancando...", como el resultado de que el applet está siendo cargado. Cuando un applet se carga, aquí está lo que sucede:

1. Se crea un ejemplar de la clase del Applet (una subclase Applet).
2. El Applet se inicializa a si mismo.
3. El Applet empieza su ejecución.

Abandonando y Volviendo a la Página que contiene el Applet

Cuando el usuario abandona la página-- por ejemplo, para ir a otra página --el applet tiene la opción de pararse. Cuando el usuario retorna a la página, el applet puede empezar de nuevo su ejecución. La misma secuencia ocurre cuando el usuario minimiza y maximiza la ventana que contiene el applet.

Prueba esto: Abandona y luego vuelve a esta página. Veras que se añade "parando..." a la [salida del applet](#), porque el applet tiene la opción de pararse a si mismo. También veras "arrancando...", cuando el applet empiece a ejecutarse de nuevo.

Nota del Navegador: Si estás utilizando el Netscape Navigator 2.0 o 2.01 , podría observar que el applet se inicializa más de una vez. Específicamente, el navegador le pide al applet que se inicialice de nuevo cuando el usuario retorna a la página del applet siguiendo un enlace. El navegador NO se lo pedirá cuando el usuario retorne a la página utilizando el botón Back. Lo último es el comportamiento esperado.

Prueba esto: Minimiza esta ventana y luego abrela de nuevo. Muchos sistemas de ventanas proporcionan un botón en la barra de título para hacer esto. Debería ver "parando..." y "arracando..." en la [salida del applet](#).

Recargar el Applet

Algunos navegadores le permiten al usuario recargar los applets, que consiste en descargar el applet y luego cargarlo otra vez. Antes de descargar el applet, este tiene la opción de pararse el mismo y luego realizar un limpiado final, para que el applet pueda liberar los recursos que ha tomado. Después de esto, el applet se descarga y se carga de

nuevos, como se describe en [Cargar el applet](#).

Nota del Navegador: El Netscape Navigator 2.0 algunas veces recargará el applet si se pulsa Shift mientras hace click en el botón de Reload. Si esto falla, puede intentar vaciar los caches de memoria y disco del Netscape (desde el diálogo Options/Network Preferences) y recargar otra vez.

Saliendo del Navegador

Cuando el usuario sale o abandona el Navegador (o la aplicación que muestre el applet), el applet tiene la opción de pararse a si mismo y hacer una limpieza final antes de salir del navegador.

Sumario

Un applet puede reaccionar a los principales eventos de la siguiente forma:

- Puede inicializarse a si mismo.
- Puede empezar su ejecución.
- Puede parar la ejecución.
- Puede realizar una limpieza final, para preparar la descarga.

La página siguiente describe los cuatros métodos del applet que corresponden con estos cuatro tipos de reacciones.

Métodos de Applets

```
public class Simple extends Applet {  
    . . .  
    public void init() { . . . }  
    public void start() { . . . }  
    public void stop() { . . . }  
    public void destroy() { . . . }  
    . . .  
}
```

El Applet Simple, como cualquier otro, es una subclase de la clase Applet. La clase Simple sobrescribe cuatro métodos de la clase Applet para poder responder a los principales eventos:

init()

Para inicializar el applet cada vez que se carga.

start()

Para iniciar la ejecución del applet, una vez cargado el applet o cuando el usuario vuelve a visitar la página que contiene el applet.

stop()

Para parar la ejecución del applet, cuando el usuario abandona la página o sale del navegador.

destroy()

Realiza una limpieza final para preparar la descarga.

No todos los applets necesitan reescribir estos métodos. Algunos applets muy sencillos no sobrescriben ninguno de ellos, por ejemplo el ["Applet Hola Mundo"](#) no sobrescribe ninguno de estos métodos, ya que no hace nada excepto dibujarse a si mismo. El Applet "Hola Mundo" sólo muestra una cadena una vez, utilizando su método paint(). (El método paint() se describe en la página siguiente.) Sin embargo, la mayoría de los applets, hacen mucho más.

El método init() se utiliza para una inicialización de una sola vez, que no tarda mucho tiempo. En general, el método init() debería contener el código que se pondría normalmente en un constructor. La razón por la que normalmente los applets no tienen constructores es porque un applet no tiene garantizado un entorno completo hasta que se llama a su método init(). Por ejemplo, el método para cargar imágenes en un applet simplemente no funciona dentro del constructor de un applet. Por otro lado, el método init(), es un buen lugar para llamar a los métodos para cargar imágenes, ya que estos métodos retornan rápidamente.

Nota del Navegador: Netscape Navigator 2.0 and 2.0.1 algunas veces llaman al método init() más de una vez después de haber cargado el applet. Puedes ver la [pagina anterior](#) para mas detalles.

Todos los applets que hacen algo despues de la inicializacion (excepto la respuesta a una accion del usuario) deben sobrescribir el metodo `start()`. Este método realiza el trabajo del applet o (dicho de otra forma) arranca una o mas tareas para realizar el trabajo. Aprenderá mas sobre las tareas mas adelante en esta ruta, en [Tareas en un Applet](#). Aprenderá mas sobre el manejo de eventos en representación del usuario en la [página siguiente](#).

La mayoría de los applets que sobrescriben el método `start()` tambien deberian sobrescribir el método `stop()`. Este método deberia parar la ejecucion del applet, para que no gaste recursos del sistema cuando el usuario no esta viendo la página del applet. Por ejemplo, un applet que muestra animaciones debe parar de mostrarlas cuando el usuario no esta mirando.

La mayoría de los applets no necesitan sobrescribit el método `destroy()`, porque su método `stop()` (al que se llama antes del método `destroy()`) hace todo lo necesario para detener la ejecucion del applet. Sin embargo, el método `destroy()` esta disponible para los applets que necesitan liberar recursos adicionales.

Los métodos `init()`, `start()`, `stop()`, y `destroy()` se utilizan muy frecuentemente en este tutorial.

Métodos para Dibujar y Manejar Eventos

```
class Simple extends Applet {  
    . . .  
    public void paint(Graphics g) { . . . }  
    . . .  
}
```

El applet Simple define su apariencia en la pantalla sobrescribiendo el método `paint()`. El método `paint()` es uno de los dos métodos de pantalla que los applets pueden sobrescribir:

`paint()`

El método de dibujo básico. Muchos applets implementan el método `paint()` para mostrar la representación de un applet dentro de la página del navegador.

`update()`

Un método que se puede utilizar junto en el método `paint()` para aumentar el rendimiento de los gráficos.

Los Applets heredan sus métodos `paint()` y `update()` de la clase `Applet`, y esta lo hereda de la clase `Component AWT` (Abstract Window Toolkit).

De la clase `Component`, los applets heredan un grupo de métodos para el manejo de eventos. La clase `Component` define varios métodos (como son `action()` y `mouseDown()`) para manejar eventos particulares, y uno que captura todos los eventos, `handleEvent()`.

Para reaccionar a un evento, un applet debe sobrescribir el método especializado apropiado o el método `handleEvent()`. Por ejemplo, añadiendo el siguiente código al applet Simple applet hace que éste responda a los clicks de ratón.

```
import java.awt.Event;  
.  
.  
.  
public boolean mouseDown(Event event, int x, int y) {  
    addItem("click!... ");  
    return true;  
}
```

Lo que se ve abajo es el resultado del applet. Cuando se hace click dentro de su rectángulo, muestra la palabra "click!...".

Métodos para añadir componentes UI

El código de visualización del applet Simple (implementado por su método `paint()`) es defectuoso: no soporta desplazamiento. Una vez que el texto mostrado alcanza el final del rectángulo, no se puede ver el resto del texto.

La solución mas simple para este problema es utilizar un componente de un Interface de Usuario pre-fabricado (UI) que tenga un comportamiento correcto.

Nota: Esta página glosa sobre muchos detalles. Para aprender realmente a utilizar los componente de UI puedes ver [Crear un Interface de Usuario](#)

Componentes del UI Pre-Fabricado

El AWT suministra los siguientes componentes (la clase que implementa cada componente se lista entre parentesis):

- Botones (`java.awt.Button`)
- Checkboxes (`java.awt.Checkbox`)
- Campos de Texto de una linea (`java.awt.TextField`)
- Areas de Edicion y visualizacion de texto (`java.awt.TextArea`)
- Etiquetas (`java.awt.Label`)
- Listas (`java.awt.List`)
- Listas desplegables (`java.awt.Choice`)
- Barras de Desplazamiento (`java.awt.Scrollbar`)
- Areas de Dibujo (`java.awt.Canvas`)
- Menús (`java.awt.Menu`, `java.awt.MenuItem`, `java.awt.CheckboxMenuItem`)
- Contenedores (`java.awt.Panel`, `java.awt.Window` y sus subclases)

Métodos para utilizar Componentes UI en Applets

Como la clase `Applet` hereda desde la clase `Container` de AWT, es muy facil añadir componentes a un applet y utilizar manejadores de distribución para controlar la posición de los componentes en la pantalla. Aqui tiene algunos de los métodos `Container` que puede utilizar un applet:

`add()`

Añade el componente especificado.

`remove()`

Elimina el componente especificado.

`setLayout()`

Activa el manejador de Distribucion.

Añadir un Campo de Texto no Editable al applet Simple

Para hacer que el applet Simple utilice un campo de texto no editable con desplazamiento, utilizaremos la clase `TextField`. Aqui tienes el [código fuente](#) revisado. Los cambios se muestran mas abajo:

```
//Ya no se necesita importar java.awt.Graphics
//porque el applet no implementa el método paint().
. . .
import java.awt.TextField;

public class ScrollingSimple extends Applet {

    //En vez de utilizar un StringBuffer, usa un TextField:
    TextField field;

    public void init() {
        //Crea el campo de texto y lo hace no editable.
        field = new TextField();
        field.setEditable(false);

        //Activa el manejador de distribución para que el campo de
        //texto sea lo más ancho posible.
        setLayout(new java.awt.GridLayout(1,0));

        //Añade el campo de texto al applet.
        add(field);

        validate();

        addItem("initializing... ");
    }

    . . .
    void addItem(String newWord) {
        //Esto se utiliza para añadir el string al StringBuffer;
        //ahora se añade al campo de texto TextField.
        String t = field.getText();
        System.out.println(newWord);
        field.setText(t + newWord);
        repaint();
    }

    //El método paint() method ya no se necesita,
    //porque TextField se redibuja a sí mismo automáticamente.
```

El método revisado de `init()` crea un campo de texto no editable (un ejemplar de `TextField`). Se activa el manejador de distribución para que haga el campo de texto tan ancho como sea posible (conocerá más al detalle el manejador de distribución en [Distribuyendo componentes con un Contenedor](#)) y luego se añade el campo de texto al applet.

Después de todo esto, el método `init()` llama al método `validate()` (que el applet hereda de la clase `Component`). Se debe llamar a `validate()` cada vez que se añada uno o más componentes a un applet es una buena práctica que asegura que los componentes se dibujen a sí mismos en la pantalla. Si quieres ahondar en las razones de por qué trabaja el método `validate()`, lee [Detalles de la Arquitectura de Component](#)

Lo que ve abajo es el resultado del Applet modificado.

Threads en Applets

Nota: Esta página asume que ya sabes qué son los threads. Si no es así, puedes leer [¿Qué son los threads?](#) antes de leer esta página.

Cada applet puede ejecutar múltiples threads. A los métodos de dibujo de un Applet (paint() y update()) se les llama siempre desde los threads de dibujo y manejo de eventos del AWT. Las llamadas a las threads más importante de los métodos -- init(), start(), stop(), y destroy() --dependen de la aplicación que esté ejecutando el applet. Pero la aplicación siempre las llama desde los threads de dibujo y manejo de eventos del AWT.

Muchos navegadores, como el Netscape Navigator 2.0 para Solaris, asignan un thread para cada uno de los applets de una página, utilizando este thread para llamar a los métodos más importantes del applet. Algunos navegadores asignan un grupo de threads para cada applet, así es fácil eliminar todas los threads que pertencen a un applet particular. En cualquier caso, puedes estar seguro que cada una de los threads de los métodos más importantes de cualquier applet pertence al mismo grupo de threads.

Abajo tienes dos applets PrintThread. PrintThread es una versión modificada del applet Simple que imprime los threads llamados por sus métodos init(), start(), stop(), destroy(), y update(). Aquí tienes el código para el [ejemplo incompleto](#), y para el [ejemplo más interesante](#). Como es normal, para ver los métodos de salida como es destroy() que se llama durante la descarga necesitará ver la salida standard.

Entonces ¿Por qué necesita un applet crear y utilizar sus propios threads? Imagina un applet que realiza alguna inicialización que consume mucho tiempo --como la carga de imágenes, por ejemplo --en su método init(). El thread que llamó a init() no puede hacer nada hasta init()retorne. En algunos navegadores, esto podría significar que el navegador no podría mostrar el applet o nada hasta que el applet se hubiera inicializado a si mismo. Entonces si el applet está al principio de la página, por ejemplo, no se verá nada en la página hasta que el applet haya terminado su inicialización.

Incluso en los navegadores que crean un thread separado para cada applet, tiene sentido introducir los threads que consumen mas tiempo en threads creadas por el applet, para que el applet pueda realizar otros threads mientras espera a que se terminen los threads que consumen más tiempo.

Regla del Pulgar: Si un applet realiza trabajos que consumen mucho tiempo, deberá crear y utilizar sus propios threads para realizar esos trabajos.

Ejemplos de Threads en Applets

Esta página explica dos ejemplos de la utilización de threads en un applet. El primer applet, AnimatorApplet, muestra como utilizar un thread para realizar una tarea repetitiva. El applet AnimatorApplet viene de la página [Crear un Bucle de Animación](#). El segundo applet de esta página, SoundExample, muestra como utilizar threads para tareas de inicialización de una sólo vez. El applet SoundExample se encuentra en la página [Ejecutar Sonidos](#).

Esta página no explica el código básico de los threads. Para aprender algo más acerca de la implementación de los threads en Java, puedes referirte a [threads de Control](#)

Utilizar un thread para Realizar Tareas Repetitivas.

Normalmente, un applet que realiza la misma tarea una y otra vez debería tener un thread con un bucle while (o do...while) que realice la tarea. Un ejemplo típico es un applet que realiza una animación temporizada, como un visualizador de películas o un juego. Un applet de animación necesita un thread que le pida redibujarse a intervalos regulares.

Normalmente los applets crean los threads para las tareas repetitivas en el método start(). Crear el thread aquí hace muy fácil que el applet pueda parar el thread cuando el usuario abandone la página. Todo lo que se tiene que hacer es implementar el método stop() para que él detenga el thread del applet. Cuando el usuario retorne a la página del applet, se llama de nuevo al método start(), y el applet puede crear de nuevo el thread para realizar la tarea repetitiva.

Lo que puedes ver más abajo es la implementación de los métodos start() y stop() en la clase AnimatorApplet. (Aquí tienes todo el [código fuente](#) del applet.)

```
public void start() {
    if (frozen) {
        //No hace Nada. El usuario ha pedido que se pare la animación
    } else {
        //Empezar la animación!
        if (animatorThread == null) {
            animatorThread = new Thread(this);
        }
        animatorThread.start();
    }
}

public void stop() {
    animatorThread = null;
}
```

La palabra this en new Thread(this) indica que el applet proporciona el

cuerpo del thread. Y hace esto implementado el interface java.lang Runnable, que requiere que el applet proporcione un método run() que forme el cuerpo del thread. Explicaremos el método run() de AnimatorApplet un poco más adelante.

En el applet AnimatorApplet, el navegador no es el único que llama a los métodos start() y stop(). El applet se llama a sí mismo cuando el usuario pulsa con el ratón dentro del área del applet para indicar que la animación debe terminar. El applet utiliza un ejemplar de la variable frozen para seguir la pista de cuando el usuario pide que se pare la animación.

Habrás podido observar que no aparece en ningún sitio ninguna llamada al método stop() en la clase AnimatorApplet. Esto es así porque llamar al método stop() es como si quisiera meterle el thread a golpes por la cabeza. Es una manera drástica de conseguir que el applet deje lo que estaba haciendo. En su lugar, se puede escribir el método run() de forma que el thread salga de una forma adecuada y educada cuando se le golpee en el hombro. Este golpecito en el hombro viene a ser como poner a null un ejemplar de la variable del tipo Thread.

En AnimatorApplet, este ejemplar de la variable se llama animatorThread. El método start() lo activa para referirse a objeto Thread recientemente creado. Cuando el usuario necesite destruir el thread, pone animatorThread a null. Esto no elimina el thread realizando una recolección de basura --no puede haber recolección de basura mientras este ejecutable-- pero como al inicio del bucle, el thread comprueba el valor de animatorThread, y continúa o abandona dependiendo de este valor. Aquí tiene un código revelador:

```
public void run() {  
    . . .  
    while (Thread.currentThread() == animatorThread) {  
        ...//Muestra un marco de la aplicación.  
    }  
}
```

Si animatorThread se refiere al mismo thread que el que está realmente en ejecución el thread continua ejecutandose. Si por el contrario, animatorThread es null, el thread se termina y sale. Si animatorThread se refiere a otro thread, entonces se ha alcanzado una condición de competición: se ha llamado demasiado pronto a start(), antes de llamar a stop() (o este thread a tardado demasiado en su bucle) que start() ha creado otro thread antes de que este thread haya alcanzado el inicio del bucle while. Debido a la condición de competición este thread debería salir.

Para obtener más información sobre AnimatorApplet, puedes ir a: [Crear un Bucle de Animación](#).

Usar un thread para realizar Inicializaciones de una Vez

Si su applet necesita realizar alguna inicialización que tarde un poco, se debería considerar el modo de realizar esa inicialización en un thread. Por ejemplo, algo que requiera un trabajo de conexión a la red generalmente debería hacerse en un thread en segundo plano (background).

Afortunadamente, la carga de imágenes GIF y JPEG se hace automáticamente en background (utilizando threads de los que no tienes que preocuparse).

Desafortunadamente la carga de sonido, no está garantizado que se haga en segundo plano. En las implementaciones actuales los métodos de applet `getAudioClip` no retornan hasta que se han cargado los datos de audio. Como resultado, si se quiere precargar los sonidos, podría crear una o más tareas para hacer esto.

Utilizar un thread para realizar una tarea de inicialización en un applet es una variación del típico escenario productor/consumidor. El thread que realiza la tarea es el productor, y el applet el consumidor. [Sincronizar threads](#) explica como utilizar los threads de Java en un escenario productor/consumidor.

El applet `SoundExample` se adhiere al modelo presentado en [Sincronizar threads](#). Como el ejemplo anterior, el applet se caracteriza por tres classes:

- El productor [SoundLoader](#), una subclase `Thread`.
- El consumidor: [SoundExample](#), una subclase `Applet`. Al contrario que el consumidor en el ejemplo de sincronización de tareas, `SoundExample` no es un `Thread`; incluso tampoco implementa el interface ejecutable. Sin embargo, al menos, dos threads llaman a los ejemplares de los métodos de `SoundExample` dependiendo de la aplicación que ejecute el applet.
- El almacenaje del Objeto: [SoundList](#), Una subclase `Hashtable`. Al contrario que en el ejemplo de Sincronizar Tareas `SoundList` puede devolver valores nulos si los datos del sonido no se han almacenado todavía. Esto tiene sentido porque necesita estar disponible para reaccionar inmediatamente a una petición del usuario de que ejecute el sonido, incluso si el sonido no se ha cargado todavía.

Para más información sobre el `SoundExample`, puedes ir a [Ejecutar Sonidos](#)

Qué puede y qué no puede hacer un Applet

Esta página ofrece una amplia perspectiva tanto de las restricciones de los applets, como de las características especiales que estos tienen. Podrás encontrar más detalles en la lección: [Entender las Capacidades y las Restricciones de los Applets](#).

Rescticciones de Seguridad

Cada navegador implementa unos controladores de seguridad para prevenir que los applets no hagan ningún daño. Esta sección describe los controladores de Seguridad que poseen los navegadores actuales. Sin embargo, la implementación de controladores de seguridad es diferente de un navegador a otro. Los controladores de Seguridad también están sujetos a cambios. Por ejemplo, si el navegador está desarrollado para trabajar solo en entornos fiables, entonces sus controladores de seguridad serán mucho más flojos que los descritos en esta página.

Los Navegadores actuales imponen las siguientes restricciones a los applets que se cargan a través de la Red:

- Un applet no puede cargar librerías ni definir métodos nativos.
- No puede leer ni escribir ficheros en el Host en el que se está ejecutando.
- No puede realizar conexiones en la Red, excepto con el Host del que fue cargado.
- No puede arrancar ningún programa en el Host donde se está ejecutando.
- No puede leer ciertas propiedades del sistema.
- Las ventanas que proporcionan los applets tienen un aspecto diferente a las de cualquier aplicación.

Cada Navegador tiene un objeto `SecurityManager` que implementa sus controladores de seguridad. Cuando el objeto `SecurityManager` detecta una violación, lanza una `SecurityException` (Excepción de seguridad). Su applet puede capturar esta `SecurityException` y reaccionar del modo apropiado.

Capacidades de los Applets

El paquete `java.applet` proporciona una API que contiene algunas capacidades de los applets que las aplicaciones no tienen. Por ejemplo, los applets pueden ejecutar sonidos, que otros programas no pueden todavía.

Aquí tienes algunas cosas que pueden hacer los applets y que no se esperan:

- Los Applets pueden hacer conexiones al host del que fueron cargados.
 - Los Applets que se ejecutan dentro de un navegador Web pueden hacer que se muestren páginas HTML de una forma muy sencilla.
 - Los Applets pueden invocar métodos públicos de otros Applets que se encuentren en la misma página.
 - Los Applets que se han cargado desde un directorio local (desde un directorio en el CLASSPATH del usuario) no tienen ninguna restricción como los applets cargados a través de la Red.
 - Aunque la mayoría de los applets paran su ejecución cuando el usuario abandona la página, no tienen porque hacerlo.
-

Añadir un Applet a una Página HTML

Una vez que se ha escrito el código para un applet, se querrá ejecutarlo para probarlo. Para ver el applet en un navegador o en el AppletViewer del JDK, es necesario añadir el applet a una página HTML, usando la etiqueta `<APPLET>`. Después se debe especificar la URL de la página HTML en un navegador o en el AppletViewer.

Nota: Algunos navegadores no soportan fácilmente la recarga garantizada de applets. Por esta razón, tiene mucho sentido probar los applets en el AppletViewer hasta que haya alcanzado el punto en que se necesite probarlo en un navegador.

Esta página explica todo lo que debes saber para utilizar la etiqueta `<APPLET>`. Empieza utilizando la forma más simple de la etiqueta, y luego explica algunas de sus adiciones más comunes a la forma simple --el atributo `CODEBASE`, la etiqueta `<PARAM>`, y el código alternativo HTML.

El etiqueta `<APPLET>` más sencilla posible.

Aquí tiene la forma más simple de la etiqueta `<APPLET>`:

```
<APPLET CODE=SubclaseApplet.class WIDTH=anchura HEIGHT=altura>
</APPLET>
```

La etiqueta que se ve arriba le indica al navegador o visualizador de applets que cargue el applet cuya subclase `Applet`, llamada `SubclaseApplet`, se encuentra en el fichero `.class` situado en el mismo directorio que el documento HTML que contiene la etiqueta. La etiqueta anterior también especifica la altura y la anchura en pixels de la ventana del applet.

Cuando un navegador encuentra esta etiqueta, reserva un área de la pantalla con las dimensiones especificadas con `width` y `height`, carga los bytecodes de la subclase `Applet` especificada, crea un ejemplar de la subclase, y luego llama a los métodos `init()` y `start()` del ejemplar.

Especificar el Directorio del Applet con `CODEBASE`

Aquí tienes una etiqueta applet un poco más compleja. Se le ha añadido el atributo `CODEBASE` para decirle al navegador en que directorio se encuentran los bytecodes de la subclase `Applet`.

```
<APPLET CODE=SubclaseApplet.class CODEBASE=unaURL
        WIDTH=anchura HEIGHT=altura>
</APPLET>
```

Haciendo la `unaURL` como una dirección absoluta, se puede hacer que un documento cargado desde un servidor `HTTP://` ejecute un applet que se encuentra en otro servidor. Si `unaURL` es una URL relativa, entonces será interpretada en relación a la posición del documento HTML.

Por ejemplo, aquí tiene la etiqueta `<APPLET>` que incluye el applet Simple en [El Ciclo de Vida de un Applet](#):

```
<applet code=Simple.class width=500 height=20>
</applet>
```


Especificar Parámetros con la etiqueta <PARAM>

Algunos applets le permiten al usuario variar la configuración del applet utilizando parámetros. Por ejemplo, AppletButton (un applet utilizado en este tutorial para proporcionar un botón que muestra una ventana) le permite al usuario seleccionar el texto del botón especificando un valor en el parámetro llamado BUTTONTEXT. Aprenderás como escribir código para proporcionar parámetros en [Definir y Utilizar parámetros en un Applet](#).

Aquí tienes un ejemplo del formato de la etiqueta <PARAM>. Observa que la etiqueta <PARAM> debe aparecer entre las etiquetas <APPLET> y </APPLET> para que tenga efecto en el applet.

```
<APPLET CODE=SubclaseApplet.class WIDTH=anchura HEIGHT=altura>
<PARAM NAME=parámetro1 VALUE=Valor>
<PARAM NAME=parámetro2 VALUE=otroValor>
</APPLET>
```

Aquí tienes un ejemplo del uso de la etiqueta <PARAM>.

```
<applet code=AppletButton.class width=350 height=60>
<param name=windowType value=BorderWindow>
<param name=windowText value="BorderLayout">
<param name=buttonText value="Pulse aquí para ver BorderLayout en acción">
. . .
</applet>
```

Especificar el Texto a mostrar en los Navegadores que no soportan Java

Habrás notado los puntos (". . .") en el ejemplo anterior. ¿Qué ha dejado fuera el ejemplo? Código HTML alternativo --código HTML interpretado solo por lo navegadores que no entienden la etiqueta <APPLET>.

Si la página que contiene un applet pudiera ser vista por personas que no tienen navegadores compatibles con Java, se debe proporcionar código alternativo HTML para que la página pueda tener sentido. El código alternativo HTML es cualquier texto entre las etiquetas <APPLET> y </APPLET>, excepto las etiquetas <PARAM>. Los navegadores compatibles con Java ignoran el código HTML alternativo.

Aquí tienes el código HTML completo para el ejemplo anterior de AppletButton:

```
<applet code=AppletButton.class width=350 height=60>
<param name=windowType value=BorderWindow>
<param name=windowText value="BorderLayout">
<param name=buttonText value="Pulse aquí para ver BorderLayout en acción">
<blockquote>
<hr>
Su navegador no puede ejecutar Applets Java,
<hr>
</blockquote>
</applet>
```

Un navegador que no entienda la etiqueta <APPLET> ignora todo lo que hay hasta la etiqueta <blockquote>. Un navegador que entienda la etiqueta <APPLET> ignora todo lo que hay entre las etiquetas <blockquote> y </blockquote>.

Sumario: Applets

Esta lección ofrece mucha información --casi todo lo que se necesita saber para escribir applets Java. Esta página resume todo lo que se ha aprendido, añadiendo un poco más de información para ayudarte a entender el cuadro completo.

Lo primero que se aprendió para escribir un applet es que se debe crear una subclase de la clase `java.applet`. En esta subclase, se debe implementar al menos uno de los siguientes métodos: `init()`, `start()`, y `paint()`. Los métodos `init()` y `start()`, junto con `stop()` y `destroy()`, son los llamados eventos más importantes (milestones) que ocurren en el ciclo de vida de un applet. Se llama al método `paint()` cuando el applet necesita dibujarse en la pantalla.

La clase `Applet` descende de la clase `AWT Panel`, que descende a su vez de la clase `AWT Container`, que descende a su vez de la clase `AWT Component`. De la clase `Component`, un applet hereda las capacidades de dibujar y manejar eventos. De la clase `Container`, un applet hereda la capacidad de añadir otros componentes y de tener un manejador de distribución para controlar su tamaño y posición. De la clase `Panel`, un applet hereda mucho, incluyendo la capacidad de responder a los principales eventos en el ciclo de vida, como son la carga y la descarga.

Los applets se incluyen en páginas HTML utilizando la etiqueta `<APPLET>`. Cuando un usuario visita una página que contiene un applet, aquí tienes lo que sucede:

1. El navegador encuentra el fichero `.class` que contiene la subclase `Applet`. La posición del fichero `.class` (que contiene los bytecodes Java) se especifica con los atributos `CODE` y `CODEBASE` de la etiqueta `<APPLET>`.
2. El navegador trae los bytecodes a través de la red al ordenador del usuario.
3. El navegador crea un ejemplar de la subclase `Applet`. Cuando nos referimos a un applet, normalmente nos referimos a este ejemplar.
4. El navegador llama al método `init()` del applet. Este método realiza una necesaria inicialización.
5. El navegador llama al método `start()` del applet. Este método normalmente arranca un thread para realizar las tareas del applet.

Lo principal en un applet es la subclase `Applet`, clase controladora, pero los applets también pueden utilizar otras clases. Estas otras clases pueden ser propias del navegador, proporcionadas como parte del entorno Java o clases del usuario suministradas por ti. Cuando un applet intenta ejecutar una clase por primera vez, el navegador intenta encontrarla en el host donde se está ejecutando el navegador. Si no puede encontrarla allí, la busca en el mismo lugar de donde cargó la subclase `Applet` del applet. Cuando el navegador encuentra la clase, carga sus bytecodes (a través de la Red, si es necesario) y continua la ejecución del applet.

Cargar código ejecutable a través de la red es un riesgo de seguridad. Para los applets Java, algunos de estos riesgos se reducen, porque el lenguaje Java está

diseñado para ser seguro --por ejemplo, no permite punteros a posiciones de memoria. Además, los navegadores compatibles con Java proporcionan seguridad imponiendo algunas restricciones. Estas restricciones incluyen no permitir a los applets la carga de código escrito en otro lenguaje que no sea Java, y tampoco permiten leer o escribir ficheros en el host donde se está ejecutando el navegador.

Ozito

Crear un Interface Gráfico de Usuario

Casi todos los applets tienen un interface gráfico de usuario (GUI). Esta página explica unos pocos problemas que son particulares de los GUI de los applets.

Un Applet está en un Panel.

Como Applet es una subclase de la clase Panel del AWT, los applets pueden contener otros Componentes, al igual que puede hacerlo la clase Panel. Los Applets heredan el controlador de distribución por defecto de la clase Panel: FlowLayout. Al igual que la clase Panel (y por lo tanto la clase Components), la clase Applet participa en herencia de dibujo y eventos del AWT.

Los Applets aparecen en una ventana ya existente del navegador.

Esto tiene dos implicaciones.

Primero, al contrario que las aplicaciones basadas en GUI, los applets no tienen que crear una ventan para mostrarse. Pueden hacerlo si tienen una buena razón, pero normalmente sólo se muestran en la ventana del navegador.

Segundo, dependiendo de la implementación del navegador, los componentes de sus applets podrían no ser mostrados a menos que el applet llame al método validate() después de haber añadido componentes.

Afortunadamente, llamar a validate() no hace ningún daño.

El color de fondo del applet podría no coincidir con el color de la página.

Por defecto, los applets tienen un color de fondo Gris brillante. Sin embargo, las páginas HTML, pueden tener otros colores de fondo y pueden utilizar dibujos de fondo. Si el diseño del applet y de la página no son cuidadosos, si el applet tiene difetente color de fondo puede hacer que sobresalga sobre la página, o puede causar un parpadeo notable cuando se dibuje el applet. Una solución es definir un parámetro del applet que especifique el color del fondo. La subclase Applet puede utilizar el método setBackground() de Component para seleccionar el color del fondo con un color especificado por el usuario. Utilizando el parámetro del color de fondo el applet, el diseñador de la página puede elegir el color de fondo del applet que sea apropiado para el color de la página. Aprenderás más sobre los parámetros en la lección [Definir y Utilizar Parámetros en un Applet](#).

Cada Applet tiene un tamaño predetermiando especificado por el usuario.

Como la etiqueta <APPLET> requiere que se especifiquen la altura y anchura del applet, y cómo el navegador no permite necesariamente que el applet se redimensione a sí mismo, los applets deben trabajar con una cantidad de espacio que podría no ser la ideal. Incluso si el espacio es ideal para una plataforma, las partes específicas para una plataforma de un applet (como los botones) podrían requerir una mayor cantidad de espacio en otra plataforma. Se puede compensar esto recomendando que las páginas que incluyen sus applets especifiquen un poco más de espacio del que podría ser necesario, y mediante la utiliziación de distribuciones flexibles como las del AWT -- que

proporciona las clases `GridBagLayout` y `BorderLayout` para adaptar el espacio extra.

Los Applets cargan imágenes utilizando el método `getImage()` de la clase `Applet`.

La clase `Applet` proporciona un forma conveniente de `getImage()` que le permite especificar una URL base como un argumento, seguido por un segundo argumento que especifica la posición del fichero de imagen, relativo a la URL base. Los métodos `getCodeBase()` y `getDocumentBase()` de la clase `Applet` proporcionan la dirección URL base que casi todos los applets utilizan. Las imágenes que siempre necesita un applet, o necesita relacionarlas con un backup, normalmente se especifican relativas a la dirección de donde fue cargado el applet (indicada en la etiqueta code base). Las imágenes especificadas por el usuario del applet (normalmente como parámetros en el fichero HTML) son normalmente relativas a la dirección de la página que incluye al applet (el document base).

Las clases `Applet` (y todos los ficheros de datos que estas utilizan) son cargados a través de la red, lo que puede ser bastante lento.

Los Applets pueden hacer muchas cosas para reducir el tiempo de arranque. La subclase `Applet` puede ser una clase pequeña que muestre inmediatamente un mensaje de estado. Y, si algunas de las clases del applet o los datos no se utilizan por el momento, el applet puede precargar las clases o los datos en un thread en segundo plano.

Por ejemplo, el método `start()` de la clase [AppletButton](#) lanza un thread que obtiene el objeto `Class` para la ventana en la que se va a mostrar el botón. El propósito principal del applet para hacer esto es asegurarse de que el nombre de la clase es válido. Un beneficio añadido es que obtener el objeto `Class` fuerza a que se cargue el fichero de la clase antes de ejemplarizarla. Cuando el usuario pide que se cree una ventana, el applet ejemplariza la clase `Windows` mucho más rápido que si el applet todavía tuviera que cargar el fichero de la clase.

Ejecutar Sonidos en Applets

En el paquete `java.applet`, la clase `Applet` y el interface `AudioClip` proporcionan un interface básico para ejecutar sonidos. Actualmente, el API de Java soporta un sólo formato de sonido: 8 bit, 8000 hz, un canal, en ficheros ".au" de Sun. Se pueden crear estos ficheros en una estación Sun utilizando la aplicación `audiotool`. Se puede convertir ficheros desde otros formatos de sonido utilizando un programa de conversión de formatos de sonidos.

Métodos Relacionados con el Sonido

Abajo tienes los métodos para Applets relacionados con el sonido. El formato de dos argumentos de cada método toma una dirección URL base (generalmente devuelta por `getDocumentBase()` o `getBase()`) y la posición del fichero de sonido relativa a la URL base. Se debería utilizar la dirección base para sonidos que están integrados en el applet. La dirección del documento base se utiliza generalmente para sonidos especificados por el usuario del applet a través de los parámetros del applet.

[`getAudioClip\(URL\)`](#), [`getAudioClip\(URL, String\)`](#)

Devuelven un objeto que implementa el interface `AudioClip`.

[`play\(URL\)`](#), [`play\(URL, String\)`](#)

Ejecutan el `AudioClip` correspondiente a la URL especificada.

El Interface `AudioClip` define los siguientes métodos:

[`loop\(\)`](#)

Empieza la ejecución del Clip repetidamente.

[`play\(\)`](#)

Ejecuta el clip una vez.

[`stop\(\)`](#)

Para el Clip. Trabaja con el método `loop()` y el método `play()`.

Un Ejemplo

Aquí tiene un applet llamado `soundExample` que ilustra unas pocas cosas sobre el sonido. Observa que, para propósitos de instrucciones, el applet añade 10 segundos al tiempo de carga de cada sonido. Si el fichero de sonido fuera mayor o la conexión del usuario fuera más lenta que la nuestra, este retardo podría ser realista.

El applet `SoundExample` proporciona una arquitectura para la carga y ejecución de múltiples sonidos en un applet. Por esta razón, es más complejo de lo necesario. Exencialmente, los códigos de carga y ejecución del sonido se parecen a esto:

```
AudioClip onceClip, loopClip;
onceClip = applet.getAudioClip(getCodeBase(), "bark.au");
loopClip = applet.getAudioClip(getCodeBase(), "train.au");
onceClip.play();           //Lo ejecuta una vez..
loopClip.loop();           //Empieza el bucle de sonido.
loopClip.stop();           //Para el bucle de sonido.
```

Como no hay nada más molesto que un applet que continúe haciendo ruido después de haber abandonado la página, el applet `SoundExample` para de ejecutar el bucle de sonido continuo cuando el usuario abandona la página y empieza de nuevo cuando el usuario vuelve a ella. Hace esto mediante la implementación de sus métodos `stop()` y `start()` de la siguiente forma:

```
public void stop() {
    //Si el sonido de una vez fuera largo, lo parariamos aquí.
    //looping es un ejemplar de una variable booleana que esta inicializada a false.
    //Se pone a true cuando se pulsa el botón de "Start sound loop" y se pone a false
    //cuando se pulsa los botones de "Stop sound loop" o "Reload sounds".
```

```

        if (looping) {
            loopClip.stop();    //Para el bucle de sonido.
        }
    }

    public void start() {
        if (looping) {
            loopClip.loop();    //Reinicia el bucle de sonido.
        }
    }
}

```

El applet SoundExample construye tres clases:

- Una subclase de Applet, [SoundExample](#), que controla la ejecución del applet.
- Una subclase de Hashtable, [SoundList](#), que contiene los AudioClip. Esto es demasiado para este applet, pero si usted escribiera una applet que utilizará muchos ficheros de sonidos, una clase como esta le sería muy útil.
- Una subclase de Thread, [SoundLoader](#), cada ejemplar de la cual carga un AudioClip en segundo plano. Durante la inicialización del applet, éste precarga cada sonido mediante la creación de un SoundLoader para él.

La carga de sonidos en threads de segundo plano (con SoundLoader) aumenta el rendimiento percibido mediante la reducción del tiempo que el usuario tiene que esperar para poder interactuar con el applet. Se hace así para reducir el tiempo gastado por el método init(). Si se llama simplemente a getAudioClip() en el método init() del applet, tardará bastante antes de que getAudioClip retorne, lo que significa que el applet no puede realizar otras sentencias de su método init(), y que no se llamará al método start() del applet. (Para este applet en particular, un retardo en la llamada al método start() no tiene importancia).

Otra ventaja de la carga de sonidos utilizando threads en segundo plano es que permite que el usuario responda apropiada (e inmediatamente) a una entrada del usuario, que normalmente cuasaría la ejecución de un sonido, incluso si el sonido no se ha cargado todavía. Por ejemplo, si simplemente se utiliza el método play() de Applet, la primera vez que el usuario haga algo para que se ejecute un sonido particular, el dibujo del applet y el manejo de eventos quedaran congelados mientras se carga el sonido. En vez de esto, éste applet detecta que el sonido no ha sido cargado todavía y responde de forma apropiada.

Este ejemplo se explica con más detalle en [Threads en Applets: Ejemplos](#).

Definir y Utilizar Parámetros en un Applet

Los parámetros son a los applets lo que los argumentos de la línea de comandos a las aplicaciones. Permiten al usuario personalizar la operación del applet. Mediante la definición de parámetros, se puede incrementar la flexibilidad de los applets, marcando el trabajo de sus applets en múltiples situaciones sin tener que escribirlos de nuevo ni recompilarlos.

Las páginas siguientes explican los parámetros de los applets desde el punto del programador. Para aprender los parámetros desde el punto de vista del usuario puede ver: [Especificar Parámetros con la Etiqueta <PARAM>](#).

Decidir los Parámetros a Soportar

Cuando se implementan parámetros se deberá responden a cuatro preguntas:

- ¿Qué debería permitirsele configurar al usuario de un applet?
- ¿Cómo deberían llamarse los parámetros?
- ¿Qué clases de valores deberían contener los parámetros?
- ¿Cuál debería ser el valor por defecto de cada parámetro?

Escribir Código para Soportar Parámetros

Los Applets obtienen los valores de los parámetros definidos por el usuario llamando al método `getParameter()` de la clase `Applet`.

Dar Información sobre los Parámetros

Mediante la implementación del método `getParameterInfo()`, los applets proporcionan información sobre que navegadores se pueden utilizar para ayudar al usuario a seleccionar los parámetros.

Decidir los Parámetros a Soportar

Esta página te guía a través de las cuatro preguntas que deberías responder cuando implementes parámetros:

- [¿Qué debería permitirsele configurar al usuario de un applet?](#)
- [¿Cómo deberían llamarse los parámetros?](#)
- [¿Qué clases de valores deberían contener los parámetros?](#)
- [¿Cuál debería ser el valor por defecto de cada parámetro?](#)

Termina con una explicación sobre la definición de parámetros con una clase de ejemplo llamada AppletButton.

¿Qué debería permitirsele configurar al usuario de un applet?

Los parámetros que un applet deberían soportar dependen de lo que haga su applet, y de la flexibilidad que quieras que tenga. Los applets que muestran imágenes podrían tener parámetros para especificar la posición de las imágenes. De forma similar, los applets que ejecutan sonidos podrían tener parámetros para especificar los sonidos.

Junto con los parámetros que especifican localizaciones de recursos (como imágenes o ficheros de sonido), algunas veces los applets proporcionan parámetros para especificar detalles de la apariencia u operación del applet. Por ejemplo, un applet de animación podría permitir que el usuario especificara el número de imágenes por segundo. O un applet podría permitir que el usuario cambie los textos mostrados por el applet. Todo es posible.

¿Cómo deberían llamarse los parámetros?

Una vez que has decidido los parámetros soportados por el applet, necesita diseñar sus nombres. Aquí tienes algunos nombres típicos de parámetros:

SOURCE o SRC

Para un fichero de datos como un fichero de imágenes.

XXXSOURCE (por ejemplo, IMAGESOURCE)

Utilizado en los applets que permiten al usuario utilizar más de un tipo de ficheros de datos.

XXXS

Para un parámetro tomado de una lista de XXXs (donde XXX podría ser IMAGE, de nuevo).

NAME

Utilizado solo para un nombre de applet. Los nombres de applets se utilizan para la comunicación con otros applets, como se describe en [Enviar Mensajes a Otros Applets en la Misma Página](#).

Que el nombre sea claro y conciso es más importante que su longitud.

Nota: Aunque este tutorial normalmente se refiere a los nombres de parámetros utilizando MAYÚSCULAS, éstos no son sensibles al caso. Por ejemplo,

IMAGE_SOURCE e imageSource se refieren al mismo parámetro.

Por otro lado, los valores de los parámetros si son sensibles al caso, a menos que se los interprete de otra forma (como utilizando el método toLowerCase() de la clase String antes de interpretar el valor del parámetro).

¿Qué clases de valores deberían contener los parámetros?

Todos los valores de parámetros son cadenas. Tanto si el usuario pone comillas o no alrededor del valor del parámetro, el valor es pasado al applet como una cadena. Sin embargo, un applet puede interpretar las cadenas de muchas formas.

Los applets interpretan normalmente un valor de un parámetro como uno de los siguientes tipos:

- Una dirección URL
- Un entero
- Un número en coma flotante
- Un valor booleano -- típicamente "true"/"false"
- Una cadena -- por ejemplo, la cadena utilizada para el título de una ventana.
- Una lista de cualquiera de los tipos anteriores

¿Cuál debería ser el valor por defecto de cada parámetro?

Los applets deberían intentar proporcionar valores útiles por defecto para cada parámetro, para que el applet se ejecute incluso si el usuario no ha especificado ningún parámetro o los ha especificado incorrectamente. Por ejemplo, un applet de animación debería proporcionar un número adecuado de imágenes por segundo. De esta forma, si el usuario no especifica un parámetro relevante, el applet trabajará de forma correcta.

Un Ejemplo: AppletButton

A través de este tutorial, los applets que necesitan mostrar ventanas utilizan la clase [AppletButton](#) que es altamente configurable. El GUI de AppletButton es sencillo, consiste en un botón y una etiqueta que muestra el estado. Cuando el usuario pulsa sobre el botón, el applet muestra una ventana.

La clase AppletButton es tan flexible porque define parámetros para permitir que usuario especifique cualquiera de los siguientes:

- El tipo de la ventana a mostrar.
- El Título de la ventana.
- La altura de la ventana.
- La anchura de ventana.
- La etiqueta del botón que muestra la ventana.

Una etiqueta <APPLET> típica para AppletButton se parecería a esto:

```
<APPLET CODE=AppletButton.class CODEBASE=clases WIDTH=350 HEIGHT=60>
<PARAM NAME=windowClass VALUE=BorderWindow>
<PARAM NAME=windowTitle VALUE="BorderLayout">
<PARAM NAME=buttonText VALUE="Pulse aquí para ver BorderLayout en acción">
```

</APPLET>

Cuando el usuario no especifica un valor para un parámetro, AppletButton utilizan un valor por defecto razonable. Por ejemplo, si el usuario no especifica el título de la ventana, AppletButton utiliza el tipo de la ventana como título.

La página siguiente muestra el código que utiliza AppletButton para obtener los valores de sus parámetros.

Escribir el Código para Soportar Parámetros

Los applets utilizan el método `getParameter()` de la clase `Applets` para obtener los valores especificados por el usuario para sus parámetros. Este método está definido de la siguiente forma:

```
public String getParameter(String name)
```

Su applet podría necesitar convertir la cadena devuelta por `getParameter()` en otro formato, como un entero. El paquete `java.lang` proporciona clases como `Integer` que se puede utilizar para la conversión de las cadenas en tipos primitivos. Aquí tienes un ejemplo de la clase `Appletbutton` para convertir el valor de un parámetro en una cadena:

```
int requestedWidth = 0;
. . .
String windowWidthString = getParameter("WINDOWWIDTH");
if (windowWidthString != null) {
    try {
        requestedWidth = Integer.parseInt(windowWidthString);
    } catch (NumberFormatException e) {
        //Utiliza la anchura por defecto.
    }
}
```

Observa que si el usuario no especifica el valor para el parámetro `WINDOWWDTH`, el utiliza el valor por defecto de 0, lo que el applet interpreta como "utiliza el tamaño natural de la ventana". Es importante que se suministren valores por defecto donde sea posible.

Un ejemplo: AppletButton

Abajo tienes el código de `AppletButton` que obtiene los parámetros del applet. Para más información sobre `Appletbutton` puedes ver la [página anterior](#).

```
String windowClass;
String buttonText;
String windowTitle;
int requestedWidth = 0;
int requestedHeight = 0;
. . .
public void init() {
    windowClass = getParameter("WINDOWCLASS");
    if (windowClass == null) {
        windowClass = "TestWindow";
    }

    buttonText = getParameter("BUTTONTEXT");
    if (buttonText == null) {
        buttonText = "Pulse aquí para ver una ventana " + windowClass;
    }
}
```

```
windowTitle = getParameter("WINDOWTITLE");
if (windowTitle == null) {
    windowTitle = windowClass;
}

String windowWidthString = getParameter("WINDOWWIDTH");
if (windowWidthString != null) {
    try {
        requestedWidth = Integer.parseInt(windowWidthString);
    } catch (NumberFormatException e) {
        //Utiliza la anchura por defecto.
    }
}

String windowHeightString = getParameter("WINDOWHEIGHT");
if (windowHeightString != null) {
    try {
        requestedHeight = Integer.parseInt(windowHeightString);
    } catch (NumberFormatException e) {
        //Utiliza la altura por defecto.
    }
}
```

Dar Información sobre los Parámetros

Ahora que hemos proporcionado al usuario unos parámetros preciosos, necesitamos ayudarle a seleccionar los valores de los parámetros correctamente. Por supuesto, la documentación de un applet debería describir cada parámetro y un ejemplo al usuario y aconsejarle sobre su selección. Por lo tanto, nuestro trabajo no termina aquí. Deberíamos implementar el método `getParameterInfo()` para que devuelva información sobre los parámetros del applet. Los navegadores pueden utilizar esta información para ayudar al usuario a seleccionar los valores de los parámetros del applet.

Aquí tienes un ejemplo de implementación del método `getParameterInfo()`. Este ejemplo es del applet [Animator](#), que es maravillosamente flexible ya que proporciona 13 parámetros para que el usuario pueda personalizar su animación.

```
public String[][] getParameterInfo() {
    String[][] info = {
        // Parameter Name      Kind of Value      Description
        {"imagesource",        "URL",             "a directory"},
        {"startup",            "URL",             "displayed at startup"},
        {"background",         "URL",             "displayed as background"},
        {"startimage",         "int",             "start index"},
        {"endimage",           "int",             "end index"},
        {"namepattern",        "URL",             "used to generate indexed names"},
        {"pause",              "int",             "milliseconds"},
        {"pauses",             "ints",            "milliseconds"},
        {"repeat",             "boolean",         "repeat or not"},
        {"positions",          "coordinates",     "path"},
        {"soundsource",        "URL",             "audio directory"},
        {"soundtrack",         "URL",             "background music"},
        {"sounds",             "URLs",            "audio samples"},
    };
    return info;
}
```

Como puedes ver, el método `getParameterInfo()` debe devolver un array de tres-cadenas. En cada array de tres cadenas, la primera cadena es el nombre del parámetro. La segunda cadena le aconseja al usuario sobre el tipo de valor general que el applet necesita para ese parámetro. La tercera cadena describe el significado de ese parámetro.

Leer las Propiedades del Sistema

Para encontrar información sobre el entorno de trabajo actual los applets pueden leer las propiedades del sistema. Las propiedades del sistema son parejas de clave/valor que contienen información como el sistema operativo bajo el que se está ejecutando el applet. Las propiedades del sistema se cubrieron en más detalle en la página [Propiedades del Sistema](#).

Los applets pueden leer sólo algunas propiedades del sistema. Esta página lista las propiedades del sistema que el Netscape Navigator 2.0 y el Applet Viewer permiten leer a los applets, seguida por la lista de propiedades que los applets no pueden leer.

Propiedades del Sistema que Pueden Leer los Applets

Los applets pueden leer las siguientes propiedades del sistema:

Clave	Significado
"file.separator"	Separador de Fciheros (e.g., "/")
"java.class.version"	Número de Versión de la Clase Java
"java.vendor"	Cadena Especifica del Vendedor Java
"java.vendor.url"	URL del vendedor Java
"java.version"	Número de versión de Java
"line.separator"	Separador de Líneas
"os.arch"	Arquitectura del Sistema Operativo
"os.name"	Nombre del Sistema Operativo
"path.separator"	Separador de Path (e.g., ":")

Para leer las propiedades del sistema dentro de un applet, se puede utilizar el método `getProperty()` de la clase `System`. Por ejemplo:

```
String s = System.getProperty("os.name");
```

Propiedades del Sistema Prohibidas

Por razones de seguridad, no existen navegadores o visualizadores de applets que permitan a los applets leer las siguientes propiedades del sistema.

Clave	Significado
"java.class.path"	Directorio de Clases Java
"java.home"	Directorio de instalación de Java
"user.dir"	Directorio de trabajo actual
"user.home"	Directorio principal del usuario
"user.name"	Nombre de Cuenta del usuario

Mostrar Cadenas Cortas de Estado

Todos los visualizadores de applets -- desde el Applet Viewer hasta el Netscape Navigator -- permiten que los applets muestren cadenas cortas de estado. En implementaciones actuales, las cadenas aparecen en la línea de estado en la parte inferior de la ventana del visualizador. En los navegadores, todos los applets de la misma página, así como el propio navegador comparten la misma línea de estado.

Nunca se debería poner información crucial en la línea de estado. Si muchos usuarios necesitaran leer la información, se debería mostrar en el área del applet. Si sólo unos pocos y sofisticados usuarios necesitan esa información, consideraremos el mostrar la información en la salida estándar (puedes ver la [página siguiente](#)).

La línea de estado no es normalmente muy prominente y puede ser sobreescrita por otros applets o por el navegador. Por estas razones, es mejor utilizarla para información transitoria. Por ejemplo, un applet que carga muchos ficheros de imágenes podría mostrar el nombre de la imagen que está cargando en ese momento.

Los applets pueden mostrar líneas de estado con el método [showStatus\(\)](#). Aquí tienes un ejemplo de su utilización:

```
showStatus("MyApplet: Cargando el Fichero de Imagen " + file);
```

Mostrar diagnósticos en los Canales de Salida y Error Estandards

Mostrar diagnósticos en la salida estandard puede ser una herramienta imprescindible cuando se está depurando applets. Otro momento en el que se podrán ver mensajes en la salida estandard es cuando ocurre una excepción no capturada en un applet. Los applets también tienen la opción de utilizar el canal de error estandard.

Donde están exactamente los canales de salida y error estandard varia, dependiendo de cómo está implementado el visualizador de applets, la plataforma en la que se esté ejecutando, y (algunas veces) cómo se ha lanzado el navegador o el visualizador de applets. Cuando se lanza el Applet Viewer desde una ventana del Shell de UNIX, por ejemplo, las cadenas mostradas en los canales de salida y error estandards aparecen en esa ventana del shell, a menos que se redireccione la salida. Cuando se lanza el Applet Viewer desde un menú de XWindows, la salida y el error estandard van a la ventana de la consola. Por otro lado, Netscape Navigator 2.0 siempre muestra los canales de salida y de error estandards en la Consola Java, que está disponible desde el menú Options.

Los applets muestran un canal de salida estandard utilizando `System.out.print(String)` y `System.out.println(String)`.

Mostrar el canal del error estandard es similar; sólo se debe especificar `System.err` en vez de `System.out`. Aquí tienes un ejemplo utilizando la salida estandard:

```
//Donde se declaren las variables de ejemplar:
boolean DEBUG = true;
. . .
//Después, cuando queramos imprimir algún estado:
if (DEBUG) {
    System.out.println("Called someMethod(" + x + "," + y + ")");
}
```

Nota: Mostrar los canales de salida y de error estandard es relativamente lento. Si se tienen problemas relacionados con el tiempo, imprimir mensajes en alguno de estos canales podría no ser útil.

Comunicarse con Otros Programas

Un applet puede comunicarse con otros programas de tres formas:

- Invocando los métodos públicos de otros applets de la misma página (sujetos a las restricciones de seguridad).
- Utilizando el API definido en el paquete `java.applet`, que permite las comunicaciones de una forma limitada con el navegador o el visualizador de applets que lo contiene.
- Utilizando el API del paquete `java.net` para comunicarse a través de la red con otros programas. Los otros programas deben estar ejecutándose en el host de donde se trajo el applet originalmente.

Estas lecciones explican y muestran ejemplos de estas tres clases de comunicación de applets.

Enviar Mensajes a otros Applets en la Misma Página

Utilizando los métodos `getApplet()` y `getApplets()` del `AppletContext` un applet puede obtener objetos de la clase `Applet` de otros applets que se están ejecutando en la misma página. Una vez que un applet tiene un objeto `Applet` de otro applet puede enviarle mensajes.

Comunicarse con el Navegador

Varios métodos de `Applet` y `AppletContext` proporcionan comunicación limitada entre el applet y el navegador o el visualizador en el que se está ejecutando. Los más interesantes son probablemente los métodos `showDocument()` de `AppletContext`, que le permite al applet decirle al navegador que URL debe mostrar.

Trabajar con Aplicaciones en el Lado del Servidor

Los applets pueden utilizar las características de la Red como lo haría cualquier programa Java, con la restricción de que todas las comunicaciones deben ser con el host que es el host actual para el applet bajado. Esta sección presenta una versión de un applet de [Trabajar con Datagrama Cliente y Servidor](#).

También en esta sección hay un ejemplo de cómo utilizar una aplicación en el lado del servidor para evitar las restricciones de seguridad de los applets. En este ejemplo, los applets originarios del mismo host pero que se están ejecutando en diferentes máquina pueden comunicarse utilizando una aplicación del lado del servidor como intermediario.

Enviar Mensajes a otros Applets en la misma Página

Los applets pueden encontrar otros applets y enviarles mensajes, con la siguientes restricciones de seguridad:

- Los applets deben ejecutarse en la misma página, en la misma ventana del navegador.
- Muchos visualizadores de applets requieren que los applets sean originales del mismo servidor.

Un applet puede encontrar otro applet buscándolo por su nombre (utilizando el método `getApplet()` de `AppletContext`) o buscando todos los applets de la página (utilizando el método `getApplets()` de `AppletContext`). Ambos métodos, si tienen éxito, le dan al llamador uno o más objetos `Applet`. Una vez que el llamador ha encontrado un objeto `Applet`, el llamador puede llamar a los métodos del objeto.

Encontrar un Applet por el nombre: el método `getApplet()`

Por defecto, un applet no tiene nombre. Para que un applet tenga nombre debe especificarse uno en el código HTML que se añade a la página del applet. Se puede especificar un nombre para un applet de dos formas diferentes:

- Mediante la especificación de un atributo `NAME` dentro de la etiqueta `<APPLET>` del applet. Por ejemplo:

```
<applet codebase=clases/ code=Sender.class width=450 height=200 name="buddy">
. . .
</applet>
```

- Mediante la especificación de un parámetro `NAME` con una etiqueta `<PARAM>`. Por ejemplo:

```
<applet codebase=clases/ code=Receiver.class width=450 height=35>
<param name="name" value="old pal">
. . .
</applet>
```

Nota del Navegador: Las versiones 2.0 y 2.1 del Netscape Navigator no permiten que los nombres tengan letras mayúsculas. Específicamente, el método `getApplet()` (el método que busca un applet por su nombre) parece convertir el nombre especificado a minúsculas antes de comenzar la búsqueda del applet.

Abajo tienes dos applets que ilustran la búsqueda por el nombre. El primero, Remitente, busca al segundo, Receptor. Cuando el Remitente encuentra al Receptor, le envía un mensaje llamando a uno de los métodos del Receptor (pasando el nombre del Remitente como un argumento). El Receptor reacciona a la llamada de este método cambiando la cadena situada a la izquierda por "Received message from sender-name!".

Intenta Esto: Pulse el botón Send message en el applet superior (Remitente). Aparecerá alguna información de estado en la ventana del Remitente, y el Receptor confirmará (con su propia cadena de estado) que ha recibido el mensaje. Después de haber leído el mensaje del Receptor, pulsa el botón Clear del Receptor para resetearlo.

Intenta Esto: En el campo del texto del Remitente llamado "Receiver name:", teclea buddy y pulsa Return. Como "buddy" es el nombre del propio Remitente, encontrará un applet llamado buddy pero no le enviará un mensaje, ya que no es un ejemplar de Receptor.

Aquí tiene el programa [Remitente Completo](#). El código es utiliza para buscar y comunicarse con el Receptor listado más abajo. El código que se puede utilizar sin cambiarlo se encuentra en **negrita**.

```
Applet receiver = null;
String receiverName = nameField.getText();           //Obtiene el nombre por el que
buscar.
receiver = getAppletContext().getApplet(receiverName);
```

El Remitente se asegura de que se ha encontrado un Receptor y que es un ejemplar de la clase correcta (`Receiver`). Si todo va bien, el Remitente envía un mensaje al receptor. (Aquí tiene el

programa del [Receptor](#).)

```
if (receiver != null) {
    //Utiliza el ejemplar del operador para asegurarse de que el applet
    //que hemos encontrado un objeto Receiver
    if (!(receiver instanceof Receiver)) {
        status.appendText("Found applet named "
            + receiverName + ", "
            + "but it's not a Receiver object.\n");
    } else {
        status.appendText("Found applet named "
            + receiverName + ".\n"
            + "    Sending message to it.\n");
        //Fuerza el tipo del Receptor a un objeto Receiver
        //(en vez de sólo un objeto applet) para que el compilador
        //nos permita llamar a un método del Receiver.
        ((Receiver)receiver).processRequestFrom(myName);
    }
} . . .
```

Desde el punto de vista de un applet, su nombre es almacenado en un parámetro llamado NAME. Se puede obtener el valor del parámetro utilizando el método `getParameter()` de la clase `Applet`. Por ejemplo, el Remitente obtiene su propio nombre con el siguiente código:

```
myName = getParameter("NAME");
```

Para más información de la utilización de `getParameter()`, puedes ir a [Escribir el Código para Soportar Parámetros](#).

Los applets de ejemplo de esta página realizan una comunicación en un sólo sentido -- desde el Remitente al Receptor. Si se quiere que el Receptor también pueda enviar mensajes al Remitente, lo único que se tiene que hacer es darle al Remitente una referencia de sí mismo (`this`) al receptor. Por ejemplo:

```
((Receiver)receiver).startCommunicating(this);
```

Encontrar Todos los Applets de una Página: El método `getApplets()`

El método `getApplets()` devuelve una lista (una [Enumeración](#), para ser preciso) de todos los applets que hay en la página.

Por razones de seguridad, la mayoría de los navegadores y visualizadores de applets implementan `getApplets()` para que solo devuelva aquellos applets originarios del mismo servidor que el applet que llamó a `getApplets()`. Aquí tienes una simple lista de todos los applets que puedes encontrar en esta página:

Abajo tienes las partes relevantes del método que llama a `getApplets()`. (Aquí tienes el [programa completo](#).)

```
public void printApplets() {
    //Una enumeración que contiene todos los applets de esta página (incluyendo este)
    //a los que podemos enviar mensajes.
    Enumeration e = getAppletContext().getApplets();
    . . .
    while (e.hasMoreElements()) {
        Applet applet = (Applet)e.nextElement();
        String info = ((Applet)applet).getAppletInfo();
        if (info != null) {
            textArea.appendText("- " + info + "\n");
        } else {
            textArea.appendText("- " + applet.getClass().getName() + "\n");
        }
    }
}
```

} . . .

Ozito

Comunicarse con el Navegador

Muchos de los métodos de Applet y AppletContext envuelven alguna pequeña comunicación con el navegador o el visualizador de applets. Por ejemplo, los métodos `getDocumentBase()` y `getBase()` de la clase Applet obtienen información del navegador o el visualizador de applet sobre el applet y la página HTML de la que éste viene. El método `showStatus()` de la clase Applet le dice al navegador o visualizador que muestre un mensaje de estado. El método `getParameterInfo()` puede darle al navegador una lista con los parámetros que entiende el applet. Y por supuesto, el navegador o el visualizador pueden llamar a los métodos `init()`, `start()`, `stop()`, y `destroy()` del applet para informarle sobre los cambios de su estado.

También son interesantes los métodos `showDocument` de AppletContext. Con estos métodos, un applet puede controlar la dirección URL que mostrará el navegador, y en que ventana del navegador. (Por supuesto, el Applet Viewer ignora estos métodos, ya que no es un navegador de Web). Aquí tienes dos formas de `showDocument()`:

```
public void showDocument(java.net.URL url)
public void showDocument(java.net.URL url, String targetWindow)
```

La forma de un sólo argumento de `showDocument()` sólo le dice al navegador que muestre la URL especificada, sin especificar la ventana en la que se mostrará el documento.

La forma de dos argumentos de `showDocument()` le permite especificar la ventana o marco HTML se mostrará el documento. El segundo argumento debe ser uno de los valores mostrados abajo.

Nota de Terminología: En esta explicación, marco no se refiere a un marco del AWT sino a un marco HTML dentro de una ventana del navegador.

"_blank"

Muestra el documento en una nueva ventana sin nombre.

"**windowName**"

Muestra el documento en una ventana llamada `windowName`. Esta ventana se creará si es necesario.

"_self"

Muestra el documento en la ventana y marco que contiene el applet.

"_parent"

Muestra el documento en la ventana del applet en el marco padre del marco del applet. Si el marco del applet no tiene padre, esto tiene el mismo efecto que "_self".

"_top"

Muestra el documento en la ventana del applet pero en el marco de más alto nivel. Si el marco del applet tiene el nivel más alto, esto actúa igual que "_self".

Abajo hay un applet que te permite probar cada opción de las dos formas de `showDocument()`. Trae una ventana que le permite teclear una URL y elegir una de las opciones de `showDocument()`. Cuando pulses la tecla return o el botón Show document, el applet llama al método `showDocument()`.

Abajo tienes el código del applet que llama a `showDocument()`. (Aquí tienes el [programa completo](#))

```
...//En una subclase de Applet:
    urlWindow = new URLWindow(getAppletContext());
    . . .
```

```

class URLWindow extends Frame {
    . . .
    public URLWindow(AppletContext appletContext) {
        . . .
        this.appletContext = appletContext;
        . . .
    }
    . . .
    public boolean action(Event event, Object o) {
        . . .
        String urlString = /* Cadena introducida por el usuario */;
        URL url = null;
        try {
            url = new URL(urlString);
        } catch (MalformedURLException e) {
            ...//Informa al usuario y retorna...
        }

        if (url != null) {
            if (/* el usuario no quiere especificar una ventana */) {
                appletContext.showDocument(url);
            } else {
                appletContext.showDocument(url, /* user-specified window */);
            }
        }
        . . .
    }
}

```


Trabajar con una aplicación del Lado del Servidor

Los applets, al igual que otros programas Java, pueden utilizar el API definido en el paquete `java.net` para comunicarse a través de la red. La única diferencia es que, por razones de seguridad, el único servidor con el que se puede comunicar un applet es el servidor desde el que vino.

Es fácil encontrar el servidor del que vino un applet. Sólo se debe utilizar el método `getCodeBase()` del `Applet` y el método `getHost()` de `java.net.URL`, de esta forma:

```
String host = getCodeBase().getHost();
```

Si se especifica un nombre de servidor incluso sólo ligeramente diferente del especificado por el usuario del applet, se corre el riesgo de que el manejador de seguridad corte la comunicación incluso si los dos nombres especifican el mismo servidor. Utilizando el código anterior (en vez de teclear el nombre del servidor) se asegura que el applet utiliza el nombre de servidor correcto.

Una vez que se tiene el nombre correcto, se puede utilizar todo el código de red que está documentado en [Red de Cliente y Seguridad](#).

Restricciones de Seguridad

Uno de los objetivos principales del entorno Java es hacer que los usuarios de navegadores se sientan seguros cuando ejecutan cualquier applet. Para conseguir este objetivo, hemos empezado conservadores, restringiendo las capacidades, quizás más necesarias. Cuando pase el tiempo los applets probablemente tendrán más y más capacidades.

Esta página cuenta las restricciones de seguridad actuales en los applets, desde el punto de vista del cómo afecta el diseño de applets.

Cada visualizador de Applets tiene un objeto `SecurityManager` que comprueba las violaciones de seguridad de un applet. Cuando el `SecurityManager` detecta una violación, crea y lanza un objeto `SecurityException`. Generalmente, el constructor de la `SecurityException` imprime un mensaje de aviso en la salida estandar. Un applet puede capturar esa excepción y reaccionar de forma apropiada, como tranquilizar al usuario y saltando a una forma "segura" (pero menos ideal) para realizar la tarea,

Algunos visualizadores de applets se tragan algunas `SecurityException`, para que el applet nunca pueda capturarlas. Por ejemplo, la implementación de los métodos `getApplet()` y `getApplets()` del `AppletContext` del Applet Viewer del JDK simplemente capturan e ignoran cualquier `SecurityException`. El usuario puede ver un mensaje de error en la salida estandar, pero al menos el applet obtiene un resultado válido desde los métodos. Esto tiene algúnsentido, ya que `getApplets()` debería poder devolver cualquier applet válido que encontrara, incluso si encuentra uno no válido. (El Applet Viewer considera que un applet es válido si se ha cargado desde el mismo host que el applet que llamo a `getApplets()`.)

Para aprender más sobre los controladores de seguridad y las clases de violaciones de seguridad que pueden comprobar, puedes ver [Introducción a los Manejadores de Seguridad](#).

Cómo se menciona en la lección anterior, los visualizadores de applets existentes (incluyendo los navegadores de la Web como Netscape Navigator 2.0) imponen las siguientes restricciones:

Los Applets no pueden cargar librerías ni definir métodos nativos.

Los applets sólo pueden utilizar su propio código Java y el API Java que le proporciona el visualizador. Como mínimo, cada visualizador de applets debe proporcionar acceso al API definido en los paquetes `java.*`.

Un applet no puede leer o escribir ficheros de forma ordinaria en el host donde se está ejecutando

El Applet Viewer del JDK permite algunas excepciones especificadas por el usuario a esa regla, pero Netscape Navigator 2.0 no lo permite. Los applets en cualquier visualizador pueden leer ficheros especificados con una dirección

URL completa, en vez por un nombre de fichero. Un atajo para no tener que escribir ficheros es hacer que el applet envíe los datos a una aplicación en el servidor de donde el es original. Esta aplicación puede escribir ficheros de datos en su propio servidor. Puedes ver [Trabajar con una Aplicacion del Lado del Servidor](#) para ver más ejemplos.

Un applet no puede hacer conexiones de red excepto con el host del que el applet es original

El atajo para esta restricción es hacer que el applet trabaje con una aplicación en el host del que vino. La aplicación puede hacer conexiones a cualquier lugar de la red.

Un applet no puede arrancar ningún programa en el host donde se está ejecutando

De nuevo, el applet puede trabajar con una aplicación en el lado del servidor en su lugar.

Un applet no puede leer todas las propiedades del sistema.

Puedes ver [Leer las Propiedades del Sistema](#) para obtener más información.

Las ventanas que genera una applet son distintas a las ventanas que genera una aplicación.

Las ventanas de applet tiene algún texto de aviso y una barra coloreada o una imagen. Esto ayuda al usuario a distinguir las ventanas de los applets de las ventanas de las aplicaciones verdaderas.

Capacidades de los Applets

La página anterior te podría haber hecho sentir que los applets son sólo aplicaciones mutiladas. No es cierto! Junto con la característica obvia de que los applets se pueden cargar a través de la red, los applets tienen más capacidades de las que podrías imaginar. Tienen acceso al API de todos los paquetes java.* (sujeto a las restricciones de seguridad) además tienen algunas capacidades que las aplicaciones no tienen.

Capacidades que las Aplicaciones no tienen

Los applets tienen capacidades extras porque están soportados por el código de la aplicación donde se están ejecutando. Los applets tienen acceso a este soporte a través del paquete java.applet, que contiene la clase Applet y los interfaces AppletContext, AppletStub, y AudioClip.

Aquí tienes algunas capacidades de los applets que las aplicaciones no tienen:

Los applets pueden ejecutar sonidos.

Puedes ver [Ejecutar Sonidos](#) para más información.

Los Applets que se ejecutan dentro de un Navegador pueden hacer fácilmente que se visualicen documentos HTML.

Esto está soportado por los métodos showDocument() de AppletContext. Puedes ver [Comunicarse con el Navegador](#) para obtener más información.

Los Applets pueden invocar a los métodos públicos de otros applets de la misma página

Puedes ver [Enviar Mensajes a Otros Applets en la Misma Página](#) para más información.

Más Capacidades de los Applets

Junto con las capacidades anteriores los applets tienen otras que podrías no esperar:

Los applets cargados desde un directorio del sistema local de ficheros (desde un directorio en el CLASSPATH del usuario) no tienen ninguna de las restricciones que tienen los applets cargados a través de la red.

Esto es porque los applets que están en el CLASSPATH del usuario se convierten en parte de la aplicación cuando son cargados.

Aunque la mayoría de los applets detienen su ejecución una vez que se ha salido de la página, no tienen por qué hacerlo.

La mayoría de los applets, para ser educados, implementan el método `stop()` (si es necesario) para parar cualquier proceso cuando el usuario abandona la página. Sin embargo, algunas veces, es mejor que continúe la ejecución del applet. Por ejemplo, si el usuario le dice al applet que realice un cálculo complejo, el usuario podría querer el cálculo continuase. (Aunque el usuario deberá poder especificar si quiere que el applet continúe o no). Otro ejemplo, si un applet pudiera ser útil durante varias páginas, debería utilizar una ventana para su interface (y no ocultar la ventana en su método `stop()`). El usuario puede cerrar la ventana cuando ya no la necesite.

Antes de Exportar un Applet

Stop!

Antes de permitir que el mundo entero conozca tu applet, asegurate de responder si a todas las siguientes cuestiones:

1. ¿Has eliminado o desactivado todas las salidas de depurado?

Las salidas de depurado (creadas generalmente con `System.out.println()`), son útiles para ti pero generalmente confunden o molestan a los usuarios. Si intentas darle realimentación textual al usuario, intenta hacerlo dentro del área del applet o en el área de estado en la parte inferior de la ventana. La información sobre cómo utilizar el área de estado puedes encontrarla en [Mostrar Cadenas Cortas de Estado](#).

2. ¿El applet detiene su ejecución cuando sale de la pantalla?

La mayoría de los applets no deberían utilizar recursos de la CPU cuando el navegador está minimizado o mostrando una página que no contiene un applet. Si el código del applet no lanza explícitamente ningún thread, entonces está correcto.

Si el applet lanza algún thread, entonces a menos que tengas una excusa REALMENTE BUENA para no hacerlo, deberas implementar el método `stop()` para que se paren y se destruyan (mediante su selección a null) los threads que has lanzado. Para ver un ejemplo de implementación del método `stop()` puedes ir a [Ejemplos de Threads en Applets](#).

3. Si el applet hace algo que pudiera resultar molesto -- ejecutar sonidos o animaciones, por ejemplo -- ¿Le has proporcionado al usuario alguna forma de parar el comportamiento molesto?

Se amable con tus usuarios. Dale una oportunidad a tus usuarios para que paren el applet sin tener que abandonar la página. En un applet que de otra forma no responde a las pulsaciones del ratón, se puede hacer esto mediante la implementación del método `mouseDown()` para que un click del ratón suspenda el thread molesto. Por ejemplo:

```
boolean frozen = false;           //Una variable de ejemplar

public boolean mouseDown(Event e, int x, int y) {
    if (frozen) {
        frozen = false;
        start();
    } else {
        frozen = true;
    }
}
```

```
        stop();  
    }  
    return true;  
}
```

Ozito

El Applet Finalizado Perfectamente

La página anterior lista algunas formas con las que se podría evitar que el usuario de tus applets quisieran torturarlo. Esta página cuenta algunas cosas más para hacer que sean tan placenteros como sea posible.

Haz tus Applets tan flexibles como sea posible.

Puedes definir parámetros que permitan que tu applet sea utilizado en una gran variedad de situaciones sin tener que re-escribirlo. Puedes ver [Definir y Utilizar Parámetros en un Applet](#) para obtener más información.

Implementa el método `getParameterInfo()`

Implementando este método puedes hacer que tu applet sea más fácil de personalizar en el futuro. Actualmente, ningún navegador utilizar este método. Sin embargo, se espera que pronto los navegadores utilicen este método para ayudar a generar un GUI para permitir que el usuario interactue con los parámetros. Puedes ver [Dar Información sobre los Parámetros](#) para más información sobre la implementación del método `getParameterInfo()`.

Implementa el método `getAppletInfo()`.

Este método devuelve una cadena corta descriptiva del applet. Aunque los navegadores actuales no utilizan este método, se espera que lo hagan en un futuro. Aquí tienes un ejemplo de implementación de `getAppletInfo()`:

```
public String getAppletInfo() {  
    return "GetApplets by Kathy Walrath";  
}
```


Problemas más Comunes con los Applets (y sus Soluciones)

Problema: El AppletViewer dice que no hay etiqueta applet en tu página HTML, pero si existe:

- Comprueba si has cerrado la etiqueta applet: `</APPLET>`.

Problema: He recompilado mi applet, pero mi visualizador no muestra la nueva versión, incluso si le digo que lo recargue.

- En muchos visualizadores (incluyendo los navegadores) la recarga de applets no es posible. Esto es por lo que sólo debes utilizar el Applet Viewer del JDK, llamándolo cada que cambies un applet.
- Si obtienes una versión vieja del applet, sin importar lo que haga, asegurate de que no tienes una copia vieja del applet en algún directorio de tu CLASSPATH. Puedes ver [Controlando los Paquetes](#) para más información sobre la variable de entorno CLASSPATH.

Problema: El fondo gris de mi applet hace que este parpadee cuando se dibuja sobre una página con un color diferente.

- Necesitas seleccionar el color de fondo del applet para que trabaje bien con el color de la página. Puedes ver [Crear un Interface Gráfico de Usuario \(GUI\)](#) para más detalles.

Problema: El método getImage del applet no funciona

- Asegurate de que has llamado a getImage desde el método init o desde un método llamado después de init. El método getImage no funciona cuando es llamado desde un constructor.

Problema: ahora que he copiado mi fichero .class del applet en mi servidor HTTP, el applet no funciona.

- ¿Tu applet ha definido más de una clase? Si lo ha hecho, asegurate de que todos los ficheros .class (**ClassName.class**) de todas las clases están en el servidor HTTP. Incluso si todas las clases están definidas en el mismo fichero fuente, el compilador produce un fichero .class para cada clase.
- ¿Has copiado todos los ficheros de datos de tu applet -- ficheros de imágenes o sonidos, por ejemplo -- al servidor?
- Asegurate que todas las clases del applet y todos los ficheros de datos pueden ser leídos por todos (comprueba los permisos).

Cambios del API que afectan a los Applets

Con la ayuda del atributo ARCHIVE, puedes decirle a los navegadores que carguen tus applets desde ficheros de archivo Java (ficheros JAR). Utilizar los ficheros JAR puede reducir significativamente el tiempo de descarga del Applet y ayudarte a evitar algunas restricciones de seguridad innecesarias.

Puedes ver [Cambios en 1.1: Archivos JAR y los Applets](#).

Ozito

Introducción al UI de Java

Esta lección ofrece una introducción a todo lo que le proporciona el entorno Java para ayudarte a crear un interface de usuario (UI). UI es un término que se refiere a todos los caminos de comunicación entre un programa y sus usuarios. UI no es sólo lo que ve el usuario, también es lo que el usuario oye y siente. Incluso la velocidad con la que un programa interactúa con el usuario es una parte importante del UI del programa.

El entorno Java proporciona clases para las siguientes funcionalidades del UI:

Presentar un UI gráfico (GUI)

Este es el UI preferido por la mayoría de los programas Java. El resto de esta sección se concentra sobre este punto.

Ejecutar Sonidos

Justo ahora, los applets pueden ejecutar sonidos, pero las aplicaciones no pueden (al menos de una forma portable). Puedes ver [Ejecutar Sonidos](#) para más información sobre la ejecución de sonidos en los applets.

Obtener información de configuración

Los usuarios pueden configurar la información del applet utilizando argumentos de la línea de comandos (sólo las aplicaciones) y parámetros (sólo los applets). Para más información sobre los argumentos de la línea de comandos, puede ver [Argumentos de la Línea de Comandos de una Aplicación](#). Para más información sobre los parámetros, puede ver [Definir y Utilizar Parámetros en un Applet](#).

Grabar las preferencias del usuario utilizando propiedades

Para la información que las aplicaciones necesitan guardar cuando no se están ejecutando, puedes utilizar las propiedades. Normalmente los applets no pueden escribir propiedades en el sistema local de ficheros, debido a las restricciones de seguridad. Para obtener más información sobre las propiedades puede ver [Propiedades](#).

Obtener y mostrar texto utilizando los canales de entrada, salida y error estandar

Los canales de entrada, salida y error estandar son una forma al viejo estilo de presentar un interface de usuario. Todavía es útil para probar y depurar programas, así como para alguna funcionalidad no dirigida al usuario típico. Puedes ver [Los Canales de I/O Estandar](#) para obtener información sobre la utilización de los canales de entrada, salida y error estandar.

Los applets y las aplicaciones presentan información al usuario y le invitan a interactuar utilizando un GUI. La parte del entorno Java llamada Herramientas de Ventanas Abstractas (Abstract Windows Toolkit - AWT) contiene un completo conjunto de clases para escribir programas GUI. .

Componentes de AWT

El AWT proporciona muchos componentes GUI standards, como botones, listas, menús y áreas de texto. También incluye contenedores (como ventanas y barras de menú) y componentes de alto nivel (cómo un cuadro de diálogo para abrir y guardar ficheros).

Otras Clases AWT

Otras clases del AWT incluyen aquellas que trabajan en un contexto gráfico (incluyendo las operaciones de dibujo básico), imágenes, eventos, fuentes y colores. Otro grupo importante de clases del AWT son los controladores de distribución o disposición que controlan el tamaño y la posición de los componentes.

La Anatomía de un Programa Basado en GUI

El AWT proporciona un marco de trabajo para dibujo y manejo de eventos. Utilizando un programa especificando la herencia de los contenedores y los componentes, el AWT envía eventos (como pulsaciones del ratón) al objeto apropiado. La misma herencia determina cómo se dibujarán a sí mismos los contenedores y los componentes.

Componentes del AWT

El applet de esta página muestra los componentes gráficos del GUI proporcionados por el AWT. Con la excepción de los menús, todas los componentes GUI son implementados con una subclase de la clase [Component](#) del AWT.

Nota de Implementación: El applet está implementado como un botón que trae una ventana que muestra los componentes. La ventana es necesaria porque el programa incluye un menú, y los menús sólo se pueden utilizar en las ventanas. Para los curiosos, aquí está el [código fuente](#) de la ventana que muestra los componentes. El programa tiene un método `main()` por lo que se puede ejecutar como una aplicación. La clase [AppletButton](#) proporciona un marco de trabajo en el applet para la ventana. `AppletButton` es un applet altamente configurable que se explicó en las siguientes páginas: [Decidir los Parámetros a Soportar](#) y [Escribir el Código para Soportar Parámetros](#).

Los controles básicos: botones, checkbox, choices, listas, menús y campos de texto

Las clases [Button](#), [Checkbox](#), [Choice](#), [List](#), [MenuItems](#), y [TextField](#) proporcionan los controles básicos. Estas son las formas más comunes en que el usuario da instrucciones al programa Java. Cuando un usuario activa uno de estos controles -- pulsa un botón o presiona la tecla return en un campo de texto, por ejemplo -- envía un evento (`ACTION_EVENT`). Un objeto que contiene el control puede reaccionar al evento implementando el método `action()`.

Otras formas de obtener entradas del usuario: Barras de Desplazamiento y Áreas de Texto.

Cuando los controles básicos no son apropiados, puede utilizar las clases [Scrollbar](#) y [TextArea](#) para obtener la entrada del usuario. La clase `Scrollbar` se utiliza tanto para deslizadores como para barras de desplazamiento. Puedes ver los deslizadores en [La Anatomía de un Programa Basado en GUI](#). Puedes ver las barras de desplazamiento en las listas y áreas de texto en el applet de esta página.

La clase `TextArea` sólo proporciona un área en la que mostrar o permitir editar varias líneas de texto. Como puedes ver en el applet de esta página, las áreas de texto incluyen automáticamente barras de desplazamiento.

Crear Componentes del Cliente: Lienzos

La clase [Canvas](#) permite escribir componentes personalizados. Con la subclase de Canvas, se puede dibujar un gráfico de usuario en la pantalla -- en un programa de dibujo, un procesador de imágenes o un juego, por ejemplo -- e implementar cualquier clase de manejo de eventos.

Etiquetas

La clase [Label](#) sólo muestra una línea de texto no editable por el usuario.

Contenedores: Ventanas y Paneles

El AWT proporciona dos tipos de contenedores, ambos son implementados como subclases de [Container](#) (que es una subclase de [Component](#)). Las subclases de Windows -- [Dialog](#), [FileDialog](#), y [Frame](#) -- proporcionan ventanas para contener componentes. La clase [Frame](#) crea ventanas normales y completamente maduras, como oposición a las creadas por la clase [Dialogs](#), que son dependientes del marco y pueden ser modales. Los Paneles agrupan los componentes dentro de un área de una ventana existente.

El programa de ejemplo al principio de esta sección utiliza un Panel para agrupar la etiquetas y el área de texto, otro Panel para agruparlo con un lienzo, y un tercer panel para agrupar el campo de texto, el checkbox y una lista de opciones desplegable. Todos estos paneles están agrupados por un objeto [Frame](#), que representa la ventana en la que estos se muestran. El Marco también contiene un menú y una lista.

Cuando seleccione la opción "File dialog..." en el menú, el programa crea un objeto [FileDialog](#) que es un cuadro de diálogo que puede servir para Abrir o Guardar ficheros.

Nota del Navegador: Netscape Navigator 2.0 no implementa la clase [FileDialog](#), ya que no permite nunca leer o escribir ficheros en el sistema de ficheros local. En vez de ver el cuadro de diálogo verá un mensaje de error en la Consola Java.

Sumario

Esta página representa una visión relámpago sobre los componentes del AWT. Cada componente mencionado en esta página se describe con más detalle en [Utilizar Componentes, los Bloques de Construcción del GUI](#).

Otras Clases del AWT

El AWT contiene algo más que componentes. Contiene una variedad de clases relacionadas con el dibujo y el manejo de eventos. Esta sección explica las clases del AWT que están en el paquete `java.awt`. El AWT contiene otros dos paquetes -- `java.awt.image` y `java.awt.peer` -- que la mayoría de los programas no tendrán que utilizar.

Como has aprendido en la página anterior, los componentes se agrupan dentro de contenedores. Lo que la página anterior no te contó es que cada contenedor utiliza un controlador de disposición para controlar el tamaño y posición de los componentes sobre la pantalla. El paquete `java.awt` suministra varias clases de controladores de disposición. Podrás aprender más sobre los controladores de disposición en la lección [Distribuyendo los Componentes dentro de un Contenedor](#).

El paquete `java.awt` suministra varias clases para representar tamaños y formas. Un ejemplo es la clase `Dimension`, que especifica el tamaño de un área rectangular. Otra clase es `Inset` que normalmente se utiliza para especificar cuanto espacio debe quedar entre el borde de un contenedor y el área de trabajo del propio contenedor. Las clases de formas incluyen `Point`, `Rectangle` y `Polygon`.

La clase `Color` es útil para la representación y manipulación de colores. Define constantes para los colores más utilizados -- por ejemplo, `Color.Black`. Generalmente utiliza colores en formato RGB (rojo-verde-azul) pero también entiende el formato HSB (hue-saturation- brightness).

La clase `Image` proporciona una forma de representar una imagen. Los applets pueden obtener objetos `Image` para formatos GIF y JPEG utilizando el método `getImage()` de la clase `Applet`. Las aplicaciones pueden obtener imágenes utilizando otra clase : `Toolkit`. Esta clase proporciona in interface independiente de la plataforma para la implementación del AWT dependiente de la plataforma. Aunque no suene muy expresivo, la mayoría de los programas no tratan directamente con objetos `Toolkit`, excepto para obtener imágenes. Las imágenes se cargan asíncronamente -- se puede tener un objeto `Image` válido incluso si los datos de la imagen no se han cargado todavía (o incluso no existen). Utilizando un objeto `MediaTracker`, se puede seguir la pista al estado de carga de una imagen. `MediaTracker` realmente sólo trabaja con imágenes, pero eventualmente podremos hacer que trabaje con otros tipos de medios, como sonidos. [Utilizar Imágenes](#) cuenta todo sobre el trabajo con imágenes.

Para controlar el aspecto del texto en los programas de dibujo, se pueden utilizar objetos `Font` y `FontMetrics`. La clase `Font` permite obtener información básica sobre la fuente y crear objetos para representar varias fuentes. Con un objeto `FontMetrics`, se puede obtener información detallada sobre las características de tamaño de una fuente determinada. Se puede seleccionar la fuente utilizada por un componente utilizando los métodos `setFont()` de las clases `Component` y

Graphics. [Trabajar con Texto](#) cuenta todo sobre la utilización de fuentes.

Finalmente, las clases Graphics y Event son cruciales para el dibujo y el manejo de eventos del sistema en el aWT. Un objeto Graphics representa un contexto de dibujo -- sin un objeto Graphics, ningún programa puede dibujar en la pantalla. Un objeto Event representa una acción del usuario, como una pulsación del ratón. Aprenderás más sobre los objetos Graphics y Event más adelante en esta sección de introducción.

Ozito

La Anatomía de un Programa Basado en el GUI

Esta página y la que le sigue son parte de un sencillo programa Java que tiene un UI gráfico, y que explican:

- La clases utilizadas por el programa.
- El árbol de herencia de los Componentes del Programa
- Cómo se dibujan los propios Componentes
- Cómo se propagan los eventos a través del árbol de herencia.

El programa convierte medidas entre el sistema métrico y el sistema americano. Para los curiosos, aquí está el [código fuente](#). No esperamos que entiendas completamente el código fuente sin leer el resto de esta lección y las páginas más relevantes en las lecciones [Utilizar Componentes, los Bloques de Construcción del GUI](#) y [Distribuir Componentes dentro de un Contenedor](#). Aquí tienes el programa, ejecutándose como un applet.

Las Clases del Programa de Ejemplo

El programa de ejemplo define tres clases y crea ejemplares de varias clases proporcionadas por el AWT. Define una subclase de Applet para poder ejecutarse como un applet. Crea Componentes para proporcionar un control básico para que el usuario pueda interactuar con él. Utilizando Containers y LayoutManager agrupa los Componentes.

El árbol de herencia de los componentes

Los componentes del programa están ordenados en forma de árbol, con contenedores que definen la estructura del árbol.

Dibujo

Los componentes se dibujan desde la parte superior del árbol -- la ventana del programa -- hasta los componentes sin contenedor.

Manejo de Eventos

Las acciones del usuario resultan en eventos, que son pasados a través del árbol de componentes hasta que un objeto responda al evento.

Las Clases del Programa de Ejemplo

El [programa de ejemplo](#) define dos clases que descienden de las clases del AWT. También define una clase para almacenar datos. Sin embargo, la mayoría de los objetos del programa son ejemplares de las clases del AWT.

Esta página recuenta todos los objetos que son creados en el programa. No te preocupes -- no esperamos que lo entiendas todo aún. Sólo queremos explicarte la clase de objetos que un programa GUI podría utilizar.

Las Clases definidas en el programa de ejemplo

El programa define dos subclases de Panel; Converter y ConversionPanel y una clase sencilla llamada Unit.

La clase Converter es el corazón del programa de ejemplo. Contiene el método main() del programa (que es llamado si el programa se utiliza como una aplicación), y así como el código de inicialización y de arranque, que es llamado por el método main() o por la aplicación que carga el programa como un applet. La clase Converter realmente desciende de la clase Applet (que a su vez desciende de la clase Panel) en lugar de descender directamente de la clase Panel. Esto es necesario porque todos los applets deben contener una subclase de Applet. Sin embargo, como el programa de ejemplo también se puede ejecutar como una aplicación, la clase Converter no debe utilizar ninguna de las funcionalidades proporcionadas por la clase Applet. En otras palabras, la clase Converter debe implementarse como si descendiera de la clase Panel.

La clase ConversionPanel proporciona una forma de agrupar todos los controles que definen un conjunto particular de distancias. El programa de ejemplo crea dos objetos ConversionPanel, uno para las medidas en el sistema métrico y otro para las medidas en el sistema americano.

La clase Unit proporciona objetos que agrupan una descripción (como "Centímetros") con un multiplicador que indica el número de unidades por metro (0,01 por ejemplo).

Objetos del AWT en el Programa de Ejemplo

El programa de ejemplo utiliza varios LayoutManagers, Containers, y Components proporcionados por el paquete AWT. También crea dos objetos Insets y dos objetos GridBagConstraints.

El programa de ejemplo crea tres objetos que conforman el interface Layoutmanager: un GridLayout y dos GridBagLayout. El GridLayout controla la disposición de los componentes en el ejemplar de Converter.

Cada `ConversionPanel` utiliza un objeto `GridBagLayout` para manejar sus componentes, y un objeto `GridBagConstraints` para especificar como colocar cada uno de los componentes.

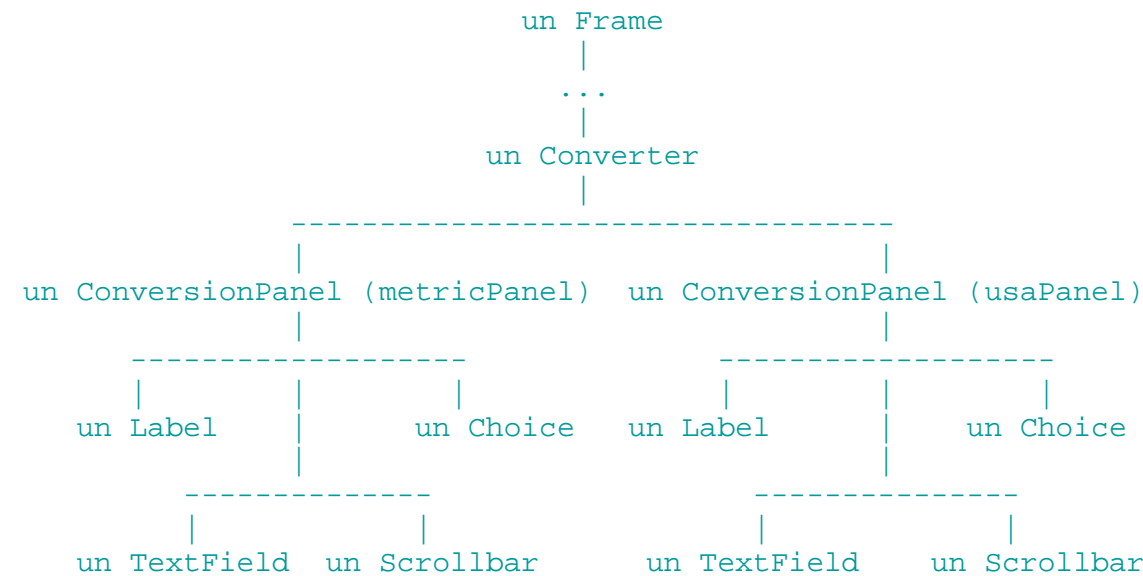
Junto con los objetos `Converter` y `ConversionPanel`, el programa crea un contenedor más. Específicamente, si el programa se ejecuta como una aplicación (en vez de como un applet), crea un objeto `Frame` (una ventana independiente).

Todos los componentes que no son contenedores en el programa ejemplo son creados por `ConversionPanel`. Cada `ConversionPanel` contiene un ejemplar de las clases `Label`, `Choice`, `TextField`, y `Scrollbar` del AWT.

Tanto la clase `Converter` como la `ConversionPanel` crean ejemplares de `Insets` que especifican el espacio que debe aparecer entre sus representaciones en la pantalla.

El Arbol de Componentes

El Programa de Ejemplo tiene varios niveles de herencia en el árbol de componnetes. El padre de cada nivel es un contenedor (que desciende de Component). Abajo tienes una figura de la herencia. >



Explicación

En la parte superior de la herencia esta la ventana (ejemplar de Frame) que muestra el programa. Cuando el programa de ejemplo se ejecuta como una aplicación, el Marco se crea dentro del método main(). Cuando el ejemplo se ejecuta como un applet dentro de un navegador, el Marco es la ventana del navegador.

Debajo del Frame esta un objeto Converter, que desciende de la clase Applet y que es un Contenedor (específicamente, un Panel). Dependiendo el visualizador en que se esté mostrando el applet, podría haber uno o más contenedores entre el objeto Converter y el Frame de la parte superior del árbol de herencia.

Directamente debajo del objeto Converter hay dos ConversionPanel. El siguiente código los pone debajo del Converter, utilizando el método add(). La clase Converter hereda el método add() de la clase Container (Converter desciende de la clase Applet, que a su vez desciende de la clase Panel, que a su vez desciende de la clase Container).

```
public class Converter extends Applet {
    . . .
    public void init() {
        ...//Crea metricPanel y usaPanel, que son dos ConversionPanels.
        add(metricPanel);
        add(usaPanel);
        . . .
    }
}
```

Cada ConversionPanel tiene cuatro hijos: un Label, un TextField, un Scrollbar, y un Choice. Aquí tienes el código para añadir estos niños:

```
class ConversionPanel extends Panel {
    . . .
    ConversionPanel(Converter myController, String myTitle, Unit myUnits[]) {
        . . .
        //Añade la etiqueta. Muestra este título de panel, centrado
        Label label = new Label(myTitle, Label.CENTER);
        ...//Selecciona GridBagConstraints para este componente Component.
        gridbag.setConstraints(label, c);
        add(label);
    }
}
```

```

//Añade el campo de texto. Inicialmente muestra "0" y necesita ser de al menos
//10 caracteres de ancho.
textField = new TextField("0", 10);
...//Selecciona GridBagConstraints para este Component.
gridbag.setConstraints(textField, c);
add(textField);

//Añade la lista desplegable (Choice).
unitChooser = new Choice();
...//Genera los items de la lista.
...//Selecciona GridBagConstraints para este Component.
gridbag.setConstraints(unitChooser, c);
add(unitChooser);

//Añade la barra deslizador. Es horizontal, su valor inicial es 0.
//un click incrementa el valor 100 pixels, y tiene especificados los valores
//mínimo y máximo especificados por las variables del ejemplar min y max.
slider = new Scrollbar(Scrollbar.HORIZONTAL, 0, 100, min, max);
...//Selecciona GridBagConstraints para este Component.
gridbag.setConstraints(slider, c);
add(slider);
}

```

GridBagConstraints es un objeto que le dice a GridBagLayout (El controlador de disposición para cada ConversionPanel) como colocar un componente particular. GridBagLayout, junto con los otros controladores de disposición del AWT se explican en [Distribuir Componentes dentro de un Contenedor](#).

Sumario

La herencia de Componentes del programa de ejemplo contiene ocho componentes que no son contenedores -- componentes que representan el UI gráfico del programa. Estos son las etiquetas, los campos de texto, las elecciones y las barras de desplazamiento que muestra el programa. Podría haber componentes adicionales como controles de ventana bajo el marco.

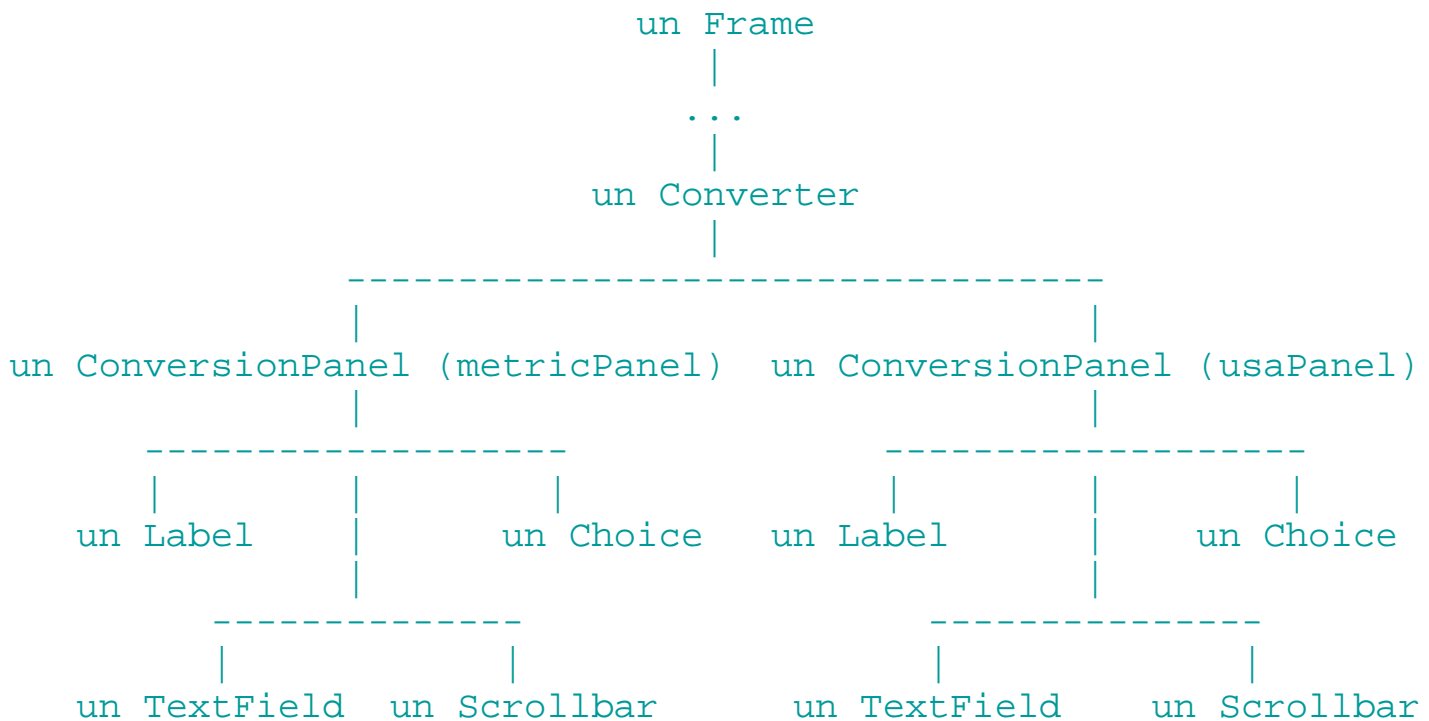
Este árbol de componentes tiene al menos cuatro contenedores -- un Frame (ventana), un Converter (una clase cliente de Panel), y dos ConversionPanels (otros dos Paneles del Cliente).

Observa que si añadimos una ventana -- por ejemplo, un Frame que contiene un ejemplar de Converter que maneje la conversión de distancias -- la nueva ventana tendrá su propio árbol de herencia, totalmente independiente del árbol de herencia que presenta esta lección.

Dibujo de Componentes

Cuando un programa Java con un GUI necesita dibujarse a sí mismo -- si es la primera vez, o porque se ha vuelto visible o porque necesita cambiar su apariencia para reflejar algo que ha sucedido dentro del programa -- empieza con el componente más alto que necesita ser redibujado (por ejemplo, el componente superior en el árbol de la herencia) y va bajando hacia los componentes inferiores. Esto está orquestado por el sistema de dibujo del AWT.

Aquí tienes, como recordatorio, el árbol de herencia del programa conversor:



Aquí tiene lo que sucede cuando la aplicación Converter se dibuja a sí misma:

1. Primero se dibuja el Frame (marco).
2. Luego se dibuja el objeto Converter, dibujando una caja alrededor de su área.
3. Después se dibuja un de los dos ConversionPanels, dibujando una caja alrededor de su área.
Nota: No se puede contar con el orden en que se van a dibujar dos componentes con el mismo nivel. Por ejemplo, no se puede contar con que el panel metrico se dibujará antes que el americano. Similarmente, tampoco se puede depender del orden de dibujo de dos componentes de diferentes niveles si el componente inferior no está contenido en el componente superior.
4. Por último dibujan los contenidos de los ConversionPanel -- Label, TextField, Scrollbar, y Choice.

De esta forma, cada componente se dibuja a sí mismo antes que los componentes que contiene. Esto asegura que el fondo de un Panel, por ejemplo, sólo sea visible cuando no está cubierto por ninguno de sus componentes.

Cómo ocurre la petición de redibujado

Los programas sólo pueden dibujarse cuando el AWT les dice que lo hagan. La razón es que cada ocurrencia de que un dibujo se dibuje a sí mismo debe ejecutarse sin interrupción. De otra forma podrían producirse resultados impredecibles, como que un botón se dibujará sólo por la mitad, cuando fuera interrumpido por una animación lenta. El AWT ordena las peticiones de redibujado mediante su ejecución en un thread. Un componente puede utilizar el método `repaint()` para pedir que sea programado para redibujarse.

El AWT pide que un componente se redibuje llamando al método `update()` del componente. La implementación por defecto de este método sólo limpia el fondo del componente (dibujando un rectángulo sobre el área del componente con el color del fondo del propio componente) y luego llama al método `paint()` del componente. La implementación por defecto del método `paint()` no hace nada.

El Objeto Graphics

El único argumento para los métodos `paint()` y `update()` es un objeto `Graphics` que representa el contexto en el que el componente puede realizar su dibujo. La clase `Graphics` proporciona métodos para lo siguiente:

- Dibujar y rellenar rectángulos, arcos, líneas, óvalos, polígonos, texto e imágenes.
- Obtener o seleccionar el color actual, la fuente o el área de trabajo.
- Seleccionar el modo de dibujo.

Cómo dibujar

La forma más sencilla de que un componente pueda dibujarse es poner código de dibujo en su método `paint()`. Esto significa que cuando el AWT hace una petición de dibujo (mediante la llamada al método `update()` del componente, que está implementado como se describió [arriba](#)), se limpia todo el área del componente y luego se llama al método `paint()` del método. Para programas que no se redibujan a sí mismos de forma frecuente, el rendimiento de este esquema está bien.

Importante: Los métodos `paint()` y `update()` deben ejecutarse muy rápidamente! De otra forma, destruirán el rendimiento percibido del programa. Si se necesita realizar alguna operación lenta como resultado de una petición de dibujo, háglo arrancando otro thread (o enviando una petición a otro thread) para realizar la operación. Para obtener ayuda sobre la utilización de threads puedes ver [Threads de Control](#).

Abajo hay un ejemplo de implementación del método `paint()`. Las clases `Converter` y `ConversionPanel` dibujan una caja alrededor de su área utilizando este código. Ambas clases también implementan un método `insets()` que especifica el área libre alrededor del contenido de los paneles. Si no tuvieran este método, el método `paint()` solaparía los límites externos del panel.

```
public void paint(Graphics g) {  
    Dimension d = size();  
    g.drawRect(0,0, d.width - 1, d.height - 1);  
}
```

Los programas que se redibujan muy frecuentemente pueden utilizar dos técnicas para aumentar su rendimiento: implementar los dos métodos `update()` y `paint()`, y utilizar un doble buffer. Estas técnicas se explican en [Eliminar el Parpadeo](#).

Para más información sobre cómo dibujar, puedes ver la lección [Trabajar con Gráficos](#).

Manejo de Eventos

Cuando el usuario actúa sobre un componente -- pulsando el ratón o la tecla Return, por ejemplo -- se crea un objeto Event. El sistema manejador de eventos del AWT pasa el Evento a través del árbol de componentes, dando a cada componente una oportunidad para reaccionar ante el evento antes de que el código dependiente de la plataforma que implementan todos los componentes lo procese.

Cada manejador de eventos de un componente puede reaccionar ante un evento de alguna de estas formas:

- Ignorando el evento y permitiendo que pase hacia arriba en el árbol de componentes. Esto es lo que hace la implementación por defecto de la clase Component. Por ejemplo, como la clase TextField y su superclase TextComponent no implementan manejadores de eventos, los Campos de texto obtienen la implementación por defecto de la clase Component. Así cuando un TextField recibe un evento, lo ignora y permite que su contenedor lo maneje.
- Mediante la modificación del ejemplar de Event antes de dejarlo subir por el árbol de componentes. Por ejemplo, una subclase de TextField que muestra todas las letras en mayúsculas podría reaccionar ante la pulsación de una letra minúscula cambiando el Event para que contuviera la versión mayúscula de la letra.
- Reaccionando de alguna otra forma ante el evento. Por ejemplo, una subclase de TextField (o un contenedor de TextField) podrían reaccionar ante la pulsación de la tecla Return llamando a un método que procese el contenido del campo.
- Interceptando el evento, evitando un procesamiento posterior. Por ejemplo, un carácter no válido se ha introducido en un campo de texto, un manejador de eventos podría parar el evento resultante y evitar su propagación. Como resultado, la implementación dependiente de la plataforma del campo de texto nunca vería el evento.

Desde el punto de vista de un Componente, el sistema de manejo de eventos del AWT es como un sistema de filtrado de eventos. El código dependiente de la plataforma genera un evento, pero los Componentes tienen una oportunidad para modificar, reaccionar o interceptar el evento antes de que el código dependiente de la plataforma los procese por completo.

Nota: En la versión actual, los eventos del ratón se envían a los Componentes después de que los haya procesado el código dependiente de la plataforma. Por eso aunque los Componentes pueden interceptar todos los eventos del teclado, actualmente no pueden interceptar los eventos del ratón.

Aunque el AWT define una amplia variedad de tipos de eventos, el AWT no ve todo lo que ocurre. De este modo no todas las acciones del usuario se convierten en

eventos. El AWT sólo puede ver aquellos eventos que le deja ver el código dependiente de la plataforma. Por ejemplo, los campos de texto Motif no envían los movimientos del ratón al AWT. Por esta razón, las subclases de TextField y los contenedores no pueden contar con obtener los eventos de movimiento del ratón -- en Solaris, al menos, no hay una forma sencilla de conocer que ese evento se ha producido, ya que no recibe ningún evento cuando se mueve el ratón. Si se quiere acceder a un amplio rango de tipos de eventos se necesitará implementar una subclase de Canvas, ya que la implementación dependiente de la plataforma de la clase Canvas reenvía todos los eventos.

El Objeto Event

Cada evento resulta en la creación de un objeto [Event](#). Un objeto Event incluye la siguiente información:

- El tipo del evento -- por ejemplo, una pulsación de tecla o un click del ratón, o un evento más abstracto como "acción" o iconificación de una ventana.
- El objeto que fue la "fuente" del evento -- por ejemplo, el objeto Button correspondiente al botón de la pantalla que pulsó el usuario, o el objeto TextField en el que el usuario acaba de teclear algo.
- Un campo indicando el momento en que ocurrió el evento.
- La posición (x,y) donde ocurrió el evento. Esta posición es relativa al origen del Componente a cuyo manejador de eventos se pasó este evento.
- La tecla que fue pulsada (para eventos de teclado).
- Un argumento arbitrario (como una cadena mostrada en el Componente) asociada con el evento.
- El estado de las teclas modificadoras, como Shift o Control, cuando ocurrió el evento.

Cómo implementar un Manejador de Eventos

La clase Component define muchos métodos manejadores de eventos, y se puede sobrescribir alguno de ellos. Excepto el método utilizado para todos los propósitos `handleEvent()`, cada método manejador de eventos sólo puede ser utilizado para un tipo de evento particular. Recomendamos que se evite el método multi-propósito, si es posible, y en su lugar se sobrescriba el método de manejo de evento que está especificado por el tipo de evento que se necesita manejar. Esta aproximación tiende a tener menor número de efectos laterales adversos.

La clase Component define los siguientes métodos para responder a los eventos (el tipo de evento manejado se lista después del nombre del

método):

- `action()` (`Event.ACTION_EVENT`)
- `mouseenter()` (`Event.MOUSE_ENTER`)
- `mouseExit()` (`Event.MOUSE_EXIT`)
- `mousemove()` (`Event.MOUSE_MOVE`)
- `mousedown()` (`Event.MOUSE_DOWN`)
- `mouseDrag()` (`Event.MOUSE_DRAG`)
- `mouseup()` (`Event.MOUSE_UP`)
- `keyDown()` (`Event.KEY_PRESS` or `Event.KEY_ACTION`)
- `keyUp()` (`Event.KEY_RELEASE` or `Event.KEY_ACTION_RELEASE`)
- `gotFocus()` (`Event.GOT_FOCUS`)
- `lostFocus()` (`Event.LOST_FOCUS`)
- `handleEvent()` (all event types)

Cuando ocurre un evento, se llama al método manejador de evento que coincide con el tipo del evento. Específicamente, el Evento se pasa primero al método `handleEvent()`, que (en la implementación por defecto de `handleEvent()`) llama al método apropiado por el tipo de evento.

El método `action()` es un método especialista importante en el manejo de eventos. Sólo los componentes de control básicos -- `Button`, `Checkbox`, `Choice`, `List`, `MenuItem`, y `TextField` objects -- producen eventos `action`. Ellos lo hacen cuando el usuario indica de alguna forma que el control debería realizar alguna acción. Por ejemplo, cuando el usuario pulsa un botón, se genera un evento `action`. Mediante la implementación del método `action()`, se puede reaccionar a las acciones sobre los controles sin preocuparse por el eventos de bajo nivel, como la pulsación de tecla o el click del ratón que causó la acción.

Todos los métodos manejadores de eventos tienen al menos un argumento (el objeto `Event`) y devuelven un valor booleano. El valor de retorno indica si el método a manejado completamente el evento. Devolviendo `false`, el manejador indica que se debe continuar pasando el evento a través del árbol de componentes. Devolviendo `true`, el manejador indica que el evento no debe seguir pasandose. El método `handleEvent()` casi siempre deberá devolver `super.handleEvent()`, para asegurarse que todos los eventos sean pasados al método manejador de eventos apropiado.

Importante: Como los métodos de dibujo, todos los métodos manejadores de eventos deben ejecutarse rápidamente! De otra forma, destruirían el rendimiento percibido del programa. Si se necesita realizar alguna operación lenta como resultado de un evento, háglo arrancando

otro thread (o enviando una petición a otro thread) para que realice la operación. Puedes ver [Threads de Control](#).

En el programa de ejemplo, todo el manejo de eventos lo realizan los objetos `ConversionPanels`. Utilizan el método `action()` para capturar los eventos resultantes de las acciones del usuario los campos de texto (`TextField`), y las listas desplegables (`Choice`). Para capturar los eventos resultantes de la acción del usuario sobre las barras deslizantes (`Scrollbar`), deben utilizar el método `handleEvent()`, ya que los `Scrollbar`s no producen el evento `action` y la clase `Component` no define ningún método específico para los eventos de los objetos `Scrollbar`.

Aquí tienes la implementación de `ConversionPanel` para los métodos `action()` y `handleEvent()`:

```
/** Responde a las acciones del usuario sobre los controles. */
public boolean action(Event e, Object arg) {
    if (e.target instanceof TextField) {
        setSliderValue(getValue());
        controller.convert(this);
        return true;
    }
    if (e.target instanceof Choice) {
        controller.convert(this);
        return true;
    }
    return false;
}

/** Responde a la barra deslizable. */
public boolean handleEvent(Event e) {
    if (e.target instanceof Scrollbar) {
        textField.setText(String.valueOf(slider.getValue()));
        controller.convert(this);
    }
    return super.handleEvent(e);
}
```

Estos métodos simplemente se aseguran que la barra deslizante y el campo de texto de los `ConversionPanel` muestren el mismo valor, y le piden al objeto `Converter` que actualice el otro `ConversionPanel`. El método `action()` devuelve `true` si ha manejado el evento. Esto evita que el evento haga un viaje innecesario a través del árbol de componentes. Si el método `action()` no puede manejar el evento, devuelve `false`, para que algún componente superior en el árbol de componentes pueda echarle un vistazo al evento. El método `handleEvent()` siempre devuelve `super.handleEvent()` para que todos

los eventos sean completamente procesados.

Una Nota sobre el Método `action()`: Los eventos Action son eventos de alto nivel. Son causados por uno o más eventos de bajo nivel como una pulsación de tecla y de ratón. Por esta razón, es correcto devolver `true` para parar el evento acción antes de que siga subiendo por el árbol de componentes después de haberlo manejado -- el código dependiente de la plataforma ya ha manejado los eventos de teclas o del ratón que ha lanzado la acción, no necesita ver de nuevo el evento action.

Nota: Si `handleEvent()` devolviera `true` o `false` (en lugar de llamar a la implementación de su superclase), el método `action()` nunca sería llamado. Riesgos como este son una parte de la razón por la que hemos avisado para que se evite la utilización del método `handleEvent()` a menos que sea absolutamente necesario.

El Foco del Teclado

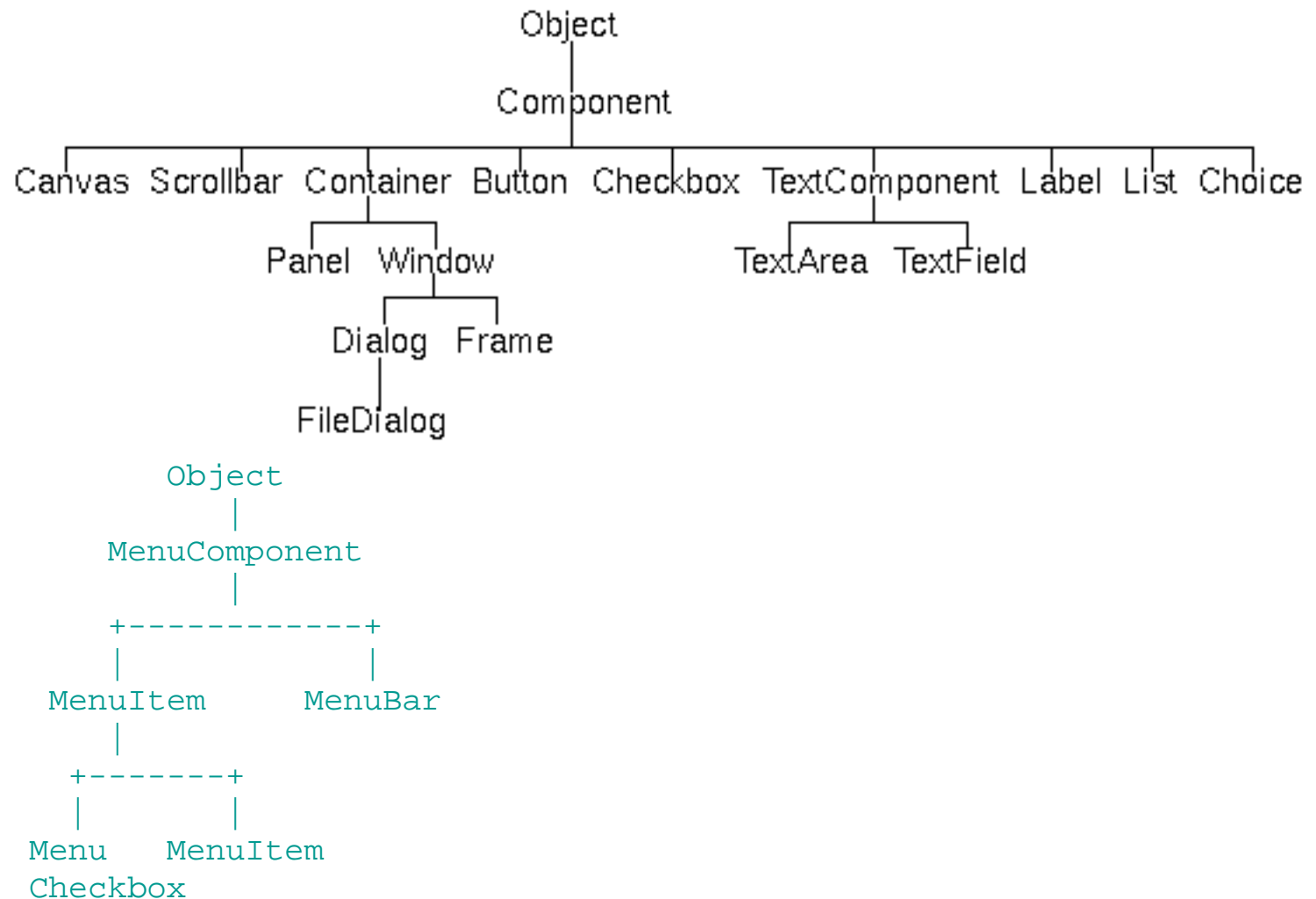
Muchos componentes --incluso que aquellos que operan primordialmente con el ratón, como los botones -- pueden operarse con el teclado. Para que tenga efecto una pulsación de tecla sobre un componente, este componente debe tener el foco del teclado.

En un momento dado, al menos una ventana y un componente de esa ventana pueden tener el foco del teclado. Cómo obtienen las ventanas el foco del teclado depende del sistema. Pero una vez que una ventana a obtenido el foco puede utilizar el método `requestFocus()` para pedir que un componente obtenga el foco.

Cuando un componente obtiene el foco se llama a su método `gotFocus()`. Cuando un componente pierde el foco se llama a su método `lostFocus()`.

Utilizar los Componentes del AWT

La siguiente figura muestra el árbol de herencia para todas las clases componentes del AWT.



Como muestra la figura, todos los componentes, excepto los componentes relacionados con los menús, descienden de la clase `Component` del AWT. A causa de las restricciones de las distintas plataformas (como la imposibilidad de seleccionar el color de fondo de los menús), todos los componentes relacionados con los menús se han sacado fuera de la clase `Component`. En su lugar, los componentes de menús descienden de la clase `MenuComponent` del AWT.

Reglas Generales para Utilizar Componentes

Antes de empezar a utilizar componentes, deberías conocer qué proporciona la clase `Component` y cómo puedes personalizar los componentes.

Cómo utilizar ...

El siguiente grupo de páginas explica cómo utilizar los componentes proporcionados por el AWT. Cada tipo de componente tiene su propia página:

- [Cómo utilizar Botones](#)
- [Cómo utilizar Lienzos](#)
- [Cómo utilizar Checkboxes](#)
- [Cómo utilizar Choices](#)
- [Cómo utilizar Cajas de Diálogo](#)
- [Cómo utilizar Marcos](#)
- [Cómo utilizar Etiquetas](#)
- [Cómo utilizar Listas](#)
- [Cómo utilizas Menús](#)
- [Cómo utilizas Panels](#)
- [Cómo utilizar Barras de Desplazamiento](#)
- [Cómo utilizar Áreas y Campos de Texto](#)

Reglas Generales para Utilizar Componentes

Esta página tiene información general sobre lo que tienen en común los componentes. Explica cómo añadir componentes a un contenedor. Cuenta las funcionalidades que heredan los componentes de la clase `Component`. También cuenta cómo puede cambiar el aspecto y comportamiento de los componentes.

Cómo añadir Componentes a un Contenedor

Cuando leas esta lección, observarás código para añadir componentes a los contenedores. Esto es así porque para que cualquier objeto `Component` pueda dibujarse en la pantalla, excepto una ventana, primero debes añadirlo a un objeto `Container`. Este Contenedor es a su vez un Componente, y también debe ser añadido a otro Contenedor. Las Ventanas, como los Marcos y los Cuadros de Diálogos son contenedores de alto nivel; y son los únicos componentes que no son añadidos a otros contenedores.

La clase [Container](#) define tres métodos para añadir componentes: un método `add()` con un argumento y dos métodos `add()` con dos argumentos. El que se necesitará utilizar depende el controlador de disposición que el contenedor está utilizando. Aprenderás todo sobre los controladores de disposición más adelante en esta lección. Ahora, estamos enseñando lo suficiente para que puedas leer los extractos de código de esta lección.

El método `add()` con un argumento sólo requiere que especifique el componente a añadir. El primer método `add()` con dos argumentos permite añadir un argumento especificando la posición en la que debería ser añadido el componente. El entero `-1` especifica que el contenedor añade el componente al final de la lista de componentes (al igual que lo hace el método de un sólo argumento). Esta forma de `add()` con dos argumentos no se utiliza frecuentemente, por eso si se refiere al "método `add()` con dos argumentos", casi siempre se refiere al segundo método que se describe en el siguiente párrafo. Todos los controladores de disposición [FlowLayout](#), [GridLayout](#), y [GridBagLayout](#) trabajan con estos métodos `add()`.

El segundo método `add()` con dos argumentos especifica el componente que se va a añadir. El primer argumento es una cadena dependiente del controlador. [BorderLayout](#) (el controlador de disposición por defecto para las subclases de `Window`) requiere que se especifique "North", "South", "East", "West", o "Center". [CardLayout](#) sólo requiere que se especifique una cadena que identifique de alguna forma el componente que está siendo añadido.

Nota: Añadir componentes a un contenedor elimina el componente de otro contenedor si es que estuviera. Por esta razón, no se puede tener un componente en dos contenedores, incluso si los dos contenedores no se muestran nunca al mismo tiempo.

Qué proporciona la clase **Component**

Todos los componentes, excepto los menús, están implementados como subclases de la clase [Component](#). De esta clase heredan una enorme cantidad de funcionalidades. Por ahora, ya deberías saber que la clase **Component** proporciona lo básico para el dibujo y el manejo de eventos. Aquí tienes una lista más completa sobre las funcionalidades proporcionadas por la clase **Component**:

Soporte básico para dibujo.

Component proporciona los métodos `paint()`, `update()`, y `repaint()`, que permiten a los componentes dibujarse a sí mismos en la pantalla. Puedes ver la página [Dibujo](#) para más información.

Manejo de Eventos.

Component define el método `handleEvent()` de propósito general y un grupo de métodos como `action()` que maneja tipos de eventos específicos. **Component** también tiene soporte para el foco del teclado, que permite controlar los componentes mediante el teclado. Puedes ver la página [Manejo de Eventos](#) para más información.

Apariencia del Control: fuente.

Component proporciona métodos para obtener y cambiar la fuente actual, y obtener información sobre ella. Puedes ver la página [Trabajando con Texto](#) para más información.

Apariencia del Control: color.

Component proporciona los siguientes métodos para obtener y cambiar los colores de fondo y de primer plano: `setForeground(Color)`, `getForeground()`, `setBackground(Color)`, y `getBackground()`. El color de primer plano es el utilizado por todo el texto del componente, así como cualquier dibujo del cliente que realice el componente. El color de fondo es el color que hay detrás del texto o los gráficos. Para aumentar el rendimiento el color de fondo debería contrastar con el color de primer plano.

Manejo de Imágenes.

Component proporciona lo básico para mostrar imágenes. Observa que la mayoría de los componentes no pueden contener imágenes, ya que su apariencia está implementada en código específico de la

plataforma. Sin embargo, los lienzos y la mayoría de los contenedores, pueden mostrar imágenes. Puedes ver la página [Utilizar imágenes](#) para más información sobre el trabajo con imágenes.

Tamaño y posición en la pantalla.

El tamaño y posición de todos los componetes (excepto para las ventanas) están sujetos a los caprichos de los controladores de disposición. Sin embargo, todos los componentes tienen al menos una forma de decir su tamaño, pero no su posición. Los métodos `preferredSize()` y `minimumSize()` permiten a los componentes informar al controlador de disposición de los tamaños mínimo y preferido. La clase `Component` también proporciona métodos para obtener o cambiar (sujeto a la supervisión del controlador de disposición) el tamaño y la posición actuales del componente.

Cómo cambiar la apariencia y el comportamiento de los Componentes.

La apariencia de la mayoría de los componentes es específica de la plataforma. Los botones son diferentes en un sistema Motif y en un sistema Macintosh, por ejemplo.

No se puede cambiar fácilmente la apariencia de un componente en grandes rasgos. Pero si puede hacer cambios menores, como la fuente o el color del fondo, utilizando los métodos y variables que afectan a la apariencia proporcionados por una clase `Component` y sus subclases. Sin embargo, no se podrá cambiar completamente la apariencia de un componente incluso creando una nueva subclase de la clase `Component`, ya que la mayoría de la implementación del componetes específica de la plataforma sobrescribe cualquier dibujo que realice el componente. Para cambiar la apariencia de un componente, se debe implementar una subclase de la clase `Canvas` que tenga el aspecto que se quiera pero el mismo comportamiento que el usuario esperaría del componente.

Aunque no puede cambiar en gran medida el aspecto del componente, si puede cambiar su comportamiento. Por ejemplo, si en un campo de texto sólo son válidos los valores numéricos, podría implementar una subclase de `TextField` que examine todos los eventos del teclado, interceptando los que no sean válidos. Esto es posible porque la implementación del componente independiente de la plataforma obtiene el evento antes de que lo haga la implementación dependiente de la plataforma. Puedes ver la página [Manejo de Eventos](#) para más detalles.

Cómo Utilizar la Clase Button

La clase [Button](#) proporciona una implementación por defecto para los botones. Un botón es un sencillo control que genera un evento Action cuando el usuario lo pulsa.

La apariencia en la pantalla de los botones depende de la plataforma en que se está ejecutando y si el botón está disponible o no. Si se quiere que los botones sean iguales para todas las plataformas o que tengan un aspecto especial, se debería crear una subclase de [Canvas](#) para implementar ese aspecto; no se puede cambiar el aspecto utilizando la clase Button. La única faceta de la apariencia de un botón que puede cambiar sin crear su propia clase son la fuente, el texto mostrado, los colores de fondo y de primer plano, y (habilitando o deshabilitando el botón) si el botón aparece activado o desactivado.

Abajo hay un applet que muestra tres botones. Cuando se pulsa el botón izquierdo, se desactivará el botón central (y así mismo, ya que ahora no es útil) y activa el botón derecho. Cuando se pulse el botón derecho, se activará el botón central y el botón izquierdo y se desactivará a sí mismo.

Abajo tienes el código que crea los tres botones y reacciona a las pulsaciones de los botones. (Aquí tienes el [programa completo](#).)

//en el código de inicialización:

```
b1 = new Button();
b1.setLabel("Disable middle button");

b2 = new Button("Middle button");

b3 = new Button("Enable middle button");
b3.disable();
. . .
```

```
public boolean action(Event e, Object arg) {
    Object target = e.target;

    if (target == b1) { //Han pulsado "Disable middle button"
        b2.disable();
        b1.disable();
        b3.enable();
        return true;
    }
    if (target == b3) { //Han pulsado "Enable middle button"
        b2.enable();
        b1.enable();
        b3.disable();
    }
}
```

```
        return true;
    }
    return false;
}
```

El código anterior muestra cómo utilizar uno de los métodos más comunes de la clase Button. Además, la clase Button define un método `getLabel()`, que le permite encontrar el texto mostrado en un botón particular.

Cómo utilizar la Clase Canvas

La clase [Canvas](#) existe para tener descendientes. No hace nada por sí misma; sólo proporciona una forma para implementar un componente personalizado. Por ejemplo, los Lienzos (Canvas) son útiles para áreas de dibujo para imágenes y gráficos del cliente, tanto si se desea o no manejar eventos que ocurran dentro del área de pantalla.

Los Canvas también son útiles cuando se quiera un control -- un botón, por ejemplo -- que se parezca a la implementación por defecto de ese control. Como no se puede cambiar la apariencia de los controles estandards creando una subclase de la clase Component correspondiente (Button, por ejemplo), en su lugar se puede crear una subclase de Canvas que tenga el comportamiento que se quiera y el mismo comportamiento que la implementación por defecto del control.

Cuando se implemente una subclase de Canvas, ten cuidado al implementar los métodos `minimumSize()` y `preferredSize()` para que reflejen adecuadamente el tamaño de su lienzo. De otro modo, dependiendo del controlador de disposición que utilice el contenedor de su Canvas, su podría terminar demasiado pequeño -- quizás invisible.

Aquí tienes un applet que utiliza dos ejemplares de una subclase de Canvas: ImageCanvas.

Abajo tienes el código de ImageCanvas. (Podrás encontrarlo en la forma electrónica en el fichero [ImageApplet.java](#).) Como los datos de la imagen se cargan asíncronamente, un ImageCanvas no conoce el tamaño que debería tener hasta algún momento después de haberse creado. Por esta razón, ImageCanvas utiliza la anchura y altura sugeridos por su creador hasta que esté disponible el dato de su tamaño. Cuando el tamaño de la imagen se vuelve disponible, el ImageCanvas cambia el tamaño que devuelven sus métodos `preferredSize()` y `minimumSize()`, intenta redimensionarse, y luego pide al contenedor de más alto nivel que se ajuste de forma adecuada al nuevo tamaño y que se redibuje.

```
class ImageCanvas extends Canvas {
    Container pappy;
    Image image;
    boolean trueSizeKnown = false;
    Dimension minSize;
    int w, h;

    public ImageCanvas(Image image, Container parent,
                       int initialWidth, int initialHeight) {
        if (image == null) {
            System.err.println("Canvas got invalid image object!");
            return;
        }
    }
}
```

```

    this.image = image;
    pappy = parent;

    w = initialWidth;
    h = initialHeight;

    minSize = new Dimension(w,h);
}

public Dimension preferredSize() {
    return minimumSize();
}

public synchronized Dimension minimumSize() {
    return minSize;
}

public void paint (Graphics g) {
    if (image != null) {
        if (!trueSizeKnown) {
            int imageWidth = image.getWidth(this);
            int imageHeight = image.getHeight(this);

            if ((imageWidth > 0) && (imageHeight > 0)) {
                trueSizeKnown = true;

                //Ooh... component-initiated resizing.
                w = imageWidth;
                h = imageHeight;
                minSize = new Dimension(w,h);
                resize(w, h);
                pappy.layout();
                pappy.repaint();
            }
        }

        g.drawRect(0, 0, w - 1, h - 1);
        g.drawImage(image, 0, 0, this);
    }
}
}

```

Para más información sobre el dibujo de gráficos, puedes ver la lección [Trabajar con Gráficos](#). Para ver un ejemplo de implementación de la clase Canvas que

dibuja gráficos del cliente y maneja eventos, puedes ver el código del applet [RectangleDemo](#). Puedes ver RectangleDemo en acción en [Dibujar Formas](#).

Ozito

Cómo Utilizar la Clase Checkbox

La clase [Checkbox](#) proporciona cajas de chequeo -- botones con dos estados que pueden estar "on" o "off". (Quizás podría conocer este elemento UI como botón de radio.) Cuando el usuario pulsa un botón de radio, cambia su estado y genera un evento Action. Otras formas de proporcionar grupos de botones de radio que pueden ser seleccionados son [choices](#), [lists](#), y [menus](#).

Si se quiere un grupo de botones de radio en el que sólo uno pueda estar activado, puede añadir un objeto [CheckboxGroup](#) para vigilar el estado de los botones de radio.

Abajo hay un applet que tiene dos columnas de botones de radio. En la izquierda hay tres botones independientes. Se pueden seleccionar los tres botones de radio, si se quiere. En la derecha hay tres botones de radio que están corrdinados por un objeto CheckboxGroup. Este objeto se asegura de que no haya más de un botón de radio seleccionado al mismo tiempo. Para ser específicos, un grupo de botones de radio puede no tener ninguno seleccionado, pero una vez que el usuario selecciona un botón, sólo uno de ellos podrá ser seleccionado en adelante.

Aquí tienes el [programa completo](#). Abajo tienes el código que crea los dos grupos de botones de radio. Observa que sólo elsegundo, el grupo mutuamente exclusivo de botones está controlado por un CheckboxGroup.

```
Panel p1, p2;
Checkbox cb1, cb2, cb3; //Estos son los botones de radio independientes.
Checkbox cb4, cb5, cb6; //Estos botones de radio son parte de un grupo.
CheckboxGroup cbg;

cb1 = new Checkbox(); //El estado por defecto es "off" (false).
cb1.setLabel("Checkbox 1");
cb2 = new Checkbox("Checkbox 2");
cb3 = new Checkbox("Checkbox 3");
cb3.setState(true); //Cambia el estado a"on" (true).
. . .
cbg = new CheckboxGroup();
cb4 = new Checkbox("Checkbox 4", cbg, false); //estado inicial: off (false)
cb5 = new Checkbox("Checkbox 5", cbg, false); //estado inicial: off
cb6 = new Checkbox("Checkbox 6", cbg, false); //estado inicial: off
```

Junto con los métodos de la clase Checkbox mostrados arriba, esta clase tiene dos métodos adicionales que podrías querer utilizar: `getCheckboxGroup()` y `setCheckboxGroup()`. Junto con el sencillo constructor `CeckboxGroup` utilizado en el código de ejemplo, `CheckboxGroup` también define los siguientes métodos: `getCurrent()` y `setCurrent()`. Estos métodos obtienen y cambian (respectivamente) el botón de radio seleccionado actualmente.

Cómo Utilizar la Clase Choice

La clase [Choice](#) proporciona una lista de opciones al estilo menú, a la que se accede por un botón distintivo. Cuando el usuario elige un ítem de la lista, la clase Choice genera un evento Action.

La clase Choice es útil cuando se necesita mostrar un número de alternativas a una cantidad de espacio limitada, y el usuario no necesita ver todas las alternativas al mismo tiempo. Otro nombre por el que se podría conocer este elemento UI es lista desplegable. Otras formas de proporcionar múltiples alternativas son [checkboxes](#), [lists](#), y [menus](#).

Abajo tienes un applet que tiene una lista desplegable y una etiqueta. Cuando el usuario elegie un ítem de la lista, la etiqueta cambia para reflejar el ítem elegido. Observa que el índice del primer ítem de una lista desplegable es 0.

Abajo tienes el código que crea la lista desplegable y maneja los eventos. (Aquí tienes el [programa completo](#).) Observa que el segundo parámetro del método action() (que es el mismo que e.arg), es la cadena del ítem seleccionado.

```
//...Donde se definen las variables de ejemplar:
Choice choice; //pop-up list of choices

//...Donde ocurre la inicialización:
choice = new Choice();
choice.addItem("ichi");
choice.addItem("ni");
choice.addItem("san");
choice.addItem("yon");
label = new Label();
setLabelText(choice.getSelectedIndex(), choice.getSelectedItem());

. . .

public boolean action(Event e, Object arg) {
    if (e.target instanceof Choice) {
        setLabelText(choice.getSelectedIndex(), (String)arg);
        return true;
    }
    return false;
}
```

Junto con el método utilizado arriba, la clase Choice define otros métodos útiles:
int countItems()

Devuelve el número de ítems de la lista.

String getItem(int)

Devuelve la cadena mostrada por el ítem en el índice especificado.

void select(int)

Selecciona el ítem del índice especificado.

void select(String)

Selecciona el ítem que está mostrando la cadena especificada.

Ozito

Cómo Utilizar la Clase Dialog

El AWT proporciona soporte para cuadros de diálogo -- ventanas que son dependientes de otras ventanas -- con la clase [Dialog](#). Ésta proporciona una subclase muy útil [FileDialog](#) que proporciona cuadros de diálogos para ayudar al usuario a abrir y guardar ficheros.

Lo único que distingue a los cuadros de diálogo de las ventanas normales (que son implementadas con objetos Frame) es que el cuadro de diálogo depende de alguna otra ventana (un Frame). Cuando esta otra ventana es destruida, también lo son sus cuadros de diálogo dependientes. Cuando esta otra ventana es miniaturizada sus cuadros de diálogo desaparecen de la pantalla. Cuando esta otra ventana vuelve a su estado normal, sus cuadros de diálogo vuelven a aparecer en la pantalla. El AWT proporciona automáticamente este comportamiento.

Como no existe un API actualmente que permita a los Applets encontrar la ventana en la que se están ejecutando, estos generalmente no pueden utilizar cuadros de diálogo. La excepción son los applets que traen sus propias ventanas (Frames) que pueden tener cuadros de diálogo dependientes de esas ventanas. Por esta razón, el siguiente applet consiste en un botón que trae una ventana que muestra un cuadro de diálogo.

Los cuadros de diálogo pueden ser modales. Los cuadros de diálogo modales requieren la atención del usuario, para evitar que el usuario haga nada en la aplicación del cuadro de diálogo hasta que se haya finalizado con él. Por defecto, los cuadros de diálogo no son modales -- el usuario puede mantenerlos y seguir trabajando con otras ventanas de la aplicación.

Aquí tienes el [código](#) para la ventana que muestra el applet anterior. Este código puede ser ejecutado como una aplicación solitaria o, con la ayuda de la clase [AppletButton](#), como un applet. Aquí sólo está el código que implementa el objeto Dialog:

```
class SimpleDialog extends Dialog {
    TextField field;
    DialogWindow parent;
    Button setButton;

    SimpleDialog(Frame dw, String title) {
        super(dw, title, false);
        parent = (DialogWindow)dw;

        ...//Crea y añade componentes, como un conjunto de botones.

        //Initialize this dialog to its preferred size.
        pack();
    }
}
```

```

public boolean action(Event event, Object arg) {
    if ( (event.target == setButton)
        | (event.target instanceof TextField)) {
        parent.setText(field.getText());
    }
    field.selectAll();
    hide();
    return true;
}
}

```

El método `pack()` en el constructor de `SimpleDialog` es un método definido por la clase `Window`. (Recuerde que `dialog` es una subclase de `Window`). El método `pack()` redimensiona la ventana para que todos sus contenidos tengan su tamaño preferido o mínimo (dependiendo del controlador de disposición de la ventana). En general, la utilización de `pack()` es preferible a llamar al método `resize()` de una ventana, ya que `pack()` deja que cargue el controlador de disposición con la decisión del tamaño de la ventana, y éste es muy bueno ajustando las dependencias de la plataforma y otros factores que afectan al tamaño de los componentes.

Aquí tienes el código que muestra el cuadro de diálogo:

```

if (dialog == null) {
    dialog = new SimpleDialog(this, "A Simple Dialog");
}
dialog.show();

```

Junto con los métodos utilizados en el primer fragmento de código, la clase `Dialog` proporciona los siguientes métodos:

`Dialog(Frame, boolean)`

Igual que el constructor utilizado arriba, pero no selecciona el título de la ventana del cuadro de diálogo.

`boolean isModal()`

Devuelve `true` si el cuadro de diálogo es modal.

`String getTitle(), String setTitle(String)`

Obienen o cambian (respectivamente) el título del cuadro de diálogo.

`boolean isResizable(), void setResizable(boolean)`

Encuentra o selecciona (respectivamente) si el tamaño de la ventana del cuadro de diálogo puede cambiarse.

Cómo Utilizar la Clase Frame

La clase [Frame](#) proporciona ventanas para los applets y las aplicaciones. Cada aplicación necesita al menos un Frame (Marco). Si una aplicación tiene una ventana que debería depender de otra ventana -- desapareciendo cuando la otra ventana se minimiza, por ejemplo -- debería utilizar un [Cuadro de Diálogo](#) en vez de un Frame para la ventana dependiente.

Desafortunadamente, los applets no puede utilizar cajas de diálogo por ahora, por eso utilizan Frames en su lugar.

Las páginas [Cómo utilizar la Clase Menu](#) y [Cómo Utilizar la Clase Dialog](#) son dos de las muchas de este tutorial que utilizan un Frame.

Abajo tienes el código que utiliza la demostración de Menús para crear su ventana (una subclase de Frame) y manejarla en caso de que el usuario cierre la ventana.

```
public class MenuWindow extends Frame {
    boolean inAnApplet = true;
    TextArea output;

    public MenuWindow() {
        ...//Este constructor llama implícitamente al constructor sin argumentos
        //de la clase Frame y añade los componentes de la ventana
    }

    public boolean handleEvent(Event event) {
        if (event.id == Event.WINDOW_DESTROY) {
            if (inAnApplet) {
                dispose();
            } else {
                System.exit(0);
            }
        }
        return super.handleEvent(event);
    }
}

. . .

public static void main(String args[]) {
    MenuWindow window = new MenuWindow();
    window.inAnApplet = false;

    window.setTitle("MenuWindow Application");
    window.pack();
    window.show();
}
```

El método `pack()`, que es llamado desde el método `main()`, está definido por la clase `Window`. Puedes ver [Cómo Utilizar la Clase Dialog](#) para más información sobre `pack()`.

Junto con el constructor sin argumentos de `Frame` utilizado implícitamente por el constructor de `MenuWindow`, la clase `Frame` también proporciona un constructor con un argumento. El argumento es un `String` que especifica el título de la ventana del `Frame`.

Otros métodos interesantes proporcionados por la clase `Frame` son:

String getTitle() y void setTitle(String)

Devuelven o cambian (respectivamente) el título de la ventana.

Image getIconImage() y void setIconImage(Image)

Devuelven o cambian (respectivamente) la imagen mostrada cuando se minimiza la ventana.

MenuBar getMenuBar() y void setMenuBar(MenuBar)

Devuelven o seleccionan (respectivamente) la barra de menús de ese Frame.

void remove(MenuComponent)

Elimina la barra de menú especificada del Frame.

boolean isResizable() y void setResizable(boolean)

Devuelven o seleccionan si el usuario puede cambiar o no el tamaño de la ventana.

int getCursorType() y void setCursor(int)

Obtiene la imagen actual del cursor o selecciona la imagen del cursor. El cursor debe ser especificado como uno de los tipos definidos en la clase Frame. Los tipos predefinidos son :

- **Frame.DEFAULT_CURSOR,**
- **Frame.CROSSHAIR_CURSOR,**
- **Frame.HAND_CURSOR,**
- **Frame.MOVE_CURSOR,**
- **Frame.TEXT_CURSOR,**
- **Frame.WAIT_CURSOR,**
- **Frame.X_RESIZE_CURSOR,** donde X es SW, SE, NW, NE, N, S, W, o E.

Cómo Utilizar la Clase Label

La clase [Label](#) proporciona una forma sencilla de colocar un texto no seleccionable en el GUI del programa. Las Etiquetas (Labels) están alineadas a la izquierda de su área de dibujo, por defecto. Se puede especificar que se centren o se alineen a la derecha especificando `Label.CENTER` o `Label.RIGHT` en su constructor o en el método `setAlignment()`. Como con todos los Componentes, también se puede especificar el color y la fuente de la etiqueta. Para más información sobre el trabajo con fuentes, puede ver [Obtener Información sobre la Fuente: FontMetrics](#).

Las etiquetas se utilizan en todos los ejemplos de este tutorial. Por ejemplo el applet de [Cómo Utilizar la Clase Choice](#), utiliza una etiqueta que muestra información sobre el ítem que está actualmente seleccionado.

Aquí tienes un applet que muestra la alineación de las etiquetas:

El applet crea tres etiquetas, cada una con una alineación diferente. Si el área de display de cada etiqueta fuera igual a la anchura del texto de la etiqueta, no se vería ninguna diferencia en la alineación de las etiquetas. El texto de cada etiqueta sólo se mostrará en el espacio disponible. Sin embargo, este applet hace que cada etiqueta sea tan ancha como el applet, que es mayor que cualquiera de las etiquetas. Como resultado, puede ver una posición horizontal diferente en el dibujo del texto de las tres etiquetas. Aquí tienes el [programa completo](#).

Abajo tienes el código que utiliza el applet para crear las etiquetas y seleccionar su alineación. Para propósitos de enseñanza este applet utiliza los tres constructores de la clase `Label`.

```
Label label1 = new Label();
label1.setText("Right");
Label label2 = new Label("Center");
label2.setAlignment(Label.CENTER);
Label label3 = new Label("RIGHT", Label.RIGHT);
```


Cómo Utilizar la Clase List

La clase [List](#) proporciona un área desplegable que contiene ítems seleccionables (uno por línea). Generalmente, un usuario selecciona una opción pulsando sobre ella, e indica que una acción debe ocurrir cuando hace doble-click sobre ella o pulsa la tecla Return. Las Listas (Lists) pueden permitir múltiples selecciones o sólo una selección a la vez. Otros componentes que permiten al usuario elegir entre varias opciones son [checkbox](#), [choice](#), y [menu](#).

Abajo tienes un applet que muestra dos listas, junto con un área de texto que muestra información sobre los eventos. La lista superior (que son números en español) permite múltiples selecciones. La inferior (que son numeros italianos) sólo permite una selección. Observa que el primer ítem de cada lista tiene índice 0.

Abajo tienes el código que crea las listas y maneja sus eventos. (Aquí tienes el [programa completo](#).) Observa que el dato e.arg para los enventos Action (que es pasado dentro del método action() como su segundo argumento) es el nombre del ítem activado, similar a los argumentos para los eventos Action de otros componentes como los botones e incluso los menús. Sin embargo, el dato e.arg para los eventos que no son Action de la lista es el índice del ítem seleccionado.

```
...//Donde se declaren las variables de ejemplar:
```

```
TextArea output;
```

```
List spanish, italian;
```

```
...//Donde ocurra la inicialización:
```

```
//Primero construye la lista que permite selecciones múltiples.
```

```
spanish = new List(4, true); //el número 4 es visible al inicializar
```

```
spanish.addItem("uno");
```

```
spanish.addItem("dos");
```

```
spanish.addItem("tres");
```

```
spanish.addItem("cuatro");
```

```
spanish.addItem("cinco");
```

```
spanish.addItem("seis");
```

```
spanish.addItem("siete");
```

```
//Construye la segunda lista, que permite sólo una selección a la vez.
```

```
italian = new List(); //Por defecto ninguno es visible, sólo uno seleccionable
```

```
italian.addItem("uno");
```

```
italian.addItem("due");
```

```
italian.addItem("tre");
```

```
italian.addItem("quattro");
```

```
italian.addItem("cinque");
```

```
italian.addItem("sei");
```

```
italian.addItem("sette");
```

. . .

```
public boolean action(Event e, Object arg) {
    if (e.target instanceof List) {
        String language = (e.target == spanish) ?
            "Spanish" : "Italian";
        output.appendText("Action event occurred on \"
            + (String)arg + "\" in \"
            + language + ".\n");
    }
    return true;
}

public boolean handleEvent(Event e) {
    if (e.target instanceof List) {
        List list = (List)(e.target);
        String language = (list == spanish) ?
            "Spanish" : "Italian";
        switch (e.id) {
            case Event.LIST_SELECT:
                int sIndex = ((Integer)e.arg).intValue();
                output.appendText("Select event occurred on item #\"
                    + sIndex + \" (\")\"
                    + list.getItem(sIndex) + \"\") in \"
                    + language + ".\n");
                break;
            case Event.LIST_DESELECT:
                int dIndex = ((Integer)e.arg).intValue();
                output.appendText("Deselect event occurred on item #\"
                    + dIndex + \" (\")\"
                    + list.getItem(dIndex) + \"\") in \"
                    + language + ".\n");
        }
    }
    return super.handleEvent(e);
}
```

Junto con los dos constructores y los métodos addItem() y getItem() mostrados arriba, la clase List proporciona los siguientes métodos:

int countItems()

Devuelve el número de opciones de la Lista.

String getItem(int)

Devuelve la cadena mostrada por la opción del índice especificado.

void addItem(String, int)

Añade la opción especificada en el índice especificado.

void replaceItem(String, int)

Reemplaza la opción el índice especificado.

`void clear()`, `void delItem(int)`, y `void delItems(int, int)`

Borran una o más opciones de la lista. El método `clear()` vacía la lista. El método `delItem()` borra la opción especificada de la lista. El método `delItems()` borra las opciones que existan entre los índices especificados (ambos incluidos).

`int getSelectedIndex()`

Devuelve el índice de la opción selecciona en la lista. Devuelve -1 si no se ha seleccionado ninguna opción o si se ha seleccionado más de una.

`int[] getSelectedIndexes()`

Devuelve los índices de las opciones seleccionadas.

`String getSelectedItem()`

Igual `getSelectedIndex()`, pero devuelve el string con el texto de la opción en lugar de su índice. Devuelve null si no se ha seleccionado ninguna opción o si se ha seleccionado más de una.

`String[] getSelectedItems()`

Igual `getSelectedIndexes()`, pero devuelve los strings con el texto de las opciones seleccionadas en lugar de sus índices.

`void select(int)`, `void deselect(int)`

Seleccionan o deseleccionan la opción con el índice especificado.

`boolean isSelected(int)`

Devuelve true si la opción con el índice especificado está seleccionada.

`int getRows()`

Devuelve el número de líneas visibles en la lista.

`boolean allowsMultipleSelections()`, `boolean setMultipleSelections()`

Devuelve o cambia si la lista permite selecciones múltiples o no.

`int getVisibleIndex()`, `void makeVisible(int)`,

El método `makeVisible()` fuerza a que opción seleccionado sea visible. El método `getVisibleIndex()` obtiene el indice de la opción que se hizo visible la última vez con el método `makeVisible()`.

Cómo Utilizar la Clase Menu

El siguiente applet muestra muchas de las características de los menús que quieras utilizar. La ventana tiene una barra de menú que contiene cinco menús. Cada menú contiene una o más opciones. El Menú 1 es un menú de arranque, pulsando la línea punteada, el usuario crea una nueva ventana que contiene las mismas opciones de menú que el menu 1. (Actualmente, los menús de arranque solo están disponibles para la plataforma Solaris, no para Windows 95/NT o MacOS.) El menú 2 sólo contiene un botón de radio. El menú 3 contiene un separador entre su segunda y tercera opciones. El menú 4 contiene un submenú. El menú 5 es la ventana del menú de ayuda, que dependiendo de la plataforma) generalmente significa que se sitúa a la izquierda. Cuando el usuario pulsa en cualquier opción del menú la ventana muestra una cadena de texto indicando qué opción ha sido pulsada y en que menú se encuentra.

La razón por la que el applet trae una ventana para demostrar los menús es que el AWT limita donde se pueden utilizar los menús. Los menús sólo pueden existir en las barras de menú, y las barras de menú sólo están disponibles en las ventanas (específicamente, en los Frames).

Si los menús no están disponibles o no son apropiados en tu programa deberías buscar otras formas de presentar opciones al usuario: [checkbox](#), [choice](#), y [list](#).

La funcionalidad de los menús del AWT la proporcionan varias clases. Estas clases no descienden de la clase Component, ya que muchas plataformas ponen muchos límites a las capacidades de los menús. En su lugar, las clases menu descienden de la clase [MenuComponent](#). El AWT proporciona las siguientes subclases de MenuComponent para soportar los menús:

[MenuItem](#)

Cada opción de un menú está representada por un objeto MenuItem.

[CheckboxMenuItem](#)

Cada opción de un menú que contiene un botón de radio está representada por un objeto CheckboxMenuItem. CheckboxMenuItem es una subclase de MenuItem.

[Menu](#)

Cada menú está representado por un objeto Menu. Esta clase está implementada como una subclase de MenuItem para se puedan crear submenús fácilmente añadiendo un menú a otro.

[MenuBar](#)

Las barras de menú están implementadas por la clase MenuBar. Esta clase representa una noción dependiente de la plataforma de un grupo de manús adheridos a una ventana. Las barras de menú no no pueden utilizarse con la clase Panel.

Para poder contener un MenuComponent, un objeto debe adherirse al interface [MenuContainer](#). Las clases Frame, Menu y MenuBar son las únicas del aWT que actualmente implementan MenuContainer.

Aquí tienes el [código](#) para la ventana del applet anterior. Esto código puede ejecutarse como una aplicación solitaria o, con la ayuda de la clase [AppletButton](#), como un applet. Aquí tiene el código que trata con los menús:

```
public class MenuWindow extends Frame {  
    . . .  
    public MenuWindow() {  
        MenuBar mb;  
        Menu m1, m2, m3, m4, m4_1, m5;  
        MenuItem mi1_1, mi1_2, mi3_1, mi3_2, mi3_3, mi3_4,
```

```

        mi4_1_1, mi5_1, mi5_2;
CheckboxMenuItem mi2_1;

...//Añade la salida mostrada a esta ventana...

//Construye la barra de menú.
mb = new MenuBar();
setMenuBar(mb);

//Construye el primer menú en la barra de menús.
//Especificando el segundo argumento como true
//hace de este un menú de arranque.
m1 = new Menu("Menu 1", true);
mb.add(m1);
mi1_1 = new MenuItem("Menu Item 1_1");
m1.add(mi1_1);
mi1_2 = new MenuItem("Menu Item 1_2");
m1.add(mi1_2);

//Construye el menú de ayuda.
m5 = new Menu("Menu 5");
mb.add(m5);           //Sólo seleccionandolo no funciona, debe añadirlo.
mb.setHelpMenu(m5);
mi5_1 = new MenuItem("Menu Item 5_1");
m5.add(mi5_1);
mi5_2 = new MenuItem("Menu Item 5_2");
m5.add(mi5_2);

//Construye el segundo menú en la barra de menús.
m2 = new Menu("Menu 2");
mb.add(m2);
mi2_1 = new CheckboxMenuItem("Menu Item 2_1");
m2.add(mi2_1);

//Construye el tercer menú en la barra de menús.
m3 = new Menu("Menu 3");
mb.add(m3);
mi3_1 = new MenuItem("Menu Item 3_1");
m3.add(mi3_1);
mi3_2 = new MenuItem("Menu Item 3_2");
m3.add(mi3_2);
m3.addSeparator();
mi3_3 = new MenuItem("Menu Item 3_3");
m3.add(mi3_3);
mi3_4 = new MenuItem("Menu Item 3_4");
mi3_4.disable();
m3.add(mi3_4);

//Construye el cuarto menú en la barra de menús.
m4 = new Menu("Menu 4");
mb.add(m4);
m4_1 = new Menu("Submenu 4_1");
m4.add(m4_1);

```

```

mi4_1_1 = new MenuItem("Menu Item 4_1_1");
m4_1.add(mi4_1_1);
}
. . .
public boolean action(Event event, Object arg) {
    String str = "Action detected";

    if (event.target instanceof MenuItem) {
        MenuItem mi=(MenuItem)(event.target);
        str += " on " + arg;
        if (mi instanceof CheckboxMenuItem) {
            str += " (state is "
                + ((CheckboxMenuItem)mi).getState()
                + ")";
        }
        MenuContainer parent = mi.getParent();
        if (parent instanceof Menu) {
            str += " in " + ((Menu)parent).getLabel();
        } else {
            str += " in a container that isn't a Menu";
        }
    }
    str += ".\n";
    ...//Muestra la Cadena de texto en al área de salida...
    return false;
}

```

Cómo Utilizar la Clase Panel

La clase [Panel](#) es una subclase de Container para propósitos generales. Se puede utilizar tal y como es para contener componentes, o se puede definir una subclase para realizar alguna función específica, como el manejo de eventos para los objetos contenidos en el Panel.

La clase [Applet](#) es una subclase de Panel con broches especiales para ejecutarse en un navegador o un visualizador de applets. Siempre que vea un programa que se puede ejecutar tanto como un applet como una aplicación, el truco está en que define una subclase de applet, pero no utiliza ninguna de las capacidades especiales del Applet, utilizando sólo los métodos heredados de la clase Panel.

Aquí tienes un ejemplo de utilización de un ejemplar de Pabel para contener algunos Componentes:

```
Panel p1 = new Panel();
p1.add(new Button("Button 1"));
p1.add(new Button("Button 2"));
p1.add(new Button("Button 3"));
```

Aquí tiene un ejemplo de una subclase de Panel que dibuja un marco alrededor de sus contenidos. Varias versiones de esta clase se utilizan en los ejemplos 1 y 2 de la página [Dibujar figuras](#).

```
class FramedArea extends Panel {
    public FramedArea(CoordinatesDemo controller) {
        ...//Selecciona el controlador de disposición.
        //Añade componentes a este panel...
    }

    //Asegura que no se han situado componentes en la parte superior del Frame.
    //Los valores de inset fueron determinados por ensayo y error.
    public Insets insets() {
        return new Insets(4,4,5,5);
    }

    //Dibuja el marco a los lados del Panel.
    public void paint(Graphics g) {
        Dimension d = size();
        Color bg = getBackground();

        g.setColor(bg);
        g.draw3DRect(0, 0, d.width - 1, d.height - 1, true);
        g.draw3DRect(3, 3, d.width - 7, d.height - 7, false);
    }
}
```

Muchos de los ejemplos de esta lección utilizan una subclase de Applet para manejar los eventos de los componentes que contienen. Por ejemplo, puedes verlos en la página [Cómo utilizar la Clase Dialog](#). Puede utilizar el manejo de eventos de estos y otros ejemplos como modelo para el manejo de los eventos de sus propias subclases de Applet o Panel.

Cómo Utilizar la Clase Scrollbar

Las barras de desplazamiento tienen dos usos:

- Una barra de desplazamiento puede actuar como un deslizador que el usuario manipula para seleccionar un valor. Un ejemplo de esto está en el programa Converter de la página [La Anatomía de un Programa basado en GUI](#).
- Las barras de desplazamiento le pueden ayudar a mostrar una parte de una región que es demasiado grande para el área de dibujo. Las barras de desplazamiento le permiten al usuario elegir exactamente la parte de la región que es visible. Aquí tiene un ejemplo (podría tardar un poco en cargarse la imagen):

Para crear una barra de desplazamiento, necesitas crear un ejemplar de la clase [Scrollbar](#). También se pueden inicializar los siguientes valores, o bien especificándolos al [Constructor de Scrollbar](#) o llamando al método `setValues()` antes de que la barra de desplazamiento sea visible.

`int orientation`

Indica si la barra de desplazamiento debe ser vertical u horizontal. Especificado con `Scrollbar.HORIZONTAL` o `Scrollbar.VERTICAL`.

`int value`

El valor inicial de la barra de desplazamiento. Para barras de desplazamiento que controlan un área desplazable, esto significa el valor x (para las barras horizontales, o el valor y (para las verticales) de la parte del área que es visible cuando el usuario la visita por primera vez. Por ejemplo, el applet anterior arranca en las posiciones 0 tanto vertical como horizontal y la porción de imagen representada empieza en (0,0).

`int visible`

El tamaño en pixels de la porción visible del área desplazable. Este valor, si se selecciona antes de que la barra de desplazamiento sea visible, determina cuantos pixels se desplazará la imagen con una pulsación en la barra de desplazamiento (pero no en el botón). Seleccionar este valor después de que la barra de desplazamiento sea visible no tiene ningún efecto. Después de que la barra de desplazamiento sea visible, se debería utilizar el método `setPageIncrement()` para obtener el mismo efecto.

`int minimum`

El valor mínimo que puede tener la barra de desplazamiento. Para barras de desplazamiento que controlan áreas desplazables este valor normalmente es 0 (la parte superior izquierda del área).

`int maximum`

El valor máximo que puede tener la barra de desplazamiento. Para barras de desplazamiento que controlan áreas desplazables este valor normalmente es: (la anchura o altura total , en pixels, del componente que está siendo

parcialmente representada) - (la anchura o altura visible actualmente del área desplazable).

Aquí tienes el [código](#) del applet anterior. Este código define dos clases. La primera es una sencilla subclase de Canvas (ScrollableCanvas) que dibuja una imagen. La segunda es una subclase de Panel (ImageScroller, que realmente desciende de Applet) que crea y contiene un ScrollableCanvas y dos Scrollbars. Este programa ilustra unos pocos detalles importantes sobre el manejo de un área desplazable:

- El manejo de eventos para una barra de desplazamiento es bastante sencillo. El programa solo debe responder a los eventos de desplazamiento guardando los nuevos valores de la barra de desplazamiento en un lugar accesible para que el componente pueda mostrarse dentro del área desplazable, y luego llamar al método `repaint()` del Componente.

```
public boolean handleEvent(Event evt) {
    switch (evt.id) {
        case Event.SCROLL_LINE_UP:
        case Event.SCROLL_LINE_DOWN:
        case Event.SCROLL_PAGE_UP:
        case Event.SCROLL_PAGE_DOWN:
        case Event.SCROLL_ABSOLUTE:
            if (evt.target == vert) {
                canvas.ty = ((Integer)evt.arg).intValue();
                canvas.repaint();
            }
            if (evt.target == horz) {
                canvas.tx = ((Integer)evt.arg).intValue();
                canvas.repaint();
            }
    }
    return super.handleEvent(evt);
}
```

- El Componente que se muestra a sí mismo dentro de un área desplazable puede ser muy sencillo. Todo lo que necesita es dibujarse a sí mismo en el origen especificado por los valores de sus barras de desplazamiento. Un Componente puede cambiar su origen (y así no tener que cambiar su código de dibujo normal), poniendo el siguiente código al principio de sus métodos `paint()` o `update()`:

```
g.translate(-tx, -ty);
```

- Cuando ocurre el desplazamiento, probablemente notarás parpadeo en el área de display. Si no se quiere que esto ocurra, se necesita implementar el método `update()` en el componente representado, y posiblemente también el doble buffer. Cómo hacer esto se explica en la página [Eliminar el Parpadeo](#).
- Si el área desplazable puede redimensionarse, cuidado con un problema común del desplazamiento. Ocurre cuando el usuario desplaza el área hacia abajo y la derecha y luego agranda el área. Si no se tiene cuidado, el área mostrará un

espacio en blanco en su parte inferior derecha. Después de que el usuario desplace y vuelve a la parte inferior derecha, el espacio en blanco no está más allí. Para evitar mostrar un espacio en blanco innecesario, cuando el área desplazable se agranda debe desplazar también el origen del Componente para aprovechar el nuevo espacio disponible. Aquí tienes un ejemplo:

```
int canvasWidth = canvas.size().width;

//Desplaza todo a la derecha si se está mostrando un espacio vacío
//en el lado derecho.
if ((canvas.tx + canvasWidth) > imageSize.width) {
    int newtx = imageSize.width - canvasWidth;
    if (newtx < 0) {
        newtx = 0;

    }
    canvas.tx = newtx;
}
```

Cómo Utilizar las Clases TextArea y TextField

Las clases [TextArea](#) y [TextField](#) muestran texto seleccionable y, opcionalmente, permite al usuario editar ese texto. Se pueden crear subclases de TextArea y TextField para realizar varias tareas cómo comprobar los errores de la entrada. Como con cualquier componente, puede especificar los colores de fondo y de primer plano y la fuente utilizada en los campos y área de texto. Sin embargo no se puede cambiar su apariencia básica.

Estas dos clases, TextArea y TextField son subclases de [TextComponent](#). De esta clase heredan métodos que les permiten cambiar y obtener la selección actual, permitir o desactivar la edición, obtener el texto seleccionado actualmente (o todo el texto), y modificar el texto.

Abajo tiene un applet que primero muestra un TextField y luego un TextArea. El TextField es editable; y el TextArea no. Cuando el usuario pulsa la tecla Return en el Campo de Texto, su contenido se copia dentro del Área de Texto y luego lo selecciona en el Campo de Texto.

Aquí tienes el [programa](#). Aquí tienes el código que crea, inicializa, y maneja eventos del Campo y el Área de texto:

```
//Donde se definan las variables de Ejemplar:
TextField textField;
TextArea textArea;

public void init() {
    textField = new TextField(20);
    textArea = new TextArea(5, 20);
    textArea.setEditable(false);

    ...//Añade los dos componentes al Panel...
}

public boolean action(Event evt, Object arg) {
    String text = textField.getText();
    textArea.appendText(text + "\n");
    textField.selectAll();
    return true;
}
```

La superclase TextComponent de TextArea y TextField suministra los métodos `getText()`, `setText()`, `setEditable()`, y `selectAll()` utilizados en el ejemplo anterior. También suministra los siguientes métodos: `getSelectedText()`, `isEditable()`, `getSelectionStart()`, y `getSelectionEnd()`. También proporciona un método `select()` que permite seleccionar el texto entre las posiciones de inicio y final que se especifiquen.

La clase `TextField` tiene cuatro constructores: `TextField()`, `TextField(int)`, `TextField(String)`, y `TextField(String, int)`. El argumento entero especifica el número de columnas del campo de texto. El argumento `String` especifica el texto mostrado inicialmente en el campo de texto. La clase `TextField` también suministra los siguientes métodos:

`int getColumns()`

Devuelve el número de columnas del campo de texto.

`setEchoChar()`

Activa el eco del caracter, es útil para los campos de Password.

`char getEchoChar()` y `boolean echoCharIsSet()`

Estos métodos le permite preguntar sobre el eco de caracteres.

Como la clase `TextField`, la clase `TextArea` tiene cuatro constructores: `TextArea()`, `TextArea(int, int)`, `TextArea(String)`, y `TextArea(String, int, int)`. Los argumentos enteros especifican el número de filas y columnas (respectivamente) del área de texto. El argumento `String` especifica el texto mostrada inicialmente en el área de texto.

La clase `TextArea` suministra el método `appendText()` utilizado en el ejemplo anterior. También suministra estos métodos:

`int getRows()`, `int getColumns()`

Devuelven el número de filas y columnas del área de texto.

`void insertText(String, int)`

Inserta el texto especificado en la posición indicada.

`void replaceText(String, int, int)`

Reemplaza el texto desde la posición inicial indicada (el primer entero) hasta la posición final indicada.

Detalles de la Arquitectura de Componentes

El AWT fue diseñado desde el principio para tener un API independiente de la plataforma y aún así preservar el aspecto y el comportamiento de cada plataforma. Por ejemplo, el AWT sólo tiene un API para los botones (proporcionado por la clase `Button`), pero un botón es diferente en un macintosh o en un PC bajo Windows 95.

El AWT consigue este objetivo ligeramente contradictorio proporcionando la clase (`Component`) que proporciona un API independiente de la plataforma para asegurarse de que utiliza las implementaciones específicas de las plataformas, los pares. Para ser específicos, cada clase componente del AWT (`Component`, `MenuComponent`, y sus subclases) tiene una clase par equivalente, y cada objeto componente tiene un objeto par que controla el aspecto y comportamiento del objeto.

Los pares de los botones son implementados en clases específicas de las plataformas que implementa el interface `ButtonPeer` de `java.awt.peer`. La clase `java.awt.Toolkit` define métodos que eligen exactamente la clases a utilizar por la implementación del par.

Cómo se crean los Pares

Los Pares son creados justo antes de que su objeto componente correspondiente sea dibujado por primera vez. Podrías haber observado un efecto laterar de esto: el tamaño de un componente no es válido hasta después de que el componente no se haya mostrado por primera vez.

Cuando se añade un componente a un contenedor no visible (un contenedor sin par), justo antes de que el contenedor se muestre por primera vez, su par, -- y los pares de todos los componentes que contiene -- son creados.

Sin embargo, si se añade un componente a un contenedor visible, necesitas decirle explícitamente al AWT que cree un par para el componente. Puedes hacer esto llamando al método `validate()`. Aunque se puede llamar a este método directamente sobre el componente que se está añadiendo, normalmente se invoca sobre el contenedor. La razón para esto es que llamar a `validate()` de un contenedor causa una reacción en cadena -- todos los componentes del contenedor también obtienen su validación. Por ejemplo, después de añadir componentes a un objeto `Applet`, se puede llamar el método `validate()` del `Applet`, el cual creará pares para todos los componentes del `Applet`.

Cómo manejan los Eventos los Pares

Los Pares implementan el comportamienteo (e, indirectamente, el aspecto) de los componentes del UI para reaccionar ante los eventos del usuario. Por ejemplo, cuando el usuario pulsa un botón, el par reacciona a la pulsación y la liberación del botón del ratón haciendo que el botón parezca que ha cambiado su aspecto y enviando el evento Action al objeto Button apropiado.

En teoría, los pares están al final de la cadena de eventos. Cuando ocurre un evento (como una pulsación de tecla), el Componente para el que se ha generado el evento obtiene el manejo para el evento, y luego (si el controlador de eventos del componente devuelve false) el Contenedor del Componente ve el evento, y así sucesivamente. Después de que todos los componentes del árbol hayan tenido la oportunidad de manejar el evento (y todos sus controladores de evento hayan devuelto false), llega al par y éste puede ver y reaccionar al evento.

En la implementación actual, el escenario anterior es verdad para las pulsaciones de teclas pero no para los eventos del ratón. Para los eventos de ratón, el par es el primero en ver el evento, y no necesita pasar todos los eventos al componente. Planeamos hacer que los eventos de ratón trabajen de la misma forma que los del teclado en una futura versión.

Para procesos como pulsaciones de teclas o del ratón, los pares algunas veces general eventos de nivel superior -- action, focus, change, minimización de ventanas, etc. Estos eventos de alto nivel se pasan al componente relevante para su manejo.

Problemas más comunes con los Componentes (y sus Soluciones)

Problema: ¿Cómo aumentar o disminuir el número de componentes de un contenedor?

- Para añadir un componente a un contenedor, se utiliza una de las tres formas del método `add()` de la clase `Container`. Puedes ver [Reglas Generales para el Uso de Componentes](#) para más información. Para eliminar un componente de un contenedor, se utilizan los métodos `remove()` o `removeAll()` de la clase `Container`. O simplemente se añade el componente a otro contenedor, y esto elimina automáticamente el componente de su contenedor anterior.

Problema: ¡Mi componente no se representa!

- ¿Has añadido el componente a un contenedor utilizando el método `add()` correcto para el controlador de disposición del contenedor? `BorderLayout` (el controlador de disposición por defecto para las ventanas), falla silenciosamente si se añade un componente llamando a la versión de un argumento de `add()`. Puedes ver la lección [Distribuir los Componentes dentro de un Contenedor](#) para ver ejemplos de utilización del método `add()`.
- Si no estás utilizando el controlador de disposición por defecto, ¿has creado satisfactoriamente un ejemplar del controlador de disposición correcto y has llamado al método `setLayout()` del contenedor?
- Si se ha añadido correctamente el componente pero el contenedor ya podría estar visible, ¿has llamado al método `validate()` del contenedor después de añadir el componente?
- Si el componente es un componente personalizado (una subclase `Canvas`, por ejemplo), ¿implementa los métodos `minimumSize()` y `preferredSize()` para devolver el tamaño correcto del componente?
- Si no utilizas un controlador de disposición del AWT, o no utilizas ninguno, ¿tiene el componente un tamaño y unas coordenadas razonables? En particular, si se utiliza posicionamiento absoluto (sin controlador de disposición), se debe seleccionar explícitamente el tamaño de sus componentes, o no se mostrarán. Puedes ver [Hacerlo sin Controlador de Disposición](#).

Problema: Mi componente personalizado no se actualiza cuando debería.

- Asegurate de que se llama al método `repaint()` del componente cada vez que la apariencia del componente deba cambiar. Para los componentes estándar, esto no es un problema, ya que el código específico de la plataforma tiene cuidado de todo el dibujo de los componentes. Sin embargo para los componentes personalizados, tiene que llamar explícitamente al método `repaint()` del componente siempre que deba cambiar su apariencia. Llamar al

método `repaint()` del contenedor del componente no es suficiente. Puedes ver [Cómo Utilizar la Clase Canvas](#) para más información.

Problema: Mi componente no obtiene el evento XYZ.

- Comprueba si el componente obtiene algún evento. Si no lo hace, asegurate de que te estás refiriendo al componente correcto -- que ha ejemplarizado la clase correcta, por ejemplo.
- Asegurate de que el componente pueda capturar la clase de eventos que se quiere capturar. Por ejemplo, muchos componentes estandard no capturan eventos del ratón, por eso el AWT no puede generar objetos `Event` para ellos. ¿Se puede utiliza otro tipo de evento como `ACTION_EVENT` (manejado por el método `action()`), en su lugar? Si no es así, te podrías ver forzado a implementar una subclase de `Canvas` (o de `Panel` o `Applet`) para que la clase pueda ver todos los eventos que ocurran.

Problema: Mi aplicación no puede obtener un evento `WINDOW_DESTROY`, por eso no puede cerrar mi ventana (o salir de la aplicación)!

- **En una subclase de `Frame`**, implementa el método `handleEvent()` para que reaccione al evento `WINDOW_DESTROY`. No se puede capturar eventos `WINDOW_DESTROY` en una subclase de `Panel`, por ejemplo, ya que el `Frame` es superior en el árbol de herencia. Para salir, utilice `System.exit()`. Para destruir la ventana, se puede llamar a `dispose()` o puede `hide()` el `Frame` y, dejar de utilizarlo asegurándose de que todas las referencias se ponen a `null`.

Problema: Todos estos ejemplos son applets. ¿Cómo puedo aplicarlo a las aplicaciones?

- Excepto donde se ha avisado, en cualquier lugar de esta ruta donde vea una subclase de la clase `Applet`, se puede sustituirla por una subclase de la clase `Panel`, o si la clase no es utilizada como contenedor, una subclase de la clase `Canvas`. En general, es sencillo convertir un applet en una aplicación, siempre que el applet no utilice ninguna de las habilidades especiales de los applets (como utilizar métodos definidos en la clase `Applet`).

Para convertir un applet en una aplicación, se necesita añadir un método `main()` que cree un ejemplar de una subclase de `Frame`, cree un ejemplar de una subclase de `Applet` (o `Panel`, o `Canvas`), añada el ejemplar al `Frame`, y luego llame a los métodos `init()` y `start()` del ejemplar. La subclase `Frame` debería tener una implementación de `handleEvent()` que maneje eventos `WINDOW_DESTROY` de la forma apropiada.

Problema: Siempre que ejecuto una aplicación Java con un GUI, obtengo este mensaje:

Warning:

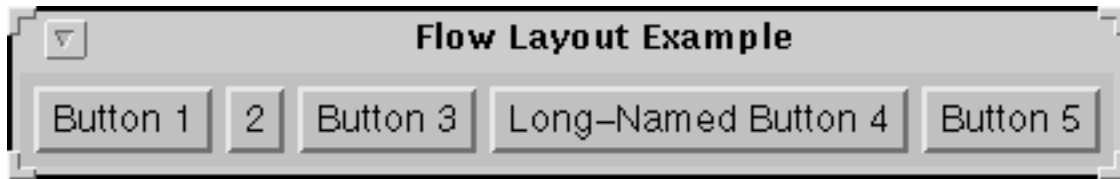
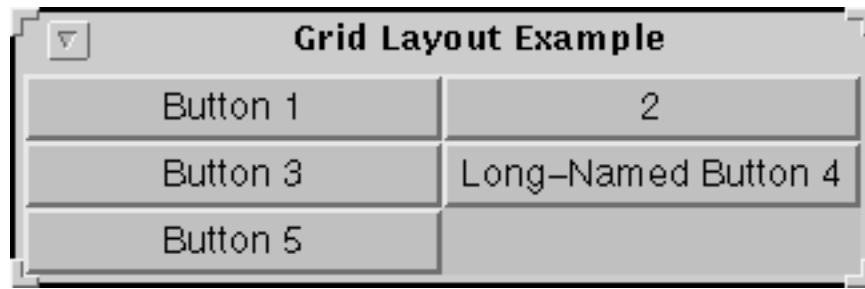
`Cannot allocate colormap entry for default background`

- Este mensaje sólo ocurre en sistemas Motif. Ocurre cuando la librería Motif es inicializada y encuentra que no hay sitio en el colormap por defecto para

asignar sus colores del GUI. La solución es ejecutar aplicaciones más pequeñas en su escritorio. El sistema de ejecución de Java se adapta a sí mismo a los colores de la paleta por defecto, pero la librería Motif es menos generosa.

Si no has visto aquí tu problema puedes ver [Problemas más comunes con la Disposición](#) y, para los componentes personalizados, [Problemas más comunes con los Gráficos](#). También podrías arrojar luz sobre un problema leyendo [Detalles de la Arquitectura de Componentes](#).

Distribuir Componentes dentro de un Contenedor



Arriba tienes las imágenes de dos programas, cada uno de ellos muestra cinco botones. El código Java de los programas es casi idéntico. ¿Entonces por qué parecen tan diferentes? Porque utilizan un controlador de disposición diferente para controlar la distribución de los botones.

Un Controlador de disposición es un objeto que controla el tamaño y posición de los componentes de un contenedor. Los controladores de disposición se adhieren al interface `LayoutManager`. Por defecto, todos los objetos `Container` tiene un objeto `LayoutManager` que controla su distribución. Para los objetos de la clase `Panel`, el controlador de disposición por defecto es un ejemplar de la clase `FlowLayout`. Para los objetos de la clase `Window`, el controlador de disposición por defecto es un ejemplar de la clase `BorderLayout`.

Esta lección cuenta cómo utilizar los controladores de disposición que proporciona el AWT. También le enseña cómo escribir su propio controlador de disposición. Incluso cuenta cómo hacerlo sin utilizar ningún controlador de disposición, especificando el tamaño y posición absolutos de los componentes. Finalmente, esta lección explica algunos de los problemas más comunes con la disposición y sus soluciones.

Utilizar Controladores de Disposición

Aquí es donde se aprenderá cómo utilizar los controladores de disposición. Esta sección le da unas reglas generales y unas instrucciones detalladas del uso de cada uno de los controladores de distribución proporcionados por el AWT.

Crear un Controlador de Disposición Personalizado

En vez de utilizar uno de los controladores de disposición del AWT, se puede crear uno personalizado. Los controladores de disposición deben

implementar el interface `LayoutManager`, que especifica cinco métodos que todo controlador de disposición debe definir.

Hacerlo sin Controlador de Disposición (Posicionamiento Absoluto)

Se pueden posicionar los componentes sin utilizar un controlador de disposición. Generalmente, esta solución se utiliza para especificar posicionamiento absoluto para los componentes, y sólo para aplicaciones que se ejecuten en una única plataforma. El posicionamiento absoluto frecuentemente está contraindicado para los applets y otros programas independientes de la plataforma, ya que el tamaño de los componentes puede diferir de una plataforma a otra.

Problemas más Comunes con la Disposición de Componentes (y sus soluciones)

Algunos de los problemas más comunes con la disposición de componentes son los componentes que se muestran demasiado pequeños o no se muestran. Esta sección le explica como eliminar estos y otros problemas comunes.

Utilizar los Controladores de Disposición

Cada contenedor, tiene un controlador de disposición por defecto -- un objeto que implementa el interface `LayoutManager`. Si el controlador por defecto de un contenedor no satisface sus necesidades, puedes reemplazarlo fácilmente por cualquier otro. El AWT suministra varios controladores de disposición que van desde los más sencillos ([FlowLayout](#) y [GridLayout](#)) a los de propósito general ([BorderLayout](#) y [CardLayout](#)) hasta el ultra-flexible ([GridBagLayout](#)).

Esta lección da algunas reglas generales para el uso de los controladores de disposición, le ofrece una introducción a los controladores de disposición proporcionados por el AWT, y cuenta cómo utilizar cada uno de ellos. En estas páginas encontrarás applets que ilustran los controladores de disposición. Cada applet trae una ventana que se puede redimensionar para ver los efectos del cambio de tamaño en la disposición de los componentes.

Reglas Generales para el uso de Controladores de Disposición

Esta sección responde algunas de las preguntas más frecuentes sobre los controladores de disposición:

- ¿Cómo puedo elegir un controlador de disposición?
- ¿Cómo puedo crear un controlador de disposición asociado con un contenedor, y decirle que empiece a trabajar?
- ¿Cómo sabe un controlador de disposición los componentes que debe manejar?

Cómo Utilizar BorderLayout

`BorderLayout` es el controlador de disposición por defecto para todas las ventanas, como Frames y Cuadros de Diálogo. Utiliza cinco áreas para contener los componentes: north, south, east, west, and center (norte, sur, este, oeste y centro). Todo el espacio extra se sitúa en el área central. Aquí tienes un applet que sitúa un botón en cada área.

Cómo Utilizar CardLayout

Utiliza la clase `CardLayout` cuando tengas un área que pueda contener diferentes componentes en distintos momentos. Los `CardLayout` normalmente son controlados por un objeto `Choice`, con el estado del objeto, se determina que Panel (grupo de componentes) mostrará el `CardLayout`. Aquí tienes un applet que utiliza un objeto `Choice` y un `CardLayout` de esta forma.

Cómo Utilizar FlowLayout

FlowLayout es el controlador por defecto para todos los Paneles. Simplemente coloca los componentes de izquierda a derecha, empezando una nueva línea si es necesario. Los dos paneles en el [applet anterior](#) utilizan FlowLayout. Aquí tienes otro ejemplo de applet que utiliza un FlowLayout.

Cómo utilizar GridLayout

GridLayout simplemente genera un razimo de Componentes que tienen el mismo tamaño, mostrándolos en una sucesión de filas y columnas. Aquí tienes un applet que utiliza un GridLayout para controlar cinco botones.

Cómo Utilizar Use GridBagLayout

GridBagLayout es el más sofisticado y flexible controlador de disposición proporcionado por el AWT. Alínea los componentes situándolos en una parrilla de celdas, permitiendo que algunos componentes ocupen más de una celda. Las filas de la parrilla no tienen porque ser de la misma altura; de la misma forma las columnas pueden tener diferentes anchuras. Aquí tiene un applet que utiliza un GridBagLayout para manejar diez botones en un panel.

Reglas Generales para el uso de los Controladores de Disposición

A menos que se le diga a un contenedor que no utilice un controlador de disposición, el está asociado con su propio ejemplar de un controlador de disposición. Este controlador es consultado automáticamente cada vez que el contenedor necesita cambiar su apariencia. La mayoría de los controladores de disposición no necesitan que los programan llamen directamente a sus métodos.

Cómo Elegir un Controlador de Disposición

Los controladores de disposición proporcionados por el AWT tienen diferentes potencias y puntos débiles. Esta sección descubre algunos de los escenarios de disposición más comunes y cómo podrían trabajar los controladores de disposición del AWT en cada escenario. Si ninguno de los controladores del AWT se adapta a su situación, deberá sentirse libre para utilizar controladores de disposición distribuidos por la red, como un `PackerLayout`.

Escenario: Necesita mostrar un componente en todo el espacio que pueda conseguir.

Considera la utilización de [BorderLayout](#) o [GridBagLayout](#). Si utilizas `BorderLayout`, necesitarás poner el componente que necesite más espacio en el centro. Con `GridBagLayout`, necesitarás seleccionar las restricciones del componente para que `fill=GridBagConstraints.BOTH`. O, si no si no te importa que todos los componentes del mismo contenedor tengan el tamaño tan grande cómo el del componente mayor, puedes utilizar un [GridLayout](#).

Escenario: Se necesita mostrar unos pocos componentes en una línea compacta en su tamaño natural.

Cosidera la utilización de un `Panel` para contener los componentes utilizan el controlado por defecto de la clase `Panel`: [FlowLayout](#).

Ecenario: Necesita mostrar unos componentes con el mismo tamaño en filas y/o columnas.

[GridLayout](#) es perfecto para esto.

Cómo crear un Controlador de Disposición y Asociarlo con un Contenedor

Cada contenedor tiene asociado un controlador de disposición por defecto. Todos los Paneles (incluyendo los Applets) están inicializados para utilizar `FlowLayout`. Todas las ventanas (excepto las de propósito

especial como los `FileDialog`) están inicializadas para utilizar `BorderLayout`.

Si quieres utilizar el controlador de disposición por defecto de un Contenedor, no tiene que hacer nada. El constructor de cada Contenedor crea un ejemplar del controlador de disposición e inicializa el Contenedor para que lo utilice.

Para utilizar un controlador de disposición que no sea por defecto, necesitarás crear un ejemplar de la clase del controlador deseado y luego decirle al contenedor que lo utilice. Abajo hay un código típico que hace esto. Este código crea un controlador `CardLayout` y lo selecciona para utilizarlo con el Contenedor.

```
aContainer.setLayout(new CardLayout());
```

Reglas del Pulgar para utilizar Controladores de Disposición

Los métodos del contenedor que resultan en llamadas al controlador de disposición del Contenedor son :

- `add()`
- `remove()`,
- `removeAll()`,
- `layout()`,
- `preferredSize()`,
- `minimumSize()`.

Los métodos `add()`, `remove()`, `removeAll()` añaden y eliminan Componentes de un contenedor; y puedes llamarlos en cualquier momento.

El método `layout()`, que es llamado como resultado de un petición de dibujado de un Contenedor, petición en la que el Contenedor se sitúa y ajusta su tamaño y el de los componentes que contiene; normalmente no lo llamarás directamente.

Los métodos `preferredSize()` y `minimumSize()` devuelve el tamaño ideal y el tamaño mínimo, respectivamente. Los valores devueltos son sólo apuntes, no tienen efecto a menos que su programa fuerce esto tamaños.

Deberás tener especial cuidado cuando llames a los métodos `preferredSize()` y `minimumSize()` de un contenedor. Los valores devueltos por estos métodos no tienen importancia a menos que el contenedor y sus componentes tengan objetos pares válidos. Puedes ver: [Detalles de la Arquitectura de Componentes](#) para más información sobre cuando se crean los pares.

Cómo Utilizar BorderLayout

Aquí tienes un Applet que muestra un [BorderLayout](#) en acción.

Como se ve en el applet anterior, un BorderLayout tiene cinco áreas: north, south, east, west, y center. Si se agranda la ventana, verás que el área central obtiene todo el espacio nuevo que le sea posible. Las otras áreas sólo se expanden lo justo para llenar todo el espacio disponible.

Abajo tienes el código que crea el BorderLayout y los componentes que éste maneja. Aquí tienes el [programa completo](#). El programa se puede ejecutar dentro de un applet, con la ayuda de [AppletButton](#), o como una aplicación. La primera línea de código realmente no es necesaria para este ejemplo, ya que está en una subclase de Window y todas las subclases de Windows tienen asociado un ejemplar de BorderLayout. Sin embargo, la primera línea sería necesaria si el código estuviera en un un Panel en lugar de una Ventana.

```
setLayout(new BorderLayout());
setFont(new Font("Helvetica", Font.PLAIN, 14));

add("North", new Button("North"));
add("South", new Button("South"));
add("East", new Button("East"));
add("West", new Button("West"));
add("Center", new Button("Center"));
```

Importante: Cuando se añada un componente a un contenedor que utiliza BorderLayout se debe utilizar la versión con dos argumentos del método add(), y el primer argumentos debe ser "North", "South", "East", "West", o "Center". Si se utiliza la versión de un argumento de add(), o si se especifica un primer argumento que no sea válido, su componente podría no ser visible.

Por defecto, un BorderLayout no deja espacio entre los componentes que maneja. En el applet anterior la separación aparente es el resultando de que los botones reservan espacio extra a su alrededor. Se puede especificar el espaciado (en pixels) utilizando el siguiente constructor:

```
public BorderLayout(int horizontalGap, int verticalGap)
```

Cómo Utilizar CardLayout

Aquí tienes un applet que muestra un [CardLayout](#) en acción.

Cómo se ve en el applet anterior, la clase CardLayout ayuda a manejar dos o más componentes (normalmente ejemplares de la clase Panel) que comparten el mismo espacio. Conceptualmente, cada componente tiene un CardLayout que lo maneja como si jugaran a cartas o las colocaran en una pila, donde sólo es visible la carta superior. Se puede elegir la carta que se está mostrando de alguna de las siguientes formas:

- Pidiendo la primera o la última carta, en el orden en el que fueron añadidas al contenedor.
- Saltando a través de la baraja hacia delante o hacia atrás,
- Especificando la carta con un nombre específico. Este es el esquema que utiliza el programa de ejemplo. Específicamente, el usuario puede elegir una carta (componente) eligiendo su nombre en una lista desplegable.

Abajo tienes el código que crea el CardLayout y los componentes que maneja. Aquí tienes el [programa completo](#). El programa se puede ejecutar dentro de un applet con la ayuda de [AppletButton](#), o como una aplicación.

```
//Donde se declaren las variables de ejemplar:
Panel cards;
final static String BUTTONPANEL = "Panel with Buttons";
final static String TEXTPANEL = "Panel with TextField";

//Donde se inicialice el Panel:
cards = new Panel();
cards.setLayout(new CardLayout());

...//Crea un Panel llamado p1. Pone los botones en él.
...//Crea un Panel llamado p2. Pone un campo de texto en él.

cards.add(BUTTONPANEL, p1);
cards.add(TEXTPANEL, p2);
```

Cuando se añaden componentes a un controlador que utiliza un CardLayout, se debe utilizar la forma de dos argumentos del método add() del contenedor: add(String name, Component comp). El primer argumento debe ser una cadena con algo que identifique al componente que se está añadiendo.

Para elegir el componente mostrado por el CardLayout, se necesita algún código adicional. Aquí puedes ver cómo lo hace el applet de esta página:

```
//Donde se inicialice el contenedor:
. . .
    //Pone la lista en un Panel para que tenga un buen aspecto.
```

```
Panel cp = new Panel();
Choice c = new Choice();
c.addItem(BUTTONPANEL);
c.addItem(TEXTPANEL);
cp.add(c);
add("North", cp);
```

. . .

```
public boolean action(Event evt, Object arg) {
    if (evt.target instanceof Choice) {
        ((CardLayout)cards.getLayout()).show(cards,(String)arg);
        return true;
    }
    return false;
}
```

Como se muestra en el código anterior, se puede utilizar el método `show()` de `CardLayout` para seleccionar el componente que se está mostrando. El primer argumento del método `show()` es el contenedor que controla `CardLayout` -- esto es, el contenedor de los componentes que maneja `CardLayout`. El segundo argumento es la cadena que identifica el componente a mostrar. Esta cadena es la misma que fue utilizada para añadir el componente al contenedor.

Abajo tienes todos los métodos de `CardLayout` que permiten seleccionar un componente. Para cada método, el primer argumento es el contenedor del que `CardLayout` es el controlador de disposición (el contenedor de las cartas que controla `CardLayout`).

```
public void first(Container parent)
public void next(Container parent)
public void previous(Container parent)
public void last(Container parent)
public void show(Container parent, String name)
```

Cómo Utilizar FlowLayout

Aquí tienes un applet que muestra un [FlowLayout](#) en acción.

Como se ve en el applet anterior, FlowLayout pone los componentes en una fila, ajustándolos a su tamaño preferido. Si el espacio horizontal del contenedor es demasiado pequeño para poner todos los componentes en una fila, FlowLayout utiliza varias filas. Dentro de cada fila, los componentes se colocan centrados (por defecto), alineados a la izquierda o a la derecha según se especifique cuando se cree el FlowLayout.

Abajo tienes el código que crea el FlowLayout y los componentes que maneja. Aquí tienes el [programa completo](#). El programa puede ejecutarse dentro de un applet, con la ayuda de [AppletButton](#), o como una aplicación.

```
setLayout(new FlowLayout());  
setFont(new Font("Helvetica", Font.PLAIN, 14));  
  
add(new Button("Button 1"));  
add(new Button("2"));  
add(new Button("Button 3"));  
add(new Button("Long-Named Button 4"));  
add(new Button("Button 5"));
```

La clase FlowLayout tiene tres constructores:

```
public FlowLayout()  
public FlowLayout(int alignment)  
public FlowLayout(int alignment, int horizontalGap, int verticalGap)
```

El Argumento alignment debe tener alguno de estos valores:

- FlowLayout.RIGHT
- FlowLayout.CENTER
- FlowLayout.LEFT

Los argumentos horizontalGap y verticalGap especifican el número de pixels del espacio entre los componentes. Si no se especifica ningún valor, FlowLayout actúa como si se hubieran especificado 5 pixels.

Cómo Utilizar GridLayout

Aquí tienes un applet que muestra un [GridLayout](#) en acción.

Como se ve en el applet anterior, un GridLayout sitúa los componentes en una parrilla de celdas. Cada componente utiliza todo el espacio disponible en su celda, y todas las celdas son del mismo tamaño. Si se redimensiona el tamaño de la ventana GridLayout, verás que el GridLayout cambia el tamaño de las celdas para que sean lo más grandes posible, dando el espacio disponible al contenedor.

Abajo está el código que crea el GridLayout y los componentes que maneja. Aquí tienes el [programa completo](#). El programa puede ejecutarse dentro de un applet, con la ayuda de [AppletButton](#), o como una aplicación.

```
//Construye un GridLayout con dos 2 columnas y un número no especificado de filas.  
setLayout(new GridLayout(0,2));  
setFont(new Font("Helvetica", Font.PLAIN, 14));
```

```
add(new Button("Button 1"));  
add(new Button("2"));  
add(new Button("Button 3"));  
add(new Button("Long-Named Button 4"));  
add(new Button("Button 5"));
```

El constructor le dice a la clase GridLayout que cree un ejemplar que tiene dos columnas y tantas filas como sea necesario. Es uno de los dos constructores de GridLayout. Aquí tienes las declaraciones de los dos constructores:

- `public GridLayout(int rows, int columns)`
- `public GridLayout(int rows, int columns, int horizontalGap, int verticalGap)`

Al menos uno de los argumentos rows o columns debe ser distinto de cero. Los argumentos horizontalGap y verticalGap del segundo constructor permiten especificar el número de pixels entre las celdas. Si no se especifica el espacio, sus valores por defecto son cero. (En el applet anterior, cualquier apariencia de espaciado es el resultado de que los botones reservan espacio extra sobre sus propias áreas.)

Cómo Utilizar GridBagLayout

Aquí tienes un applet que muestra un [GridBagLayout](#) en acción.

GridBagLayout es el más flexible - y complejo - controlador de disposición proporcionado por el AWT. Como se ve en el applet anterior, un GridBagLayout, sitúa los componentes en una parrilla de filas y columnas, permitiendo que los componentes se expandan más de una fila o columna. No es necesario que todas las filas tengan la misma altura, ni que las columnas tengan la misma anchura. Esencialmente, GridBagLayout sitúa los componentes en celdas en una parrilla, y luego utiliza los tamaños preferidos de los componentes que determina cómo debe ser el tamaño de la celda.

Si se agranda la ventana que trae el applet, observarás que la última fila obtiene un nuevo espacio vertical, y que el nuevo espacio horizontal es dividido entre todas las columnas. Este comportamiento está basado en el peso que el applet asigna a los componentes individuales del GridBagLayout. Observa también, que cada componente toma todo el espacio que puede. Este comportamiento también está especificado por el applet.

La forma en que el applet especifica el tamaño y la posición característicos de sus componentes está especificado por las obligaciones de cada componente. Para especificar obligaciones, debe seleccionar las variables de ejemplar en un objeto GridBagConstraints y decírselo al GridBagLayout (con el método `setConstraints()`) para asociar las obligaciones con el componente.

Las siguientes páginas explican las obligaciones que se pueden seleccionar y proporciona ejemplos de ellas.

[Especificar Obligaciones](#)

Esta página muestra las variables que tiene un objeto GridBagConstraints, qué valores pueden tener, y cómo asociar el objeto GridBagConstraints resultante con un componente.

[El Applet de Ejemplo Explicado](#)

En esta página se pone todo junto, explicando el código del applet de esta página.

Cómo Utilizar GridBagLayout: Especificar Obligaciones

Abajo tienes parte del código que podrás ver en un contenedor que utiliza un [GridBagLayout](#). (Verás un ejemplo explicado en la página siguiente).

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setLayout(gridbag);

//Para cada componentes que sea añadido a este contenedor:
//...Crea el componente...
//...Seleccionar las variables de ejemplar en el objeto GridBagConstraints...
gridbag.setConstraints(theComponent, c);
add(theComponent);
```

Como podrías haber deducido del ejemplo anterior, se puede reutilizar el mismo ejemplar de GridBagConstraints para varios componentes, incluso si los componentes tienen distintas obligaciones. El GridBagLayout extrae los valores de las obligaciones y no vuelve a utilizar el GridBagConstraints. Sin embargo, se debe tener cuidado de resetar las variables de ejemplar del GridBagConstraints a sus valores por defecto cuando sea necesario.

Puede seleccionar las siguientes variables de ejemplar del [GridBagConstraints](#):

gridx, gridy

Especifica la fila y la columna de la esquina superior izquierda del componente. La columna más a la izquierda tiene la dirección gridx= 0, y la fila superior tiene la dirección gridy= 0. Utiliza GridBagConstraints.RELATIVE (el valor por defecto) para especificar que el componente debe situarse a la derecha (para gridx) o debajo (para gridy) del componente que se añadió al contenedor inmediatamente antes.

gridwidth, gridheight

Especifica el número de columnas (para gridwidth) o filas (para gridheight) en el área de componente. Esta obligación especifica el número de celdas utilizadas por el componente, no el número de pixels. El valor por defecto es 1. Utiliza GridBagConstraints.REMAINDER para especificar que el componente será el último de esta fila (para gridwidth) o columna (para gridheight). Utiliza GridBagConstraints.RELATIVE para especificar que el componente es el siguiente para el último de esta fila (para gridwidth) o columna (para gridheight).

Nota: Debido a un error en la versión 1.0 de Java, GridBagLayout no permite un componente se expanda varias columnas a menos que sea el primero por la izquierda.

fill

Utilizada cuando el área del pantalla del componentes es mayor que el tamaño requerido por éste para determinar si se debe, y cómo redimensionar el componente. Los valores válidos son GridBagConstraints.NONE (por defecto), GridBagConstraints.HORIZONTAL (hace que el componente tenga suficiente anchura para llenar horizontalmente su área de dibujo, pero no cambia su altura), GridBagConstraints.VERTICAL (hace que el componente sea lo suficientemente alto para llenar verticalmente su área de dibujo, pero no cambia su anchura), y GridBagConstraints.BOTH (hace que el componente llene su área de dibujo por completo).

ipadx, ipady

Especifica el espacio interno: cuánto se debe añadir al tamaño mínimo del componente. El valor por defecto es cero. La anchura del componente debe ser al menos su anchura mínima más $\text{ipadx} * 2$ pixels (ya que el espaciado se aplica a los dos lados del componente). De forma similar, la altura de un componente será al menos su altura mínima más $\text{ipady} * 2$ pixels.

insets

Especifica el espaciado externo del componente -- la cantidad mínima de espacio entre los componentes y los bordes del área de dibujo. Es valor es especificado como un objeto [Insets](#). Por defecto, ningún componente tiene espaciado externo.

anchor

Utilizado cuando el componente es más pequeño que su área de dibujo para determinar dónde (dentro del área) situar el componente. Los valores válidos son :

- GridBagConstraints.CENTER (por defecto),
- GridBagConstraints.NORTH,
- GridBagConstraints.NORTHEAST,
- GridBagConstraints.EAST,
- GridBagConstraints.SOUTHEAST,
- GridBagConstraints.SOUTH,
- GridBagConstraints.SOUTHWEST,
- GridBagConstraints.WEST,
- GridBagConstraints.NORTHWEST.

weightx, weighty

Especificar el peso es un arte que puede tener un impacto importante en la apariencia de los componentes que controla un GridBagLayout. El peso es utiliza para determinar cómo distribuir el espacio entre columnas (weightx) y filas (weighty); esto es importante para especificar el comportamiento durante el redimensionado.

A menos que se especifique un valor distinto de cero para weightx o weighty, todos los componente se situarán juntos en el centro de su contenendor. Esto es así porque cuando el peso es 0,0 (el valor por defecto) el GidBagLayout pone todo el espacio extra entre las celdas y los bordes del contenedor.

Generalmente, los pesos son especificados con 0.0 y 1.0 como los extremos, con números entre ellos si son necesarios. Los números mayores indican que la fila o columna del componente deberían obtener más espacio. Para cada columna, su peso está relacionado con el mayor weightx especificado para un componente dentro de esa columna (donde cada componete que ocupa varias columnas es dividido de alguna forma entre el número de columnas que ocupa). Lo mismo ocurre con las filas para el mayor valor especificado en weighty.

La página siguiente explica las oblicaciones en más detalle, explicando cómo trabaja el applet del ejemplo.

Cómo Utilizar GridBagLayout: El Applet de ejemplo Explicado

De nuevo, aquí está el applet que muestra un GridBagLayout en acción.

Abajo tienes el código que crea el GridBagLayout y los componentes que maneja. Aquí tienes el [programa completo](#). El programa puede ejecutarse dentro de un applet, con la ayuda de [AppletButton](#), o como una aplicación.

```
protected void makebutton(String name, GridBagLayout gridbag, GridBagConstraints c) {
    Button button = new Button(name);
    gridbag.setConstraints(button, c);
    add(button);
}

public GridBagWindow() {
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();

    setFont(new Font("Helvetica", Font.PLAIN, 14));
    setLayout(gridbag);

    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    makebutton("Button1", gridbag, c);
    makebutton("Button2", gridbag, c);
    makebutton("Button3", gridbag, c);

    c.gridwidth = GridBagConstraints.REMAINDER; //Final de fila
    makebutton("Button4", gridbag, c);

    c.weightx = 0.0; //resetea a los valores por defecto
    makebutton("Button5", gridbag, c); //otra fila

    c.gridwidth = GridBagConstraints.RELATIVE; /sigua al último de la fila
    makebutton("Button6", gridbag, c);

    c.gridwidth = GridBagConstraints.REMAINDER; //fin de fila
    makebutton("Button7", gridbag, c);

    c.gridwidth = 1; //resetea a los valores por defecto
    c.gridheight = 2;
    c.weighty = 1.0;
    makebutton("Button8", gridbag, c);

    c.weighty = 0.0; //resetea a los valores por defecto
    c.gridwidth = GridBagConstraints.REMAINDER; //fin de fila
    c.gridheight = 1; //resetea a los valores por defecto
    makebutton("Button9", gridbag, c);
    makebutton("Button10", gridbag, c);
}
```

Este ejemplo utiliza un sólo ejemplar de GridBagConstraints para todos los componetes manejados por el GridBagLayout. Justo antes de que cada componente sea añadido al contenedor, el código selecciona (o resetea a los valores por defecto) las variables apropiadas del objeto GridBagConstraints. Luego utiliza el método setConstraints() para grabar los valores obligatorios de ese componente.

Por ejemplo, justo antes de añadir un componente al final de una fila verá el siguiente código:

```
c.gridwidth = GridBagConstraints.REMAINDER; //final de fila
```

Y justo antes de añadir el siguiente componente (si el siguiente componente no ocupa una fila completa), verás la misma variable reseteada a su valor por defecto:

```
c.gridwidth = 1; //resetea al valor por defecto
```

Para mayor claridad aquí tienes una tabla que muestra todas las obligaciones para cada componente manejado por GridBagLayout. Los valores que no son por defecto están marcados en negrita. Los valores que son diferentes de su entrada anterior en la tabla están marcados en *itálica*.

Component	Constraints
-----	-----
All components	<code>gridx = GridBagConstraints.RELATIVE</code> <code>gridy = GridBagConstraints.RELATIVE</code> <code>fill = GridBagConstraints.BOTH</code> <code>ipadx = 0, ipady = 0</code> <code>insets = new Insets(0,0,0,0)</code> <code>anchor = GridBagConstraints.CENTER</code>
Button1, Button2, Button3	<code>gridwidth = 1</code> <code>gridheight = 1</code> <code>weightx = 1.0</code> <code>weighty = 0.0</code>
Button4	<i><code>gridwidth = GridBagConstraints.REMAINDER</code></i> <code>gridheight = 1</code> <code>weightx = 1.0</code> <code>weighty = 0.0</code>
Button5	<code>gridwidth = GridBagConstraints.REMAINDER</code> <code>gridheight = 1</code> <i><code>weightx = 0.0</code></i> <code>weighty = 0.0</code>
Button6	<i><code>gridwidth = GridBagConstraints.RELATIVE</code></i> <code>gridheight = 1</code> <code>weightx = 0.0</code> <code>weighty = 0.0</code>
Button7	<i><code>gridwidth = GridBagConstraints.REMAINDER</code></i> <code>gridheight = 1</code> <i><code>weightx = 0.0</code></i> <code>weighty = 0.0</code>
Button8	<i><code>gridwidth = 1</code></i> <code>gridheight = 2</code> <code>weightx = 0.0</code> <code>weighty = 1.0</code>
Button9, Button10	<i><code>gridwidth = GridBagConstraints.REMAINDER</code></i> <i><code>gridheight = 1</code></i> <i><code>weightx = 0.0</code></i> <i><code>weighty = 0.0</code></i>

Todos los componentes de este contenedor son tan grandes como sea aposible, dependiendo de su

fila y columna. El programa consigue esto selección la variable fill de GridBagConstraints a GridBagConstraints.BOTH, dejándola seleccionada para todos los componentes.

Este programa tiene cuatro componentes que se expanden varias columnas (Button5, Button6, Button9, and Button10) y uno que se expande varias filas (Button8). Sólo en uno de estos caso (Button8) se especifica explícitamente la anchura y altura del componente. En los otros casos, la anchura de los componentes está especificada como GridBagConstraints.RELATIVE o GridBagConstraints.REMAINDER, lo que le permite a GridBagLayout determinar el tamaño del componente, teniendo en cuenta el tamaño de los otros componentes de la fila.

Cuando se agrande la ventana del programa, observará que la anchura de las columnas crece los mismo que la ventana. Este es el resultado de cada componente de la primer fila (donde cada componente tiene la anchura de una columna) tiene `weightx = 1.0`. El valor real del `weightx` de estos componentes no tiene importancia. Lo que importa es que todos los componentes (y así todas las columnas) tienen el mismo peso que es mayor que cero. Si ningún componente manejado por el GridBagLayout tuviera seleccionado `weightx`, cuando se ampliara la anchura del contenedor, los componentes permanecerían juntos en el centro del contenedor.

Otra cosa que habrás observado es que cuando se aumenta la altura de la ventana la última línea es la única que se agranda. Esto es así porque el Button8 tiene `weighty` mayor que cero. Button8 se expande dos filas, y el GridBagLayout hace que el peso de este botón asigne todos el espacio a las dos últimas filas ocupadas por él.

Crear un Controlador de Disposición Personalizado

Nota: Antes de empezar a crear un controlador de disposición personalizado, asegurate de que no existe ningún controlador que trabaje de la forma apropiada. En particular, [GridBagLayout](#) es lo suficientemente flexible para trabajar en muchos casos.

Para crear un controlador de disposición personalizado, debes crear un clase que implemente el interface [LayoutManager](#). LayoutManager requiere que se implementen estos cinco métodos:

`public void addLayoutComponent(String name, Component comp)`

Llamado sólo por el método `add(name, component)` del contenedor. Los controladores de disposición que no necesitan que sus componentes tengan nombres eneralmente no hacen nada en este método.

`public void removeLayoutComponent(Component comp)`

Llamado por los métodos `remove()` y `removeAll()` del contenedor. Los controladores de disposición que no necesitan que sus componentes tengan nombres generalmente no hacen nada en este método ya que pueden preguntarle al contenedor por sus componentes utilizando el método `getComponents()` del contenedor.

`public Dimension preferredLayoutSize(Container parent)`

Llamado por el método `preferredSize()` de contenedor, que es llamado a sí mismo bajo una variedad de circunstancias. Este método debería calcular el tamaño ideal del padre, asumiendo que los componentes contendidos tendrán aproximadamente su tamaño preferido. Este método debe tomar en cuenta los bordes internos del padre (devueltos por el método `insets()` del contenedor).

`public Dimension minimumLayoutSize(Container parent)`

Llamado por el método `minimumSize()` del contenedor, que a su vez es llamado bajo varias circunstancias. Este método debería calcular el tamaño mínimo del padre, asumiendo que todos los componentes que contiene tendrán aproximadamente su tamaño mínimo. Este método debe tomar en cuenta los bordes internos del padre (devueltos por el método `insets()` del contenedor).

`public void layoutContainer(Container parent)`

Llamado cuando el contenedor se muestra por primera vez, y cada vez que cambia su tamaño. El método `layoutContainer()` del controlador de disposición realmente no dibuja los Componentes. Simplemente invoca los métodos [resize\(\)](#), [move\(\)](#), [yreshape\(\)](#) para seleccionar el tamaño y posición de los componentes. Este método debe tomar en cuenta los bordes internos del padre (devueltos por el método `insets()` del contenedor). No se puede asumir que los métodos `preferredLayoutSize()` o

`minimumLayoutSize()` serán llamados antes de que se llame al `layoutContainer()`.

Junto a la implementación de estos cinco métodos requeridos por `LayoutManager`, los controladores de disposición generalmente implementan al menos un constructor público y el método [toString\(\)](#).

Aquí tienes el [código fuente](#) para controlador de disposición personalizado llamado `DiagonalLayout`. Sitúa sus componentes diagonalmente, de izquierda a derecha con un componente por fila.

Aquí tienes un ejemplo de `DiagonalLayout`:

Ozito

Hacerlo sin Controlador de Disposición (Posicionamiento Absoluto)

Aunque es posible hacerlo sin un controlador de disposición, se debería utilizar un controlador de disposición siempre que sea posible. Los controladores de disposición hacen más sencillo en redimensionado de un contenedor y ajustan a la apariencia de los componentes dependientes de la plataforma y los diferentes tamaños de las fuentes. También pueden ser reutilizados fácilmente por otros contenedores y otros programas. Si un contenedor personalizado no será reutilizado ni redimensionado, y controla normalmente los factores dependientes del sistema como el tamaño de las fuentes y la apariencia de los componentes (implementando sus propios controles si fuera necesario), entonces, el posicionamiento absoluto podría tener sentido.

Aquí tienes un applet que muestra una ventana que utiliza posicionamiento absoluto.

Abajo tienes las declaraciones de las variables de ejemplar, la implementación del constructor, y del método paint() de la clase window. Aquí tienes un enlace al [programa completo](#). El programa se puede ejecutar dentro de un applet con la ayuda de [AppletButton](#), o como una aplicación.

```
public class NoneWindow extends Frame {  
    . . .  
    private boolean laidOut = false;  
    private Button b1, b2, b3;  
  
    public NoneWindow() {  
        super();  
        setLayout(null);  
        setFont(new Font("Helvetica", Font.PLAIN, 14));  
  
        b1 = new Button("one");  
        add(b1);  
        b2 = new Button("two");  
        add(b2);  
        b3 = new Button("three");  
        add(b3);  
    }  
  
    public void paint(Graphics g) {  
        if (!laidOut) {  
            Insets insets = insets();  
            /*  
             *Garantizamos que insets() devuelve un Insets válido  
             * si lo llamamos desde paint() -- no sería válido si los llamaramos  
desde  
             * el constructor.  
             *  
             * Quizás podríamos guardar esto en una variable, pero insets puede  
             * cambiar, y cuando lo haga, el AWT crea un nuevo objeto  
             * Insets completo; el viejo no es válido.  
             */  
            b1.reshape(50 + insets.left, 5 + insets.top, 50, 20);
```

```
b2.reshape(70 + insets.left, 35 + insets.top, 50, 20);  
b3.reshape(130 + insets.left, 15 + insets.top, 50, 30);
```

```
laidOut = true;
```

```
}
```

```
}
```

```
...  
}
```

Problemas más Comunes con la Distribución de Componentes (y sus Soluciones)

Problema: ¿Cómo puedo especificar el tamaño exacto de un componente?

- Primero, asegúrate de que realmente necesitas seleccionar el tamaño exacto del componente. Los componentes standards tienen distintos tamaños, dependiendo de la plataforma donde se están ejecutando y de la fuente utilizada, por eso normalmente no tiene sentido especificar su tamaño exacto.

Para los componentes personalizados que tienen contenidos de tamaño fijo (como imágenes), especificar el tamaño exacto tiene sentido. Para componentes personalizados, necesitan sobrescribir los métodos `minimumSize()` y `preferredSize()` del componente para devolver el tamaño correcto del componente.

Para cambiar el tamaño de un componente que ya ha sido dibujado, puedes ver el siguiente problema.

Nota: Todos los tamaños de los componentes están sujetos a la aprobación del controlador de disposición. Los controladores `FlowLayout` y `GridBagLayout` utilizan el tamaño natural de los componentes (el último dependiendo de las obligaciones que usted seleccione), pero `BorderLayout` y `GridLayout` no. Otras opciones son escribir o encontrar un controlador de disposición personalizado o utilizando posicionamiento absoluto.

Problema: ¿Cómo puedo redimensionar un Componente?

- Una vez que el componente ha sido dibujado, puedes cambiar su tamaño utilizando el método `resize()` del Componente. Luego necesitas llamar al método `validate()` del contenedor para asegurarse de que éste se muestre de nuevo.

Problema: Mi componente personalizado se dibuja demasiado pequeño.

- ¿Has implementado los métodos `preferredSize()` and `minimumSize()` del componente? Si lo has hecho, ¿devuelven los valores correctos?
- ¿Estás utilizando un controlador de disposición que puede utilizar todo el espacio disponible? Puede ver [Reglas Generales para el Uso de Controladores de Disposición](#) para ver algunas situaciones de elección del controlador de disposición y especificar que utilice el máximo espacio disponible para un componente particular.

Si no has visto tu problema en esta lista, puedes ver [Problemas más Comunes con los Componentes](#).

Introducción al Soporte de Gráficos del AWT

Como se aprendió en la página [Drawing](#), el sistema de dibujo del AWT controla cuándo y cómo pueden dibujar los programas. En respuesta a una llamada al método `repaint()` de un componente, el AWT invoca al método `update()` del componente, para pedir que el componente se dibuje a sí mismo. El método `update()` (por defecto) invoca al método `paint()` del componente.

Una ayuda adicional en este sistema es que alguna veces el AWT llama directamente al método `paint()` en vez de llamar al método `update()`. Esto sucede casi siempre como resultado de una reacción del AWT ante un estímulo externo, como que el componente aparezca por primera vez en la pantalla, o el componente sea descubierto tras ocultarse por otra ventana. Aprenderás más sobre los métodos `paint()` y `update()` en la explicación [Eliminar el Parpadeo](#), más adelante en esta lección.

El Objeto Graphics

El único argumento para los métodos `paint()` y `update()` es un objeto [Graphics](#). Los objetos `Graphics` son la clave para todo el dibujo. Soportan las dos clases básicas de dibujo: gráficos primitivos (como líneas, rectángulos y texto) y las imágenes. Aprenderás sobre los gráficos primitivos en [Utilizar Gráficos Primitivos](#). Aprenderás sobre las imágenes en [Utilizar Imágenes](#).

Junto con los métodos suministrados para dibujar gráficos primitivos y las imágenes en la pantalla, un objeto `Graphics` proporciona un contexto de dibujo manteniendo estados, como el área de dibujo actual o el color de dibujo actual. Se puede disminuir el área de dibujo actual recortándola, pero nunca se podrá incrementar el tamaño del área de dibujo. De esta forma el objeto `Graphics` se asegura que los componentes sólo puedan dibujar dentro de su área de dibujo. Aprenderás más sobre el recorte en [Sobreescribir el Método `update\(\)`](#).

El Sistema de Coordenadas

Cada componente tiene su propio sistema de coordenadas enteras, que va desde (0,0) hasta (width - 1, height - 1), donde cada unidad representa el tamaño de un pixel. La esquina superior izquierda del área de dibujo del componente es (0,0). La coordenada X se incrementa hacia la derecha, y la coordenada Y se incrementa hacia abajo.

Aquí tienes un applet que construiremos más adelante en esta lección. Siempre que pulse dentro del área enmarcada, el applet dibuja un punto donde se pulsó el ratón y muestra una cadena describiendo donde

ocurrió la pulsación.

Las Cuatro Formas del Método repaint()

Recuerda que el programa puede llamar al método repaint() del componente para pedir que el AWT llame al método update() del componente. Aquí tienes la descripción de las cuatro formas del método repaint():

`public void repaint()`

Pide al AWT que llame al método update() del componente tan pronto como sea posible. Esta es la forma más frecuentemente utilizada de repaint().

`public void repaint(long time)`

Pide al AWT que llame al método update() del componente dentro de time milisegundos desde ahora.

`public void repaint(int x, int y, int width, int height)`

Pide al AWT que llame al método update() del componente tan pronto como sea posible, pero redibujando sólo la parte especificada del componente.

`public void repaint(long time, int x, int y, int width, int height)`

Pide al AWT que llame al método update() del componente dentro de time milisegundos desde ahora, pero redibujando sólo la parte especificada del componente.

Dibujar Formas Sencillas

La clase [Graphics](#) define métodos para dibujar los siguientes tipos de formas:

- Líneas (`drawLine()`, que dibuja una línea en el color actual del objeto `Graphics`, que es inicializado con el color de primer plano del Componente)
- Rectángulos (`drawRect()`, `fillRect()`, y `clearRect()` -- donde `fillRect()` rellena un rectángulo con el color actual del objeto `Graphics`, y `clearRect()` rellena un rectángulo con el color de fondo del Componente)
- Rectángulos en 3 dimensiones (`draw3DRect()` y `fill3DRect()`)
- Rectángulos con los bordes redondeados (`drawRoundRect()` y `fillRoundRect()`)
- Ovalos (`drawOval()` y `fillOval()`)
- Arcos (`drawArc()` y `fillArc()`)
- Polígonos (`drawPolygon()` y `fillPolygon()`)

Excepto para los polígonos y las líneas todas las formas son específicas utilizando su rectángulo exterior. Una vez que hayas comprendido el dibujo de rectángulos, las otras formas son relativamente sencillas. Por esta razón, esta página se concentra en el dibujo de rectángulos.

Ejemplo 1: Dibujar un Rectángulo Sencillo

El applet de la página anterior utilizaba los métodos `draw3DRect()` y `fillRect()` para dibujar su interface. Aquí tienes el applet de nuevo:

Aquí puedes ver el [código](#). Abajo sólo tienes el código de dibujo:

```
//en FramedArea (una subclase de Panel):
public void paint(Graphics g) {
    Dimension d = size();
    Color bg = getBackground();

    //Dibuja un marco divertido alrededor del applet.
    g.setColor(bg);
    g.draw3DRect(0, 0, d.width - 1, d.height - 1, true);
    g.draw3DRect(3, 3, d.width - 7, d.height - 7, false);
}

//en CoordinateArea (una subclase de Canvas):
public void paint(Graphics g) {
    //Si el usuario pulsa el ratón, dibuja un rectángulo pequeño en esa posición
    if (point != null) {
        g.fillRect(point.x - 1, point.y - 1, 2, 2);
    }
}
```

El applet crea (y contiene) un objeto `FramedArea`, que crea (y contiene) un objeto `CoordinateArea`. La primera llamada a `draw3DRect()` crea un rectángulo tan grande como el área de dibujo del `FramedArea`. El argumento `true` especifica que el rectángulo debe aparecer elevado. La segunda llamada a `draw3DRect()` crea un segundo rectángulo un poquito menor, con `false` especifica que el rectángulo deberá aparecer embebido. Las dos llamadas juntas producen el efecto de un marco elevado que contiene el `CoordinateArea`. (`FramedArea` implementa el método `insets()` para que el área de dibujo de la `CoordinateArea` esté unos pixels dentro del `FramedArea`.)

El `CoordinateArea` utiliza `fillRect()` para dibujar un rectángulo de 2x2 pixels en el punto en el que el usuario pulsa el botón del ratón.

Ejemplo 2: Utilizar un Rectángulo para Indicar un Area Seleccionada

Aquí tienes un applet que podrías utilizar para implementar la selección básica en un programa de dibujo. Cuando el usuario mantiene pulsado el botón del ratón, el applet muestra continuamente un rectángulo. El rectángulo empieza en la posición del cursor cuando el usuario pulsó el botón del ratón por primera vez y termina en la posición actual del cursor.

Aquí tienes el [código](#) del applet. Abajo tienes el código más importante:

```
class SelectionArea extends Canvas {
    . . .

    public boolean mouseDown(Event event, int x, int y) {
        currentRect = new Rectangle(x, y, 0, 0);
        repaint();
        return false;
    }

    public boolean mouseDrag(Event event, int x, int y) {
        currentRect.resize(x - currentRect.x, y - currentRect.y);
        repaint();
        return false;
    }

    public boolean mouseUp(Event event, int x, int y) {
        currentRect.resize(x - currentRect.x, y - currentRect.y);
        repaint();
        return false;
    }

    public void paint(Graphics g) {
        Dimension d = size();

        //Si existe currentRect exists, dibuja un rectángulo.
        if (currentRect != null) {
            Rectangle box = getDrawableRect(currentRect, d);
            controller.rectChanged(box);

            //Draw the box outline.
            g.drawRect(box.x, box.y, box.width - 1, box.height - 1);
        }
    }

    Rectangle getDrawableRect(Rectangle originalRect, Dimension drawingArea) {
        . . .
        //Asegurese de que las dimensiones de altura y anchura del rectángulo son
        positiva.
        . . .
        //El rectángulo no debe sobrepasar el área de dibujo.
        . . .
    }
}
```

Como puedes ver, el SelectionArea sigue la pista del rectángulo seleccionado actualmente, utilizando un objeto Rectangle llamado currentRect. Debido a la implementación, el currentRect mantiene el mismo origen (currentRect.x, currentRect.y) mientras el usuario arrastre el ratón. Esto significa que la altura y anchura del rectángulo podrían ser

negativas.

Sin embargo, los métodos `drawXxx()` y `fillXxx()` no dibujarán nada si su altura o anchura son negativos. Por esta razón, cuando `SelectionArea` dibuja un rectángulo, debe especificar el vértice superior izquierdo del rectángulo para que la altura y la anchura sean positivas. La clase `SelectionArea` define el método `getDrawableRect()` para realizar los cálculos necesarios para encontrar el vértice superior izquierdo. El `getDrawableRect()` método también se asegura de que el rectángulo no sobrepase los límites de su área de dibujo. Aquí tienes de nuevo un enlace al [código fuente](#). Encontrarás la definición de `getDrawableRect()` al final del fichero.

Nota: Es perfectamente legal especificar valores negativos para `x`, `y`, `height` o `width` o hacer que el resultado sea mayor que el área de dibujo. Los valores fuera del área de dibujo no importan demasiado porque son recortados al área de dibujo. Lo único es que no verás una parte de la forma. La altura o anchura negativas dan como resultado que no se dibujará nada en absoluto.

Ejemplo 3: Un Ejemplarizador de Formas

El siguiente applet demuestra todas las formas que se pueden dibujar y rellenar.

A menos que la fuente por defecto de su visualizador de applets sea muy pequeña, el texto mostrado en el applet anterior podría parecer demasiado grande en ocasiones. Las palabras podrían dibujarse unas sobre otras. Y como este applet no utiliza el método `insets()` para proteger sus límites el texto podría dibujarse sobre el marco alrededor del applet. La siguiente página amplía este ejemplo, enseñándolo como hacer que el texto quepa en un espacio dado.

Aquí tienes el [código](#) para el applet anterior. Abajo sólo tienes el código que dibuja las formas geométricas. Las variables `rectHeight` y `rectWidth` especifican el tamaño en pixels del área en que debe dibujarse cada forma. Las variables `x` y `y` se cambian para cada forma, para que no se dibujen unas sobre otras.

```
Color bg = getBackground();
Color fg = getForeground();
. . .

// drawLine()
g.drawLine(x, y+rectHeight-1, x + rectWidth, y); // x1, y1, x2, y2
. . .

// drawRect()
g.drawRect(x, y, rectWidth, rectHeight); // x, y, width, height
. . .

// draw3DRect()
g.setColor(bg);
g.draw3DRect(x, y, rectWidth, rectHeight, true);
g.setColor(fg);
. . .

// drawRoundRect()
g.drawRoundRect(x, y, rectWidth, rectHeight, 10, 10); // x, y, w, h, arcw, arch
. . .

// drawOval()
g.drawOval(x, y, rectWidth, rectHeight); // x, y, w, h
. . .
```

```

// drawArc()
g.drawArc(x, y, rectWidth, rectHeight, 90, 135); // x, y, w, h
. . .

// drawPolygon()
Polygon polygon = new Polygon();
polygon.addPoint(x, y);
polygon.addPoint(x+rectWidth, y+rectHeight);
polygon.addPoint(x, y+rectHeight);
polygon.addPoint(x+rectWidth, y);
//polygon.addPoint(x, y); //don't complete; fill will, draw won't
g.drawPolygon(polygon);
. . .

// fillRect()
g.fillRect(x, y, rectWidth, rectHeight); // x, y, width, height
. . .

// fill3DRect()
g.setColor(bg);
g.fill3DRect(x, y, rectWidth, rectHeight, true);
g.setColor(fg);
. . .

// fillRoundRect()
g.fillRoundRect(x, y, rectWidth, rectHeight, 10, 10); // x, y, w, h, arcw, arch
. . .

// fillOval()
g.fillOval(x, y, rectWidth, rectHeight); // x, y, w, h
. . .

// fillArc()
g.fillArc(x, y, rectWidth, rectHeight, 90, 135); // x, y, w, h
. . .

// fillPolygon()
Polygon filledPolygon = new Polygon();
filledPolygon.addPoint(x, y);
filledPolygon.addPoint(x+rectWidth, y+rectHeight);
filledPolygon.addPoint(x, y+rectHeight);
filledPolygon.addPoint(x+rectWidth, y);
//filledPolygon.addPoint(x, y);
g.fillPolygon(filledPolygon);

```

Trabajar con Texto

El soporte para el trabajo con texto primitivo se encuentra en las clases [Graphics](#), [Font](#), y [FontMetrics](#) del AWT.

Dibujar Texto

Cuando escribas código para dibujar texto, lo primero que deberías considerar es si puede utilizar un Componente orientado a texto como una clase `Label`, `TextField` o `TextArea`. Si no hay ningún componente apropiado puedes utilizar los métodos `drawBytes()`, `drawChars()`, o `drawString()` de la clase `Graphics`.

Aquí tienes un ejemplo de código que dibuja una cadena en la pantalla:

```
g.drawString("Hello World!", x, y);
```

Para los métodos de dibujo, `x` e `y` son enteros que especifican la posición de esquina inferior izquierda del texto. Para ser más precisos, la coordenada `y` especifica la línea base del texto -- la línea en la que descansan la mayoría de las letras -- lo que no incluye espacio para los tallos (descendentes) de letras como la "y". Asíguérate de hacer `y` lo suficientemente grande para permitir el espacio vertical para el texto, pero lo suficientemente pequeño para asignar espacio para los descendentes.

Aquí tienes una figura que muestra la línea base, así como las líneas ascendente y descendente. Aprenderás más sobre los ascendentes y los descendentes un poco más adelante.



Aquí tienes un applet sencillo que ilustra lo que sucede cuando usted no tiene cuidado con la posición del texto:

La cadena superior probablemente estará cortada, ya que su argumento `y` es 5, lo que deja sólo 5 pixels sobre la línea base para la cadena -- lo que no es suficiente para la mayoría de las fuentes. La cadena central probablemente se verá perfecta, a menos que tengas una fuente enorme por defecto. La mayoría de las letras de la cadena inferior se mostrarán bien, excepto la letras con descendentes. Todos los descendentes de la cadena inferior estarán cortados ya que el código que muestra esta

cadena no deja espacio para ellos. (Aquí tienes el [código fuente](#) del applet.)

Nota: la interpretación que los métodos de dibujo de texto hacen de x e y es diferente de la que hacen los métodos de dibujo de formas. Cuando se dibuja una forma (un rectángulo, por ejemplo) x e y especifican la esquina superior izquierda del rectángulo, en lugar de la esquina inferior izquierda.

Obtener información sobre la Fuente: FontMetrics

El ejemplo de dibujo de formas de la [página anterior](#) podría mejorarse eligiendo una fuente más pequeña que la fuente por defecto normal. El siguiente ejemplo hace esto y también agranda las formas para ocupar el espacio liberado por la fuente más pequeña. Abajo tienes el applet mejorado (aquí tienes el [código fuente](#)):

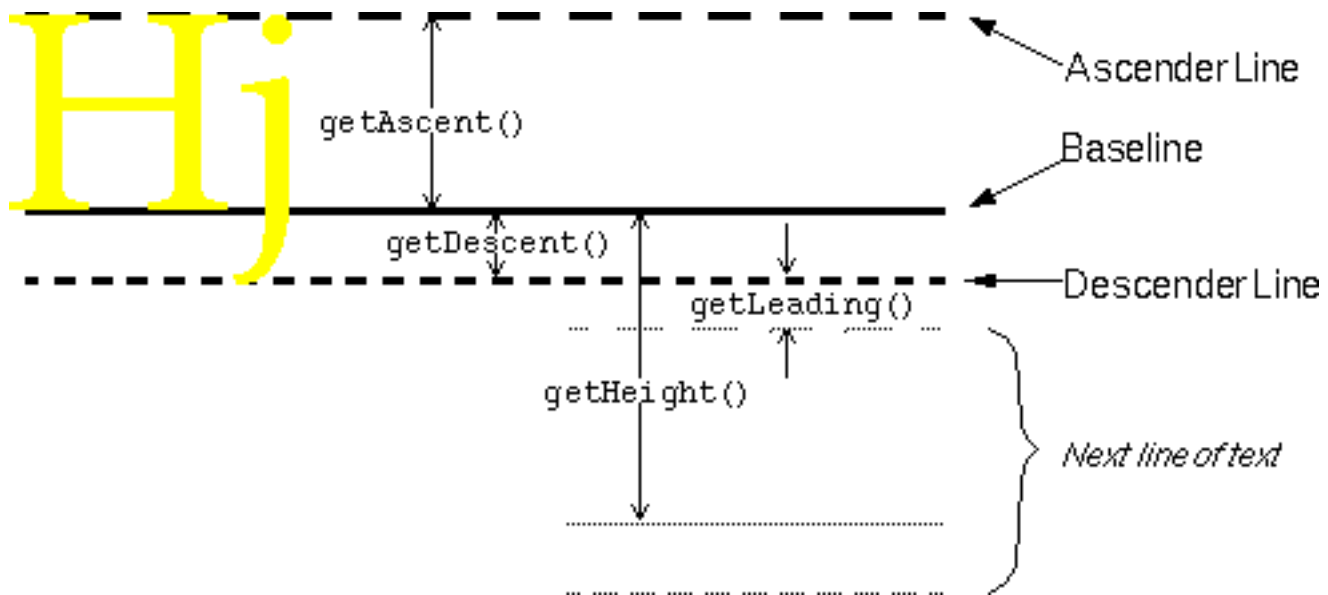
El ejemplo elige la fuente apropiada utilizando un objeto FontMetrics para obtener detalles del tamaño de la fuente. Por ejemplo, el siguiente bucle (en el método paint()) se asegura que la cadena más larga mostrada por el applet ("drawRoundRect()") entra dentro del espacio asignado a cada forma.

```
boolean textFits = false;
Font font = g.getFont();
FontMetrics fontMetrics = g.getFontMetrics();
while (!textFits) {
    if ((fontMetrics.getHeight() <= maxCharHeight)
        && (fontMetrics.stringWidth("drawRoundRect()")
            <= gridWidth)) {
        textFits = true;
    } else {
        g.setFont(font = new Font(font.getName(),
                                   font.getStyle(),
                                   font.getSize() - 1));
        fontMetrics = g.getFontMetrics();
    }
}
```

El ejemplo de código anterior utiliza los métodos getFont(), setFont(), y getFontMetrics() de la clase Graphics para obtener y seleccionar la fuente actual y para obtener el objeto FontMetrics que corresponde con el font. Desde los métodos getHeight() y getStringWidth() de FontMetrics, el código obtiene la información sobre el tamaño vertical y horizontal de la fuente.

La siguiente figura muestra parte de la información que un objeto

FontMetrics puede proporcionar sobre el tamaño de una fuente.



Aquí tienes un sumario de los métodos de FontMetrics que devuelven información sobre el tamaño vertical de la fuente:

`getAscent()`, `getMaxAscent()`

El método `getAscent()` devuelve el número de pixels entre la línea de ascendentes y la línea base. Generalmente, la línea de ascendentes representa la altura típica de una letra mayúscula. Específicamente, los valores ascendente y descendente los elige el diseñador de la fuente para representar el "color" correcto del texto, o la densidad de la tinta, para que el texto aparezca como lo planeó el diseñador. El ascendente típico proporciona suficiente espacio para casi todos los caracteres de la fuente, excepto quizás para los acentos de las letras mayúsculas. El método `getMaxAscent()` tiene en cuenta estos caracteres excepcionalmente altos.

`getDescent()`, `getMaxDescent()`

El método `getDescent()` devuelve el número de pixels entre la línea base y la línea de descendentes. En la mayoría de las fuentes, todos los caracteres caen en la línea descendente en su punto más bajo. Sólo en algunos casos, podrá utilizar el método `getMaxDescent()` para obtener una distancia garantizada para todos los caracteres.

`getHeight()`

Obtiene el número de pixels que se encuentran normalmente entre la línea base de una línea de texto y la línea base de la siguiente línea de texto. Observa que esto incluye el espacio para los ascendentes y descendentes.

`getLeading()`

Devuelve la distancia sugerida (en pixels) entre una línea de texto y la siguiente. Específicamente, esta es la distancia entre la línea descendente de una línea de texto y la línea ascendente de la siguiente.

Observa que el tamaño de la fuente (devuelto por el método `getSize()` de la clase `Font`) es una media abstracta. Teóricamente, corresponde al ascendente más el descendente. Sin embargo, en la práctica, el diseñador decide la altura que debe tener una fuente de "12 puntos". Por ejemplo, Times de 12-puntos es ligeramente más baja que Helvetica de 12-puntos. Típicamente, las fuentes se miden en puntos, que es aproximadamente 1/72 de pulgada.

La siguiente lista muestra los métodos que proporciona `FontMetrics` para devolver información sobre el tamaño horizontal de una fuente. Estos métodos tienen en cuenta el espacio entre los caracteres. Más precisamente, cada método no devuelve el número de pixels de un carácter particular (o caracteres), sino el número de pixels que avanzará la posición actual cuando se muestre el carácter (o caracteres). Llamamos a esto anchura de avance para distinguirla de la anchura del texto.

`getMaxAdvance()`

La anchura de avance (en pixels) del carácter más ancho de la fuente.

`bytesWidth(byte[], int, int)`

La anchura de avance del texto representado por el array de bytes especificado. El primer argumento entero especifica el origen de los datos dentro del array. El segundo argumento entero especifica el número máximo de bytes a chequear.

`charWidth(int), charWidth(char)`

La anchura de avance del carácter especificado.

`charsWidth(char[], int, int)`

La anchura de avance de la cadena representada por el array de caracteres especificado.

`stringWidth(String)`

La anchura de avance de la cadena especificada.

`getWidths()`

La anchura de avance de cada uno de los primeros 256 caracteres de la fuente.

Utilizar Imágenes

Las siguientes páginas proporcionan lo detalles necesarios para trabajar con imágenes. Aprenderás cómo cargarlas, mostrarlas y manipularlas.

El soporte para la utilización de imágenes está situado en los paquetes `java.applet`, `java.awt` y `java.awt.image`. Cada imagen está representada por un objeto `java.awt.image`. Además de la clase `Image`, el paquete `java.awt` proporciona otro soporte básico para imágenes, como el método `drawImage()` de la clase `Graphics`, el método `getImage()` de la clase `Toolkit` y la clase `MediaTracker`. En el paquete `java.applet`, el método `getImage()` de la clase `Applet` hace que los applet carguen imágenes de forma sencilla, utilizando URLs. Finalmente el paquete `java.awt.image` proporciona interfaces y clases que permiten crear, manipular y observar imágenes.

Cargar Imágenes

El AWT hace sencilla la carga de imágenes en estos dos formatos: GIF y JPEG. Las clases `Applet` y `Toolkit` proporcionan los métodos `getImage()` que trabajan con ambos formatos. Puedes utilizarlas de esta forma:

```
myImage = getImage(URL); //Sólo en métodos de una subclase de Applet
o
myImage = Toolkit.getDefaultToolkit().getImage(filenameOrURL);
```

Los métodos `getImage()` vuelven inmediatamente, por lo que no se tiene que esperar a que se cargue una imagen antes de ir a realizar otras operaciones en el programa. Mientras esto aumenta el rendimiento, algunos programas requieren más control sobre la imagen que se están cargando. Se puede controlar el estado de la carga de una imagen utilizando la clase `MediaTracker` o implementando el método `imageUpdate()`, que está definido por el interface `ImageObserver`.

Esta sección también explicará cómo crear imágenes al vuelo, utilizando la clase `MemoryImageSource`.

Mostrar Imágenes

Es sencillo dibujar una imagen utilizando el objeto `Graphics` que se pasó a sus métodos `update()` o `paint()`. Simplemente se llama al método `drawImage()` del objeto `Graphics`. Por ejemplo:

```
g.drawImage(myImage, 0, 0, this);
```

Esta sección explica las cuatro formas de `drawImage()`, dos de la cuales escalan la imagen. Al igual que `getImage()`, `drawImage()` es asíncrona, vuelve inmediatamente incluso si la imagen no se ha cargado o dibujado completamente todavía.

Manipular Imágenes

Esta sección ofrece una introducción sobre cómo cambiar imágenes, utilizando filtros. (El escalado de imágenes se cubre en [Mostrar Imágenes.](#))

Ozito

Cargar Imágenes

Esta página describe cómo obtener el objeto Image correspondiente a una imagen. Siempre que la imagen este en formato GIF o JPEG y se conozca su nombre de fichero o su URL, es sencillo obtener un objeto Image para ella: con solo utilizar uno de los métodos `getImage()` de Applet o Toolkit. Los métodos `getImage()` vuelven inmediatamente, sin comprobar si existen los datos de la imagen. Normalmente la carga real de la imagen no empieza hasta que el programa intenta dibujarla por primera vez.

Para muchos programas, esta carga en segundo plano funciona bien. Otros, sin embargo, necesitan seguir el proceso de carga de la imagen. Esta página explica cómo hacerlo utilizando la clase `MediaTracker` y el interface `ImageObserver`.

Finalmente, esta página contará cómo crear imágenes al vuelo, utilizando una clase como `MemoryImageSource`.

Utilizar los Métodos `getImage()`

Esta sección explica primero los métodos `getImage()` de la clase `Applet` y luego los de la clase `Toolkit`.

La clase `Applet` suministra dos métodos `getImage()`:

- `public Image getImage(URL url)`
- `public Image getImage(URL url, String name)`

Sólo los applets pueden utilizar los métodos `getImage()` de la clase `Applet`. Además, los métodos `getImage()` de `Applet` no trabajan hasta que el applet tenga un contexto completo (`AppletContext`). Por esta razón, estos métodos no trabajan si se llaman desde un constructor o en una sentencia que declara una variable de ejemplar. En su lugar, debería llamar a `getImage()` desde un método como `init()`.

El siguiente ejemplo de código muestra cómo utilizar los métodos `getImage()` de la clase `Applet`. Puede ver [Crear un GUI](#) para una explicación de los métodos `getbBase()` y `getDocumentBase()`.

```
//en un método de una subclase de Applet:  
Image image1 = getImage(getCodeBase(), "imageFile.gif");  
Image image2 = getImage(getDocumentBase(), "anImageFile.jpeg");  
Image image3 = getImage(new URL("http://java.sun.com/graphics/people.gif"));
```

La clase `Toolkit` declara dos métodos `getImage()` más:

- `public abstract Image getImage(URL url)`
- `public abstract Image getImage(String filename)`

Se puede obtener un objeto `Toolkit` llamando al método `getDefaultToolkit()` por defecto de la clase o llamando al método `getToolkit()` de la clase `Component`. Este último devuelve el `Toolkit` que fue utilizado (o que será utilizado) para implementar el `Componente`.

Aquí tienes un ejemplo de la utilización de los métodos `getImage()` de `Toolkit`. Todas las aplicaciones Java y los applets pueden utilizar estos métodos, en los applets están sujetos a las restricciones de seguridad usuales. Puedes leer sobre la seguridad de los applets en [Entender las Capacidades y las Restricciones de los Applets](#).

```
Toolkit toolkit = Toolkit.getDefaultToolkit();  
Image image1 = toolkit.getImage("imageFile.gif");  
Image image2 = toolkit.getImage(new URL("http://java.sun.com/graphics/people.gif"));
```

Petición y Seguimiento de la Carga de una Imagen: MediaTracker e ImageObserver

El AWT proporciona dos formas de seguir la carga de una imagen: la clase [MediaTracker](#) y el interface [ImageObserver](#).

La clase `MediaTracker` es suficiente para la mayoría de los programas. Se crea un ejemplar de `MediaTracker`, se le dice que haga un seguimiento a una o más imágenes, y luego se le pregunta por el estado de esas imágenes, cuando sea necesario. Puedes ver un ejemplo de esto en [Aumentar la Apariencia y el Rendimiento de una Animación de Imágenes](#).

El ejemplo de animación muestra dos características muy útiles de `MediaTracker`, petición para que sean cargados los datos de un grupo de imágenes, y espera a que sea cargado el grupo de imágenes. Para pedir que sean cargados los datos de un grupo de imágenes, se pueden utilizar las formas de `checkID()` y `checkAll()` que utilizan un argumento booleano. Seleccionando este argumento a `true` empieza la carga de los datos para todas aquellas imágenes que no hayan sido cargadas. O se puede pedir que los datos de la imagen sean cargados y esperen hasta su utilización utilizando los métodos `waitForID()` y `waitForAll()`.

El interface `ImageObserver` permite seguir más de cerca la carga de una imagen que `MediaTracker`. La clase `Component` lo utiliza para que sus componentes sean redibujados y las imágenes que muestran sean recargadas. Para utilizar `ImageObserver`, implemente el método `imageUpdate()` de este interface y asegúrese de que el objeto implementado sea registrado como el observador de imagen. Normalmente, este registro sucede cuando especifica un `ImageObserver` para el método `drawImage()`, como se describe en la siguiente página. El método `imageUpdate()` es llamado cuando la información sobre la imagen este disponible.

Aquí tienes un ejemplo de implementación del método `imageUpdate()` del interface `ImageObserver`. Este ejemplo utiliza `imageUpdate()` para posicionar dos imágenes tan pronto como se conozcan sus tamaños, y redibujarlas cada 100 milisegundos hasta que las dos imágenes estén cargadas (Aquí tienes el [programa completo](#).)

```
public boolean imageUpdate(Image theimg, int infoflags,
                           int x, int y, int w, int h) {
    if ((infoflags & (ERROR)) != 0) {
        errored = true;
    }
    if ((infoflags & (WIDTH | HEIGHT)) != 0) {
        positionImages();
    }
    boolean done = ((infoflags & (ERROR | FRAMEBITS | ALLBITS)) != 0);
    // Redibuja inmediatamente si lo hemos hecho si no vuelve a
    // pedir el redibujado cada 100 milisegundos
    repaint(done ? 0 : 100);
    return !done; //Si está hecho, no necesita más actualizaciones.
}
```

Si navegas por la documentación del API sobre `MediaTracker`, podrías haber observado que la clase `Component` define dos métodos de aspecto muy útil: `checkImage()` y `prepareImage()`. La clase `MediaTracker` ha hecho que estos métodos ya no sean necesarios.

Crear Imágenes con `MemoryImageSource`

Con la ayuda de un productor de imágenes como la clase [MemoryImageSource](#), podrás construir imágenes a partir de la improvisación. El siguiente ejemplo calcula una imagen de 100x100 representando un degradado de colores del negro al azul a lo largo del eje X y un degradado del negro al rojo a lo largo del eje Y.

```
int w = 100;
int h = 100;
int[] pix = new int[w * h];
int index = 0;
```

```
for (int y = 0; y < h; y++) {  
    int red = (y * 255) / (h - 1);  
    for (int x = 0; x < w; x++) {  
        int blue = (x * 255) / (w - 1);  
        pix[index++] = (255 << 24) | (red << 16) | blue;  
    }  
}  
Image img = createImage(new MemoryImageSource(w, h, pix, 0, w));
```

Ozito

Mostrar Imágenes

Aquí tienes un ejemplo de código que muestra una image a su tamaño normal en la esquina superior izquierda del área del Componente (0,0):

```
g.drawImage(image, 0, 0, this);
```

Aquí tienes un ejemplo de código que muestra una imagen escalada para tener 300 pixels de ancho y 62 de alto, empezando en las coordenadas (90,0):

```
g.drawImage(myImage, 90, 0, 300, 62, this);
```

Abajo tienes un applet que muestra una imagen dos veces, utilizando los dos ejemplos de código anteriores. Aquí tienes el [código completo](#) del programa.

La clase Graphics declara los siguientes métodos drawImage(). Todos devuelven un valor booleano, aunque ese valor casi nunca se utiliza. El valor de retorno es true si la imagen se ha cargado completamente, y por lo tanto se ha dibujado completamente; de otra forma, es false.

- public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)
- public abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)
- public abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)
- public abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)

Los métodos drawImage() tienen los siguientes argumentos:

Image img

La imagen a dibujar.

int x, int y

Las coordenadas de la esquina superior izquierda de la imagen.

int width, int height

Al anchura y altura (en pixels) de la imagen.

Color bgcolor

El color de fondo de la imagen. Esto puede ser útil si la imagen contiene pixels transparentes y sabe que la imagen se va a mostrar sobre un fondo sólido del color indicado.

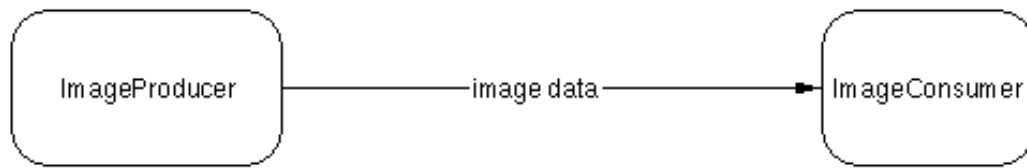
ImageObserver observer

Un objeto que implementa el interface ImageObserver. Esto registra el objeto como el observador de la imagen para que sea notificado siempre que esté disponible nueva información sobre la imagen. La mayoría de los componentes pueden especificar simplemente this.

La razón por la que this funciona como el observador de la imagen es que la clase Component implementa el interface ImageObsever. Esta implementación llama al método repaint() cuando se han cargado los datos de la imagen, que normalmente es lo que se quiere que suceda.

Los métodos drawImage() vuelven después de mostrar los datos de la imagen que ha sido cargada. Si quieres asegurarte de que drawImage() sólo dibuja imágenes completas, debes seguir la carga de la imagen. Puedes ver la [página anterior](#) para información sobre el seguimiento de la carga de una imagen.

Manipular Imágenes



La figura anterior muestra como se crean los datos de una imagen detrás de la escena. Un productor de imagen -- un objeto que implementa el interface [ImageProducer](#) -- produce una columna de datos para un objeto Image. El productor de imagen proporciona estos datos al consumidor de imagen -- un objeto que implementa el interface [ImageConsumer](#). A menos que se necesite manipular o crear imágenes personalizadas, no necesitarás saber como trabajan el productor y el consumidor de imágenes. El AWT utiliza automáticamente productores y consumidores de imágenes detrás de la escena.

El AWT soporta la manipulación de imágenes permitiendo insertar filtros de imagen entre el productor y el consumidor. Un filtro de imagen es un objeto [ImageFilter](#) que se sitúa entre el productor y el consumidor, modificando los datos de la imagen antes de que los obtenga el consumidor. ImageFilter implementa el interface ImageConsumer, ya que intercepta los mensajes que el productor envía al consumidor. La siguiente figura muestra cómo se sitúa un filtro de imagen entre el productor y el consumidor de imágenes.



Cómo utilizar un Filtro de Imagen

Utilizar un filtro de imagen existente es sencillo. Sólo tienes que utilizar el siguiente código, modificando el constructor del filtro de imagen si es necesario.

```
Image sourceImage;  
...//Inicializa sourceImage, utilizando el método getImage() de Toolkit o de Applet.  
ImageFilter filter = new SomeImageFilter();  
ImageProducer producer = new FilteredImageSource(sourceImage.getSource(), filter);  
Image resultImage = createImage(producer);
```

La página siguiente explica cómo trabaja el código anterior y te dice donde puedes encontrar algunos filtros de imagen.

Cómo escribir un Filtro de Imagen

¿Y si no encuentras un filtro de imagen que haga lo que necesitas? Puedes escribir tu propio filtro de imagen. Esta página ofrece algunos trucos sobre cómo hacerlo, incluyen enlaces a ejemplos y una explicación de un filtro personalizado que rota imágenes.

Cómo Utilizar un Filtro de Imagen

El siguiente applet utiliza un filtro para rotar una imagen. El filtro es uno personalizado llamado RotateFilter que podrás ver explicado en la página siguiente. Todo lo que necesitas saber sobre el filtro para utilizarlo es que su constructor toma un sólo argumento double: el ángulo de rotación en radianes. El applet convierte el número que introduce el usuario de grados a radianes, para que el applet pueda construir un RotateFilter.

Abajo tienes el código fuente que utiliza el filtro. (Aquí tienes el [programa completo](#).)

```
public class ImageRotator extends Applet {
    . . .
    RotatorCanvas rotator;
    double radiansPerDegree = Math.PI / 180;

    public void init() {
        //Carga la imagen.
        Image image = getImage(getCodeBase(), "../images/rocketship.gif");

        ...//Crea el componente que utiliza el filtro de imagen:
        rotator = new RotatorCanvas(image);
        . . .
        add(rotator);
        . . .
    }

    public boolean action(Event evt, Object arg) {
        int degrees;

        ...//obtiene el número de grados que se tiene que rotar la imagen.

        //Lo convierte a Radianes.
        rotator.rotateImage((double)degrees * radiansPerDegree);

        return true;
    }
}

class RotatorCanvas extends Canvas {
    Image sourceImage;
    Image resultImage;

    public RotatorCanvas(Image image) {
        sourceImage = image;
        resultImage = sourceImage;
    }
}
```

```

public void rotateImage(double angle) {
    ImageFilter filter = new RotateFilter(angle);
    ImageProducer producer = new FilteredImageSource(
        sourceImage.getSource(),
        filter);

    resultImage = createImage(producer);
    repaint();
}

public void paint(Graphics g) {
    Dimension d = size();
    int x = (d.width - resultImage.getWidth(this)) / 2;
    int y = (d.height - resultImage.getHeight(this)) / 2;

    g.drawImage(resultImage, x, y, this);
}
}

```

Cómo trabaja el Código

Para utilizar un filtro de imagen, un programa debe seguir los siguientes pasos:

1. Obtener un objeto Image (normalmente se hace con el método `getImage()`).
2. Utilizando el método `getSource()`, obtiene la fuente de los datos (un `ImageProducer`) para el objeto Image.
3. Crea un ejemplar del filtro de imagen, inicializando el filtro si es necesario.
4. Crea un objeto `FilteredImageSource`, pasando al constructor la fuente de la imagen y el objeto del filtro.
5. Con el método `createImage()` del componente, crea un nuevo objeto Image que tiene el `FilteredImageSource` como su productor de imagen.

Esto podría sonar complejo, pero realmente es sencillo de implementar. Lo realmente complejo está detrás de la escena, como explicaremos un poco más tarde. Primero explicaremos el código del applet que utiliza el filtro de imagen.

En el applet de ejemplo, el método `rotateImage()` de `RotatorCanvas` realiza la mayoría de las tareas asociadas con el uso del filtro de imagen. La única excepción es el primer paso, obtener el objeto Image original, que es realizado por el método `init()` del applet. Este objeto Image es pasado a `RotatoCanvas`, que se refiere a él como `sourceImage`.

El método `rotateImage()` ejemplariza el filtro de imagen llamando al constructor del filtro. El único argumento del constructor es el ángulo, en radianes, que se va a rotar la imagen.

```
ImageFilter filter = new RotateFilter(angle);
```

Luego, el método `rotateImage()` crea un ejemplar de `FilteredImageSource`. El primer argumento del constructor de `FilteredImageSource` es la fuente de la imagen, obtenida con el método `getSource()`. El segundo argumento es el objeto filtro.

```
ImageProducer producer = new FilteredImageSource(  
    sourceImage.getSource(),  
    filter);
```

Finalmente, el código crea una segunda Imagen, `resultImage`, llamando al método `createImage()` de la clase `Component`. El único argumento de `createImage()` es el objeto `FilteredImageSource` creado en el paso anterior.

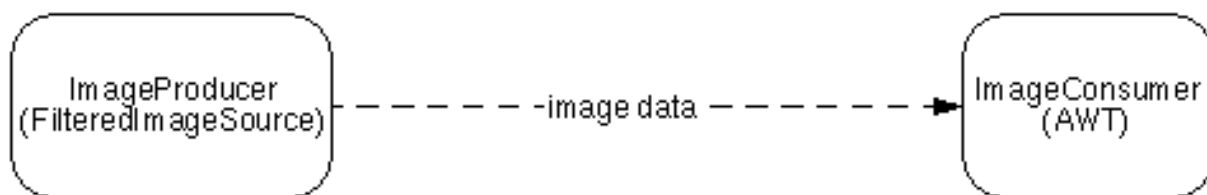
```
resultImage = createImage(producer);
```

Qué sucede detrás de la escena

Esta sección explica cómo trabaja el filtrado de la imagen, detrás de la escena. Si no te interesan estos detalles de la implementación, puedes saltar a [Donde Encontrar Filtros de Imagen](#).

Lo primer que necesitas saber es que el AWT utiliza `ImageConsumer` detrás de la escena, en respuesta a una petición a `drawImage()`. Por eso el Componente que muestra la imagen no es el consumidor de imagen -- alguno objeto profundo del AWT es el consumidor de imagen.

La llamada anterior a `createImage()` selecciona una Imagen (`resultImage`) que espera obtener los datos desde su productor, el ejemplar de `FilteredImageSource`. Aquí tienes lo que parecería el path de los datos de la imagen, desde la perspectiva de `resultImage()`:



La línea punteada indica que el consumidor de imagen realmente nunca obtiene los datos del `FilteredImageSource`. En su lugar, cuando el consumidor pide datos de la imagen (en respuesta a `g.drawImage(resultImage,...)`), el `FilteredImageSource` realiza algún escamoteo y luego lo disocia de alguna manera. Aquí tienes la magia realizada por `FilteredImageSource`:

- Crea un nuevo objeto del filtro de imagen invocando al método `getFilterInstance()` en el objeto filtro que se le ha pasado al constructor de `FilteredImageSource`. Por defecto, `getFilterInstance()` clona el objeto filtro.
- Conecta el nuevo filtro de imagen al consumidor de imagen.
- Conecta la fuente de los datos de la imagen, que se ha pasado al constructor de `FilteredImageSource`, al filtro de imagen.

Aquí tienes el resultado:



Dónde Encontrar Filtros de Imagen

Entonces, ¿dónde puedes encontrar filtros de imagen existentes? El paquete `java.awt.image` incluye un filtro listo para utilizar, [CropImageFilter](#), que produce una imagen que consiste en un región rectangular de la imagen original. También puede encontrar varios filtros de imagen utilizados por applets en la website `se.sun`. Todas estas páginas incluyen enlaces al código fuente de los applets que utilizan un filtro de imagen:

- La página [Generación Dinámica de Etiquetas de Color](#) contiene dos applets que modifican el color de una imagen. El primer applet, `AlphaBulet`, define y utiliza un `AlphaColorFilter`; el segundo `HueBulet`, define y utiliza `HueFilter`.
- La página [Realimentación en directo de Imagemap](#) demuestra un applet que mapea una imagen. Utiliza muchos filtros para proporcionar realimentación visual cuando el cursor se mueve sobre ciertas áreas o cuando el usuario pulsa un área especial.
- La página [Prueba de Imagen](#) realiza una variedad de procesos de imágenes. Además de permitir al usuario escalar o mover la imagen, define y utiliza tres filtros. El `AlphaFilter` hace la imagen transparente, el `RedBlueSwapFilter` cambia los colores de la imagen y el `RotateFilter` rota la imagen, como has podido ver en esta sección.

Cómo Escribir un Filtro de Imagen

Todos los filtros de imagen deben ser subclasses de la clase [ImageFilter](#). Si un filtro de imagen va a modificar los colores o la transparencia de una imagen, en vez de crear directamente una subclase de [ImageFilter](#), probablemente deberías crear una subclase de [RGBImageFilter](#).

Antes de escribir un filtro de imagen, deberías encontrar otros estudiando los que son similares al que planeas escribir. También deberás estudiar los interfaces [ImageProducer](#) e [ImageConsumer](#), para familiarizarte con ellos.

Encontrar ejemplos

Podrás encontrar ejemplos de subclasses de [RGBImageFilter](#) en los applets de las páginas mencionadas en la página anterior

Más adelante en esta página veras un ejemplo de una subclase directa de [ImageFilter](#), [RotateFilter](#).

Crear una subclase de ImageFilter

Como se mencionó antes, los filtros de imagen implementan el interface [ImageConsumer](#). Esto permite interceptar los datos destinados al consumidor de imagen. [ImageConsumer](#) define los siguientes métodos:

```
void setDimensions(int width, int height);
void setProperties(Hashtable props);
void setColorModel(ColorModel model);
void setHints(int hintflags);
void setPixels(int x, int y, int w, int h, ColorModel model, byte pixels[], int off,
int scansize);
void setPixels(int x, int y, int w, int h, ColorModel model, int pixels[], int off,
int scansize);
void imageComplete(int status);
```

La clase [ImageFilter](#) implementa todos los métodos anteriores para que reenvíen los datos del método al consumidor del filtro. Por ejemplo, [ImageFilter](#) implementa el método `setDimensions()` de la siguiente forma:

```
public void setDimensions(int width, int height) {
    consumer.setDimensions(width, height);
}
```

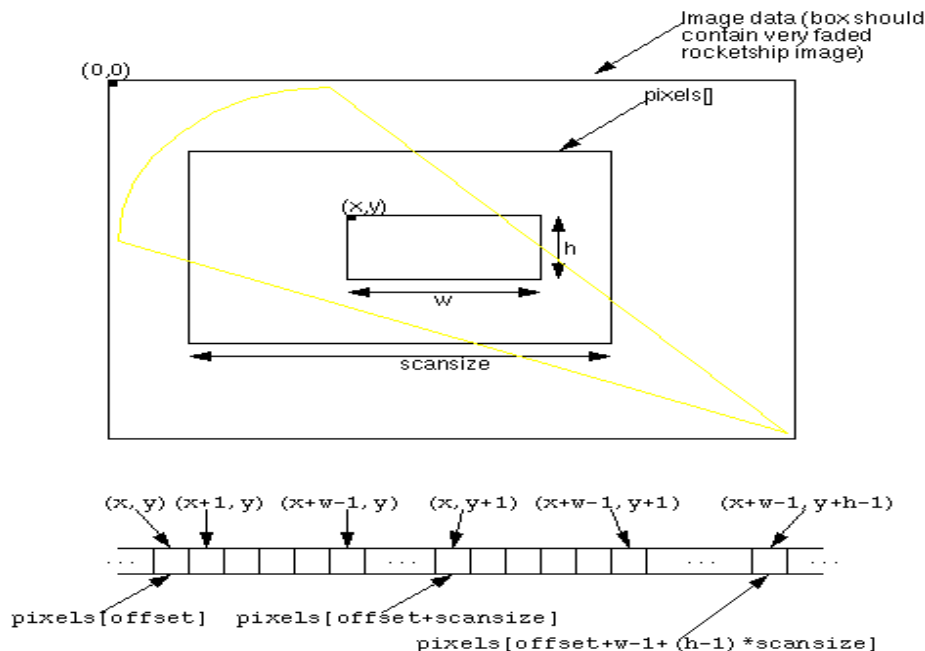
Gracias a estos métodos de [ImageFilter](#), tu subclase no necesitará implementar todos los métodos de [ImageConsumer](#). Sólo necesitará implementar los métodos que transmitan los datos que quieres cambiar.

Por ejemplo, la clase [CropImageFilter](#) implementa cuatro métodos de [ImageConsumer](#): `setDimensions()`, `setProperties()`, y dos variedades de `setPixels()`. También implementa un constructor con argumentos que especifica el rectángulo a recortar. Como otro ejemplo, la clase [RGBImageFilter](#) implementa algunos métodos de ayuda, define un método de ayuda abstracto que realiza las modificaciones reales del color de cada pixel, e implementa los siguientes métodos de [ImageConsumer](#): `setColorModel()` y dos variedades de `setPixels()`.

La mayoría, si no todos, los filtros implementan métodos `setPixels()`. Estos métodos determinan exactamente qué datos de la imagen se van a utilizar para construir la imagen. Uno o los dos métodos `setPixels()` podrían ser llamados varias veces durante la construcción de una sola imagen. Cada llamada le da información al [ImageConsumer](#) sobre un rectángulo de pixels dentro de la imagen. Cuando se llama al método `imageComplete()` del [ImageConsumer](#) con cualquier estado excepto `SINGLEFRAMEDONE` (lo que implica que aparecerán los datos para más marcos), entonces el [ImageConsumer](#) puede asumir que no va a recibir más llamadas de `setPixels()`. Un `imageComplete()` con estado de

STATICIMAGEDONE especifica no sólo que se han recibido los datos completos de la imagen, sino que además no se ha detectado ningún error.

La siguiente ilustración y la tabla describen los argumentos de los métodos setPixels().



x, y
Especifica la posición dentro de la imagen, relativa a su esquina superior izquierda, en la que empieza el rectángulo.

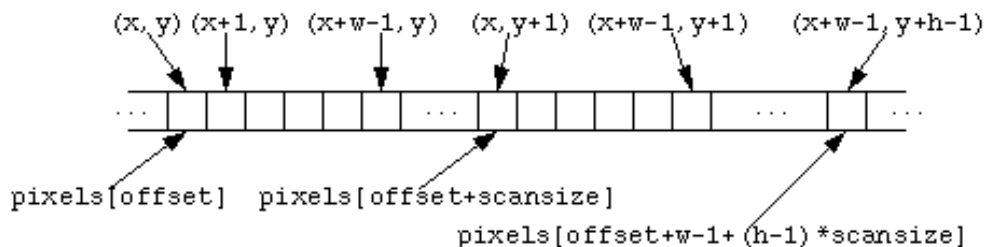
w, h
Especifica la anchura y altura, en pixels, de este rectángulo.

model
Especifica el modelo de color utilizado por los datos en el array de pixels.

pixels[]
Especifica un array de pixels. El rectángulo de los datos de la imagen está contenido en este array, pero el array debería contener más de $w \cdot h$ entradas, dependiendo de los valores de offset y scansize. Aquí tienes la fórmula para determina que entrada en el array pixels contiene los datos del pixel situado en $(x+i, y+j)$, donde $(0 \leq i < w)$ y $(0 \leq j < h)$:

$$\text{offset} + (j * \text{scansize}) + i$$

La fórmula anterior asume que (m,n) está en el rectángulo especificado por esta llamada a setPixels(), y que (m,n) es relativo al origen de la imagen. Abajo tienes una ilustración del array pixels para aclararlo. Muestra cómo un pixels específico (por ejemplo (x,y)) obtiene su entrada en el array.



offset
Especifica el índice (en el array pixels) del primer pixel del rectángulo.

scansize
Especifica la anchura de cada fila en el array pixels. Debido a consideraciones de eficiencia, este podría ser más grande que w.

El Filtro de Imagen RotateFilter

La clase [RotateFilter](#) rota una imagen el ángulo especificado. Se basa en las siguientes fórmulas gráficas para calcular la nueva posición de cada pixel:

```
newX = oldX*cos(angle) - oldY*sin(angle)
newY = oldX*sin(angle) + oldY*cos(angle)
```

RotateFilter implementa los siguientes métodos de ImageConsumer:

`setDimensions()`

Graba la anchura y altura de la imagen sin filtrar para utilizarlas en los métodos `setPixels()` y `imageComplete()`. Calcula la anchura y altura finales de la imagen filtrada, grabándolas para utilizarlas con el método `imageComplete()`, crea un buffer para almacenar los datos que entran de la imagen, y llama al método `setDimensions()` del consumidor para seleccionar la anchura y altura nuevas.

`setColorModel()`

Le dice al consumidor que espere los pixels en el modelo del color RGB, por defecto.

`setHints()`

Le dice al consumidor que espere los datos de la imagen en orden desde arriba a abajo y de izquierda a derecha (el orden en que estás leyendo esta página), en pasos de líneas completas, y que cada pixel es enviado exactamente una vez.

`setPixels()` (las dos variedades de este método)

Convierten los pixels al modelo RBG por defecto (si es necesario) y copia el pixel en buffer de almacenamiento. La mayoría de los filtros de imagen simplemente modifican el pixel y lo envían al consumidor, pero como el lado de un rectángulo girado ya no es horizontal o vertical (para la mayoría de los ángulos), este filtro puede que no envíe los pixels de forma eficiente desde su método `setPixels()`. En su lugar, `RotateFilter` almacena todos los datos de los pixels hasta que recibe un mensaje `imageComplete()`.

`imageComplete()`

Rota la imagen y luego llamada repetidamente a `consumer.setPixels()` para enviar cada línea de la imagen al consumidor. Después de enviar toda la imagen, este método llama a `consumer.imageComplete()`.

Realizar Animaciones

Lo que todas las formas de animación tienen en común es que todas ellas crean alguna clase de percepción de movimiento, mostrando marcos sucesivos a una velocidad relativamente alta. La animación por ordenador normalmente muestra 10-20 marcos por segundo. En comparación, la animación de dibujos manuales utiliza desde 8 marcos por segundo (para una animación de poca calidad) hasta 24 marcos por segundo (para movimiento realista) pasando por 12 marcos por segundo (de la animación estándar). Las siguientes páginas cuentan todo lo que se necesita saber para escribir un programa Java que realice una animación.

Antes de empezar: Comprueba las herramientas de animación existentes y los applet como [Animator](#), para ver si puedes utilizar uno de ellos en vez de escribir su propio programa.

Crear el Bucle de Animación

El paso más importante para crear un programa de animación es seleccionar correctamente el marco de trabajo. Excepto para la animación realizada sólo en respuesta directa a eventos externos (como un objeto arrastrado por el usuario a través de la pantalla), un programa que realiza una animación necesita un bucle de animación.

El bucle de animación es el responsable de seguir la pista del marco actual, y de la petición periódica de actualizaciones de la pantalla. Para los applets y muchas aplicaciones necesitará un thread separado para ejecutar el bucle de animación. Esta sección contiene un applet de ejemplo y una aplicación que se pueden utilizar como plantillas para todas tus animaciones.

Generar Gráficos

Esta sección genera un ejemplo que anima gráficos primitivos.

Eliminar el Parpadeo

El ejemplo desarrollado en la sección anterior no es perfecto, porque parpadea. Esta sección enseña cómo utilizar dos técnicas para eliminar el parpadeo:

- Sobreescribir el método `update()`
- Doble buffer (también conocido como utilizar un buffer de vuelta)

Mover una Imagen a través de la Pantalla

Esta simple forma de animación envuelve el movimiento de una imagen estable a través de la pantalla. En el mundo de la animación tradicional esto es conocido como animación recortable, ya que generalmente se conseguía cortando una forma en papel y moviendo la forma delante de la cámara. En programas de ordenadores, esta técnica se utiliza frecuentemente en interfaces del tipo drag & drop (arrastrar y soltar).

Mostrar una Secuencia de Imágenes

Esta sección enseña cómo realizar una animación clásica, al estilo de los dibujos animados, dando una secuencia de imágenes.

Aumentar la Apariencia y el Rendimiento de una Animación

Esta sección enseña cómo utilizar la clase MediaTracker para que se pueda retardar la animación hasta que las imágenes se hayan cargado. También encontrarás varios trucos para aumentar el rendimiento de una animación de una applet combinando los ficheros de imágenes utilizando un esquema de compresión como Flic.

Crear un Bucle de Animación

Todo programa que realice animaciones dibujando a intervalos regulares necesita un bucle de animación. Generalmente, este bucle debería estar en su propio thread. Nunca debería estar en los métodos `paint()` o `update()`, ya que ahí se encuentra el thread principal del AWT, que se encarga de todo el dibujo y manejo de eventos.

Esta página proporciona dos plantillas para realizar animación, una para applets y otra para aplicaciones. La versión de applets se está ejecutando justo debajo. Puedes pulsar sobre ella para parar la animación y pulsar de nuevo para arrancarla.

La animación que realiza la plantilla es un poco aburrida: sólo muestra el número de marco actual, utilizando un ratio por defecto de 10 marcos por segundo. Las siguientes páginas construyen este ejemplo, mostrándote cómo animar gráficos primitivos e imágenes.

Aquí tienes el código para la [plantilla de animación para applets](#). Aquí tienes el código equivalente para la [plantilla de animación para aplicaciones](#). El resto de esta página explica el código de la plantilla, Aquí tienes un sumario de lo que hacen las dos plantillas:

```
public class AnimatorClass extends AComponentClass implements Runnable {

    //En el código de inicialización:
        //El valor de marcos por segundos especificado por el usuario
determina
    //el retardo entre marcos.

    //En un método que no hace nada salvo bpezar la animación:
    //Crea y arranca el thread de la animación.

    //En un método que no hace nada salvo parar la animación:
    //Parar el Thread de la animación.

    public boolean mouseDown(Event e, int x, int y) {
        if (/* la animación está parada actualmente */) {
            //Llamar al método que arranca la animación.
        } else {
            //LLamar al método que para la animación.
        }
    }

    public void run() {
        //Bajar la prioridad de este thread para que no interfiera
        //con otros procesos.

        //Recuerde el momento de arranque.

        //Aquí tiene el bucle de animación:
        while (/* el thread de animación se está ejecutando todavía */) {
            //Avance un marco la animación.
            //Lo muestra.
            //Retardo dependiendo del numero de marcos por segundo.
```

```

    }
}

public void paint(Graphics g) {
    //Dibuja el marco actual de la animación.
}
}

```

Inicializar Variables de Ejemplar

Las plantillas de animación utilizan cuatro variables de ejemplar.

La primera variable (frameNumber) representa el marco actual. Es inicializada a -1, aunque el número del primer marco es 0. La razón, el número de marco es incrementado al bpezar el bucle de animación, antes de que se dibujen los marcos. Así, el primer marco a pintar es el 0.

La segunda variable de ejbplar (delay) es el número de milisegundos entre marcos. Se inicializa utilizando el número de marcos por segundo proporcionado por el usuario. Si el usuario proporciona un número no válido, las plantillas toman por defecto el valor de 10 marcos por segundo. El siguiente código convierte los marcos por segundo en el número de segundos entre marcos:

```
delay = (fps > 0) ? (1000 / fps) : 100;
```

La notación ? : del código anterior son una abreviatura de if else. Si el usuario proporciona un número de marcos mayor de 0, el retardo es 1000 milisegundos dividido por el número de marcos por segundo. De otra forma, el retardo entre marcos es 100 milisegundos.

La tercera variable de ejbplar (animatorThread) es un objeto Thread, que representa la thread en el que se va a ejecutar la animación. Si no estas familiarizado con los Threads, puedes ver la lección [Threads de Control](#).

La cuarta variable de ejbplar (frozen) es un valor booleano que está inicializado a false. La plantilla pone esa vairable a true para indicar que el usuario a pedido que termine la animación. Verás más sobre esto más adelante en esta sección.

El bucle de animación

El bucle de animación (el bucle while en el thread de la animación) hace lo siguiente una vez tras otra:

1. Avanza el número de marco.
2. Llama al método repaint() para pedir que se dibuje el número de marco actual de la animación.
3. Duerme durante delay milisegundos (más o menos).

Aquí tienes el código que realiza estas tareas:

```

while (/* El bucle de animación se está ejecutando todavía */) {
    //Avanza el marco de animación.
    frameNumber++;
}

```

```

//Lo muestra.
repaint();

...//Retardo dependiendo del número de marcos por segundo.
}

```

Asegurar un Ratio de Marcos por Segundos

La forma más obvia de implementar el tiempo de descanso del bucle de animación es dormir durante delay milisegundos. Esto, sin embargo, hace que el thread duerma demasiado, ya que ha perdido cierto tiempo mientras ejecuta el bucle de animación.

La solución de este problema es recordar cuando comenzó la animación, sumarle delay milisegundos para llegar al momento de levantarse, y dormir hasta que suene el despertador. Aquí tienes el código que implementa esto:

```

long startTime = System.currentTimeMillis();
while (/* El thread de animación se está ejecutando todavía */) {
    ...//Avanza el marco de la animación y lo muestra.
    try {
        startTime += delay;
        Thread.sleep(Math.max(0,
                               startTime-System.currentTimeMillis()));
    } catch (InterruptedException e) {
        break;
    }
}
}

```

Comportamiento Educado

Dos características más de estas plantillas de animación pertenecen a la categoría de comportamiento educado.

La primera característica es permitir explícitamente que el usuario pare (y arranque) la animación, mientras el applet o la aplicación sean visibles. La animación puede distraer bastante y es buena idea darle al usuario el poder de pararla para que pueda concentrarse en otra cosa. Esta característica está implementada sobrescribiendo el método `mouseDown()` para que pare o arranque el thread de la animación., dependiendo del estado actual del thread. Aquí tienes el código que implementa esto:

```

...//En el código de Inicialización:
boolean frozen = false;

...//En el método que arranca el thread de la animación:
if (frozen) {
    //No hacer nada. El usuario ha pedido que se pare la animación.
} else {
    //Iniciar la animación!
    ...//Crear y arrancar el thread de la animación.
}

```

```

    }
}

. . .

public boolean mouseDown(Event e, int x, int y) {
    if (frozen) {
        frozen = false;
        //Llama al método que arranca la animación.
    } else {
        frozen = true;
        //Llama al método que para la animación.
    }
    return true;
}

```

La segunda característica es suspender la animación siempre que el applet o la aplicación no sean visibles. Para la plantilla de animación de applet, esto se consigue implementando los métodos `stop()` y `start()` del applet. Para la plantilla de la aplicación, esto se consigue implementando un manejador de eventos para los eventos `WINDOW_ICONIFY` y `WINDOW_DEICONIFY`. En las dos plantillas, si el usuario congela la animación, cuando el programa detecta que la animación no es visible, le dice al thread de la animación que pare. Cuando el usuario revisita la animación, el programa reanuda el thread a menos que el usuario haya pedido que se pare la animación.

Podrías preguntarte por qué incrementar el número de marco al principio del turno en vez al final. La razón para hacer esto es lo que sucede cuando el usuario congela la aplicación, la deja y luego la revisita. Cuando el usuario congela la animación, el bucle de animación se completa antes de salir. Si el número de marco se incrementara al final del bucle, en lugar de al principio, el número de marco cuando el bucle sale sería uno más que el marco que se está mostrando. Cuando el usuario revisita la animación, la animación podría ser congelada en un marco diferente del que dejó el usuario. Esto podría ser desconcertante y, si el usuario para la animación en un marco particular, aburrido.

Animar Gráficos

Esta página genera un applet ejemplo que crea un efecto de movimiento, dibujando cuadrados alternativos. Los cuadrados son dibujados por el método `fillRect()` de `Graphics`. Aquí tienes el applet en acción:

Habrás observado que los gráficos no se animan perfectamente -- ya que ocasionalmente alguna parte del área de dibujo parpadea notablemente. La siguiente página explica la causa del parpadeo y le explica como eliminarlo.

Aquí tienes el [código del applet](#). La mayor diferencia entre este y la plantilla de animación es que el método `paint()` ha cambiado para dibujar rectángulos rellenos, usando un algoritmo que depende del número de marco actual. Este applet también introduce un par de variables de ejemplar, una que contiene el tamaño del cuadrado y otra que mantiene la pista de si la siguiente columna que será dibujada con un cuadrado negro. El usuario puede seleccionar el tamaño del cuadrado mediante un nuevo parámetro del applet.

Abajo tienes el código del método `paint()` que realiza el dibujo real. Observa que el programa sólo dibuja cuadrados negros (indicados porque `fillSquare` es `true`), no los otros cuadrados. Se puede eliminar esto porque, por defecto, el área de dibujo de un Componente se limpia (selecciona el color de fondo) justo antes de llamar al método `paint()`.

```
// Dibuja el rectángulo si es necesario.
if (fillSquare) {
    g.fillRect(x, y, w, h);
    fillSquare = false;
} else {
    fillSquare = true;
}
```


Eliminar el Parpadeo

El parpadeo que podrías haber observado en el ejemplo de la página anterior es un problema común con la animación (y ocasionalmente con los gráficos estáticos). El efecto de parpadeo es el resultado de dos factores:

- Por defecto, el fondo de la animación es limpiado (se redibuja su área completa con el color de fondo) antes de llamar al método `paint()`.
- El cálculo del método `paint()` del ejemplo anterior es tan largo que utiliza más tiempo en calcular y dibujar cada marco de la animación que el ratio de refresco de la pantalla. Como resultado, la primera parte del marco se dibuja en un pase de refresco de vídeo, y el resto del marco se dibuja en el siguiente (o incluso el siguiente al siguiente). El resultado es que aunque la primera parte del marco se anima normalmente (usualmente), puede ver una ruptura entre la primera y la segunda parte, ya que la segunda parte están en blanco hasta el segundo pase.

Se puede utilizar dos técnicas para eliminar el parpadeo: sobrescribir el método `update()` e implementar doble buffer.

Sobrescribir el método `update()`

Para eliminar el parpadeo, tanto si se utiliza como si no el doble buffer, debe sobrescribir el método `update()`. Esto es necesario, porque es la única forma para prevenir que fondo del componente sea limpiado cada vez que se dibuja el componente.

Implementar el Doble Buffer

Doble buffer implica realizar múltiples operaciones gráficas en un buffer gráfico que no está en la pantalla, y luego dibujar la imagen resultante en la pantalla. El doble buffer evita que las imágenes incompletas se dibujen en la pantalla.

Eliminar el Parpadeo: Sobreescribir el Método update()

Para eliminar el parpadeo, debe sobreescribir el método `update()`. La razón trata con la forma en que el AWT le pide a cada componente (como un Applet, un Canvas o un Frame) que se redibuje a sí mismo.

El AWT pide que se redibuje llamando al método `update()` del componente. La implementación por defecto de `update()` limpia el fondo del componente antes de llamar al método `paint()`. Cómo eliminar el parpadeo requiere que elimine todo el dibujo innecesario, su primer paso siempre es sobreescribir el método `update()` para que borre todo el fondo sólo cuando sea necesario. Cuando mueva el código de dibujo del método `paint()` al método `update()`, podría necesitar modificar el código de dibujo, para que no dependa de si el fondo ha sido borrado.

Nota: Incluso si su implementación de `update()` no llama a `paint()`, debe implementar este método. La razón: Cuando un área de un componente se revela de repente después de hacer estado oculta (detrás de alguna otra ventana, por ejemplo), el AWT llama directamente al método `paint()`, sin llamar a `update()`. Una forma sencilla de implementar el método `paint()` es hacer una llamada a `update()`.

Aquí tienes el [código](#) de una versión modificada del ejemplo anterior que implementa `update()` para eliminar el parpadeo. Aquí tienes el applet en acción:

Aquí tienes la nueva versión del método `paint()`, junto con el nuevo método `update()`. Todo el código de dibujo que era utilizado por el método `paint()` está ahora en el método `update()`. Los cambios significantes en el código de dibujo están en **negrita**.

```
public void paint(Graphics g) {
    update(g);
}

public void update(Graphics g) {
    Color bg = getBackground();
    Color fg = getForeground();

    ...//igual que el viejo método paint() hasta que dibujamos el rectángulo:
    if (fillSquare) {
        g.fillRect(x, y, w, h);
        fillSquare = false;
    } else {
        g.setColor(bg);
        g.fillRect(x, y, w, h);
        g.setColor(fg);
        fillSquare = true;
    }
    ...//igual que el viejo método paint()
}
```

Observa que ya que no se limpia automáticamente el fondo, el código de dibujo debe ahora dibujar los rectángulos que no sean negros, así como los que lo sean.

Recortar el Area de Dibujo

Una técnica que se podría utilizar en el método `update()` es recortar su área de dibujo. Esto no funciona para el applet de ejemplo de esta página, ya que en cada marco cambia todo el área de dibujo. El recortado funciona bien, aunque, sólo cuando cambia una pequeña parte del área de dibujo -- como cuando el usuario arrastra un objeto a lo largo de la pantalla.

Puede realizar el recortado utilizando el método `clipRect()`. Un ejemplo de utilización de `clipRect()` se encuentra en la página [Aumentar el Rendimiento y la Aperiencia de una Animación](#).

Eliminar el Parpadeo: Implementar el Doble Buffer

La página anterior mostró cómo eliminar el parpadeo implementando el método `update()`. Podrías haber observado (dependiendo del rendimiento de tu ordenador) que el applet resultante, aunque no parpadea, se arrastra un poco. Esto es, en lugar de actualizarse completamente el área de dibujo (o marco) de una vez, algunas veces, una parte se actualiza antes que la parte de su derecha, causando un dibujo bacheado entre columnas.

Puedes utilizar el doble buffer para evitar este efecto de arrastre forzando que todo el marco se dibuje de una sola vez. Para implementar el doble buffer, se necesita crear un buffer fuera de pantalla (normalmente llamado (backbuffer o buffer fuera de pantalla), dibujar en él, y luego mostrar la imagen resultante en la pantalla.

Aquí tienes el [código](#) para el ejemplo de animación de gráficos, modificado para implementar el doble buffer. Abajo tienes el applet resultante en acción.

Para crear un buffer fuera de pantalla con el AWT, primero necesitas crear una imagen del tamaño apropiado y luego obtener un contexto gráfico para manipular la imagen. Abajo tiene el código que hace esto:

```
//Donde se declaren las Variables de ejemplar:
Dimension offDimension;
Image offImage;
Graphics offGraphics;
...
//en el método update(), donde d contiene el tamaño del área de dibujo en la
pantalla:
if ( (offGraphics == null)
    || (d.width != offDimension.width)
    || (d.height != offDimension.height) ) {
    offDimension = d;
    offImage = createImage(d.width, d.height);
    offGraphics = offImage.getGraphics();
}
```

Abajo, en negrita, está el nuevo código de dibujo en el método `update()`. Observa que el código de dibujo ahora limpia completamente el fondo, pero no causa parpadeo porque el código está dibujando en el buffer fuera de pantalla, no en la pantalla. Observa también que todas las llamadas a `fillRect()` se realizan al buffer fuera de pantalla. El resultado final se muestra en la pantalla justo antes de que el método `update()` retorne.

```
public void update(Graphics g) {
    ...//Primero, inicializa las variables y crea el buffer fuera de pantalla
    //Luego borra la imagen anterior:
    offGraphics.setColor(getBackground());
    offGraphics.fillRect(0, 0, d.width, d.height);
    offGraphics.setColor(Color.black);

    ...//Hace todo lo que hacia el viejo método paint() -- hasta que dibujamos el
rectángulo
    if (fillSquare) {
        offGraphics.fillRect(x, y, w, h);
        fillSquare = false;
    } else {
        fillSquare = true;
    }
}
```

```
...//El resto es exactamente igual que el viejo método paint() hasta casi el final
    //donde añadimos lo siguiente:
    //dibuja la imagen en la pantalla.
    g.drawImage(offImage, 0, 0, this);
}
```

No es necesario que el método `update()` llame al método `paint()`. Todo lo necesario es que el método `update()` de alguna forma dibuje su buffer fuera de pantalla en la pantalla, y que el método `paint()` pueda dibujar la imagen apropiada cuando sea llamado directamente por el AWT.

Podrías preguntarte por qué se crean la imagen fuera de pantalla y el contexto gráfico dentro del método `update()`, en vez de hacerlo (por ejemplo) en el método `start()`. La razón es que la imagen y el contexto gráfico dependen del tamaño del área de dibujo del Panel del Applet y del tamaño del área de dibujo del componente y no son válidos hasta que el componente se haya dibujado por primera vez en la pantalla.

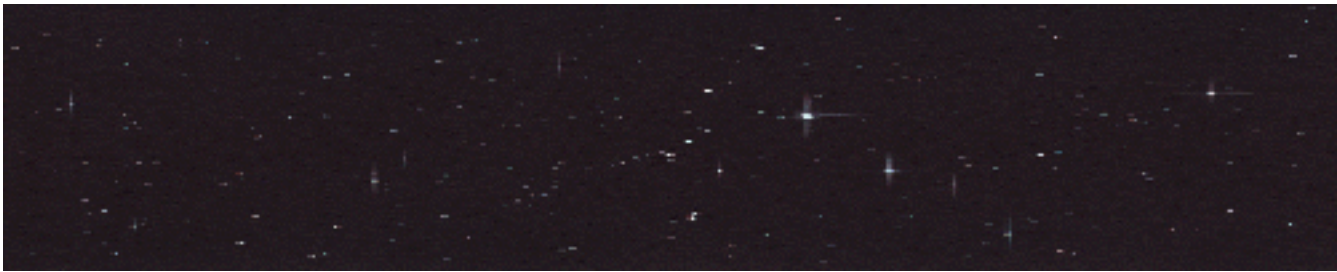
Mover una Imagen a través de la Pantalla

Esta página genera un applet de ejemplo que mueve una imagen (un cohete que realmente parece un trozo de pizza) delante de una imagen de fondo (un campo de estrellas). Esta página sólo muestra el código para un applet. El código para una aplicación sería similar excepto por el código utilizado para cargar las imágenes, como se describió en [Cargar Imágenes](#).

Abajo tienes las dos imágenes utilizadas por el applet:



rocketship.gif:



starfield.gif:

Nota: la imagen rocketship tiene un fondo transparente. El fondo transparente hace que la imagen del cohete parezca tener forma de coche sin importar el color del fondo donde se esté dibujando. Si el fondo del cohete no fuera transparente, en vez de la ilusión de ver un cohete moviéndose por el espacio, se vería un cohete dentro de un rectángulo que se mueve por el espacio.

Aquí tienes el applet en acción. Recuerda que se puede pulsar sobre el applet para detener y arrancar la animación

El código que realiza esta animación no es complejo. Esencialmente, es la plantilla de animación del applet, más un código de doble buffer que vió en la página anterior, más unas cuantas líneas de código adicionales. El código adicional, carga las imágenes, dibuja la imagen de fondo, y utiliza un sencillo algoritmo que determina donde dibujar la imagen en movimiento. Aquí tienes el código adicional:

```
...//Donde se declaren las variables de ejemplar:
```

```
Image stars;  
Image rocket;
```

```
...//en el método init():
```

```
stars = getImage(getCodeBase(), "../images/starfield.gif");  
rocket = getImage(getCodeBase(), "../images/rocketship.gif");
```

```
...//en el método update():
```

```
//dibujar el marco dentro de la imagen.
```

```
paintFrame(offGraphics);
```

...//Un nuevo método:

```
void paintFrame(Graphics g) {
    Dimension d = size();
    int w;
    int h;

    //Si tenemos una anchura y altura válidas de la imagen de fondo
    //la dibujamos.
    w = stars.getWidth(this);
    h = stars.getHeight(this);
    if ((w > 0) && (h > 0)) {
        g.drawImage(stars, (d.width - w)/2,
                    (d.height - h)/2, this);
    }

    //Si tenemos una anchura y altura válidas de la imagen móvil
    //la dibujamos.
    w = rocket.getWidth(this);
    h = rocket.getHeight(this);
    if ((w > 0) && (h > 0)) {
        g.drawImage(rocket, ((frameNumber*5) % (w + d.width)) - w,
                    (d.height - h)/2, this);
    }
}
```

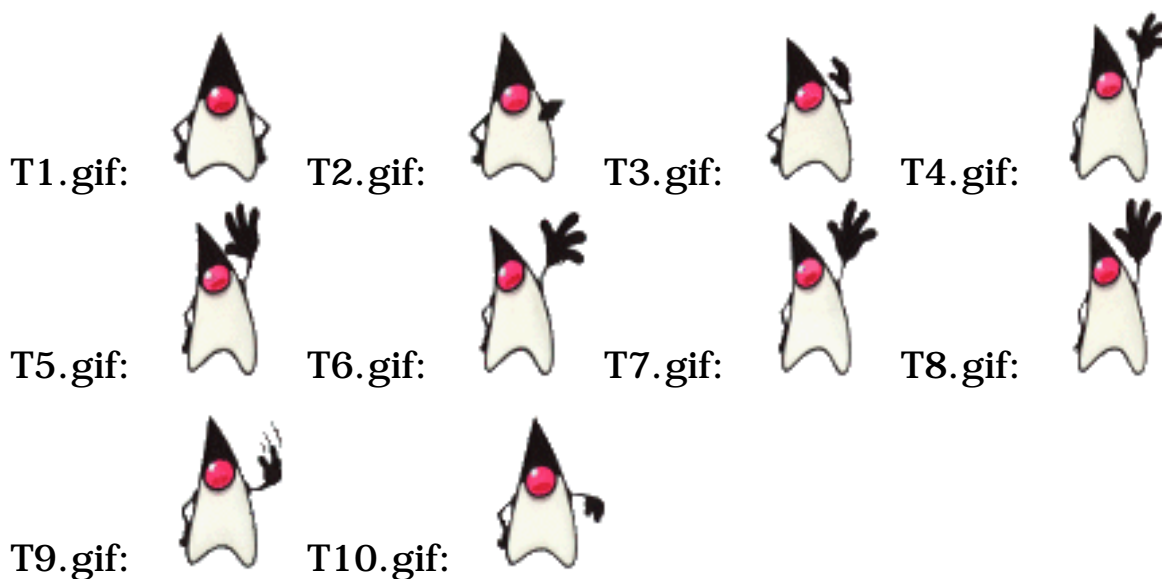
Se podría pensar que este programa no necesita limpiar el fondo ya que utiliza una imagen de fondo. Sin embargo, todavía es necesario limpiar el fondo. Una razón es que el applet normalmente empieza a dibujar antes de que las imágenes estén cargadas totalmente. Si la imagen del cohete se cargara antes que la imagen de fondo, vería partes de varios cohetes hasta que la imagen de fondo sea cargada. Otra razón es que si el área de dibujo del applet fuera más ancha que la imagen de fondo, por alguna razón, vería varios cohetes a ambos lados de la imagen de fondo.

Se podría resolver este problema retardando todo el dibujo hasta que las dos imágenes estuvieran totalmente cargadas. El segundo problema se podría resolver escalando la imagen de fondo para que cubriera todo el área del applet. Aprenderá como esperar a que las imágenes se carguen completamente en [Aumentar el Rendimiento y la Apariencia de una Animación](#), más adelante en esta lección. El escalado se describe en [Mostrar Imágenes](#).

Mostrar una Secuencia de Imágenes

El ejemplo de esta página mostrará lo básico para mostrar una secuencia de imágenes. La siguiente sección tiene trucos para aumentar el rendimiento y la apariencia de esta animación. Esta página sólo muestra el código del applet. El código de la aplicación es similar, excepto por el código utilizado para cargar las imágenes, cómo se describió en [Cargar Imágenes](#).

Abajo tienes las diez imágenes que utiliza este applet.



Aquí tienes el applet en acción; recuerda que se puede pulsar sobre el applet para detener o arrancar la animación.

El [código de este ejemplo](#) es incluso más sencillo que el de la página anterior, que movía una imagen. Aquí tiene el código que lo diferencia del ejemplo que movía la imagen:

```
. . .//Donde se declaren las variables de ejemplar:
Image duke[10];

. . .//En el método init():
for (int i = 1; i <= 10; i++) {
    images[i-1] = getImage(getCodeBase(),
                           "../.../images/duke/T"+i+".gif");
}

. . .//En el método update(), en vez de llamar a drawFrame():
offGraphics.drawImage(images[frameNumber % 10], 0, 0, this);
```


Aumentar la Apariencia y el Rendimiento de una Animación

Podrías haber observado dos cosas en la animación de la página anterior:

- Mientras se cargan las imágenes, el programa muestra algunas imágenes parcialmente y otras no las muestra.
- La carga de imágenes tarda mucho tiempo.

El problema de la muestra de imágenes parciales tiene fácil solución, utilizando la clase `MediaTracker`. `MediaTracker` también disminuye el tiempo que tardan en cargarse las imágenes. Otra forma de tratar el problema de la carga lenta es cambiar el formato de la imagen de alguna forma; esta página le ofrece algunas sugerencias para hacerlo.

Utilizar `MediaTracker` para Cargar Imágenes y Retardar el dibujo de éstas

La clase `MediaTracker` permite cargar fácilmente los datos de un grupo de imágenes y saber cuando se han cargado éstas completamente. Normalmente, los datos de una imagen no se cargan hasta que la imagen es dibujada por primera vez. Para pedir que los datos de un grupo de imágenes sean precargados asincrónicamente, puede utilizar las formas de `checkID()` y `checkAll()` que utilizan un argumento booleano, seleccionando el argumento a `true`. Para cargar los datos síncronamente (esperando a que los datos lleguen) utilice los métodos `waitForID()` y `waitForAll()`. Los métodos de `MediaTracker` que cargan los datos utilizan varios `Threads` en segundo plano para descargar los datos, resultando en un aumento de la velocidad.

Para comprobar el estado de carga de una imagen, se pueden utilizar los métodos `statusID()` y `statusAll()` de `MediaTracker`. Para comprobar si queda alguna imagen por cargar, puede utilizar los métodos `checkID()` y `checkAll()`.

Aquí tienes la [versión modificada del applet de ejemplo](#) que utiliza los métodos `waitForAll()` y `checkAll()` de `MediaTracker`. Hasta que se carguen todas las imágenes, el applet sólo muestra el mensaje "Please wait...". Puede ver la documentación de la clase [MediaTracker](#) para ver un ejemplo que dibuja el fondo inmediatamente pero espera a dibujar las imágenes animadas.

Aquí tienes el applet en acción:

Abajo tienes el código modificado que utiliza `MediaTracker` como ayuda para retardar el dibujo de las imágenes. Las diferencias se han marcado en **negrita**.

```
...//Donde se declaren las variables de ejemplar:  
MediaTracker tracker;
```

```
...//En el método init():  
tracker = new MediaTracker(this);  
for (int i = 1; i <= 10; i++) {  
    images[i-1] = getImage(getCodeBase(),  
                           ".../.../images/duke/T"+i+".gif");  
    tracker.addImage(images[i-1], 0);
```

```

}

...//Al principio del método run():
try {
    //Empieza la carga de imágenes. Esperar hasta que se hayan cargado
    tracker.waitForAll();
} catch (InterruptedException e) {}

...//Al principio del método update():
//Si no se han cargado todas las imágenes, borrar el fondo
//y mostrar la cadena de estado.
if (!tracker.checkAll()) {
    g.clearRect(0, 0, d.width, d.height);
    g.drawString("Please wait...", 0, d.height/2);
}

//Si las imágenes se han cargado, dibujarlas
else {
    ...//el mismo código de antes...
}

```

Acelerar la Carga de Imágenes

Tanto si se utiliza MediaTracker como si no, la carga de imágenes utilizando URLs (cómo hacen normalmente los applets) tarda mucho tiempo. La mayoría del tiempo se consume en inicializar las conexiones HTTP. Cada fichero de imagen requiere una conexión HTTP separada, y cada conexión tarda varios segundos en inicializarse. La técnica para evitar esto es combinar las imágenes en un sólo fichero. Se puede además aumentar el rendimiento utilizando algún algoritmo de compresión, especialmente uno diseñado para imágenes móviles.

Una forma sencilla de combinar imágenes en un único fichero es crear una tira de imágenes. Aquí tienes un ejemplo de una tira de imágenes:



jack.gif:

Para dibujar una imagen de la tira, primero se debe seleccionar el área de recorte al tamaño de una imagen. Cuando se dibuje la tira de imágenes, desplázalo a la izquierda (si es necesario) para que sólo aparezca dentro del área de dibujo la imagen que se quiere. Por ejemplo:

```

//imageStrip es el objeto Image que representa la tira de imágenes.
//imageWidth es el tamaño individual de una imagen.
//imageNumber es el número (desde 0 a numImages) de la imagen a dibujar.
int stripWidth = imageStrip.getWidth(this);
int stripHeight = imageStrip.getHeight(this);
int imageWidth = stripWidth / numImages;

```

```
g.clipRect(0, 0, imageWidth, stripHeight);  
g.drawImage(imageStrip, -imageNumber*imageWidth, 0, this);
```

Si se quiere que la carga de imágenes sea aún más rápida, se debería buscar un formato de compresión de imágenes, especialmente cómo Flic que realiza una compresión inter-marcos.

Problemas más comunes con los Gráficos (y sus Soluciones)

Problema: No se donde poner mi código de dibujo.

- El código de dibujo pertenece al método `paint()` de un componente personalizado. Se pueden crear componentes personalizados creando una subclase de Canvas, Panel, o Applet. Puedes ver [Cómo utilizar la clase Canvas](#) para más información sobre los componentes personalizados. Por eficiencia, una vez que el código de dibujo funciona, puedes modificarlo y llevarlo al método `update()` (aunque aún deberías implementar el método `paint()`), como se describe en [Eliminar el Parpadeo](#).

Problema: Las cosas que dibujo no se muestran.

- Comprueba si tu componente se muestra totalmente. [Problemas más Comunes con los Componentes](#) podría ayudarte con esto.

Problema: Estoy utilizando el mismo código que el ejemplo del tutorial, pero no funciona ¿por qué?

- ¿El código ejecutado es exactamente el mismo código que el del tutorial? Por ejemplo, si el ejemplo del tutorial tiene el código en los métodos `<>paint()` o `update()`, entonces estos métodos deberían ser el único lugar donde se garantiza que el código funciona.

Problema: ¿Cómo puedo dibujar líneas punteadas y patrones?

- Actualmente el API para gráficos primitivos del AWT está bastante limitado. Por ejemplo, sólo soporta una anchura de línea. Puedes simular las líneas punteadas, dibujando varias veces con un espacio de un píxel o dibujando rectángulos rellenos. El AWT tampoco soporta los patrones de relleno.

Si no has visto tu problema es esta lista, puedes ver [Problemas Comunes con los Componentes](#) y [Problemas Comunes con la Distribución](#)

Introducción al API 2D de Java

El API 2D de Java introducido en el JDK 1.2 proporciona gráficos avanzados en dos dimensiones, texto, y capacidades de manejo de imágenes para los programas Java a través de la extensión del AWT. Este paquete de rendering soporta líneas artísticas, texto e imágenes en un marco de trabajo flexible y lleno de potencia para desarrollar interfaces de usuario, programas de dibujo sofisticados y editores de imágenes.

El API 2D de Java proporciona:

- Un modelo de rendering uniforme para pantallas e impresoras.
- Un amplio conjunto de primitivos geométricos, como curvas, rectángulos, y elipses y un mecanismo para renderizar virtualmente cualquier forma geométrica.
- Mecanismos para detectar esquinas de formas, texto e imágenes.
- Un modelo de composición que proporciona control sobre cómo se renderizan los objetos solapados.
- Soporte de color mejorado que facilita su manejo.
- Soporte para imprimir documentos complejos.

Estos tópicos se explican en las siguientes páginas:

- [Dibujado 2D de Java](#)
 - [Systema de Coordenadas](#)
 - [formas](#)
 - [Texto](#)
 - [Imágenes](#)
 - [Impresión](#)
-

Rendering en Java 2D

El mecanismo de rendering básico es el mismo que en las versiones anteriores del JDK -- el sistema de dibujo controla cuando y como dibuja un programa. Cuando un componente necesita ser mostrado, se llama automáticamente a su método `paint` o `update` dentro del contexto `Graphics` apropiado.

El API 2D de Java presenta [java.awt.Graphics2D](#), un nuevo tipo de objeto `Graphics`. `Graphics2D` descende de la clase [Graphics](#) para proporcionar acceso a las características avanzadas de rendering del API 2D de Java.

Para usar las características del API 2D de Java, tenemos que forzar el objeto `Graphics` paado al método de dibujo de un componente a un objeto `Graphics2D`.

```
public void Paint (Graphics g) {  
    Graphics2D g2 = (Graphics2D) g;  
    ...  
}
```

Contexto de Rendering de Graphics2D

Al conjunto de atributos de estado asociados con un objeto `Graphics2D` se le conoce como Contexto de Rendering de Graphics2D. Para mostrar texto, formas o imágenes, podemos configurar este contexto y luego llamar a uno de los métodos de rendering de la clase `Graphics2D`, como `draw` o `fill`. Como muestra la siguiente figura, el contexto de rendering de `Graphics2D` contiene varios atributos.



El estilo de lápiz que se aplica al exterior de una forma. Este atributo `stroke` nos permite dibujar líneas con cualquier tamaño de punto y patrón de sombreado y aplicar finalizadores y decoraciones a la línea.



El estilo de relleno que se aplica al interior de la forma. Este atributo `paint` nos permite rellenar formas con colores sólidos, gradientes o patrones.



El estilo de composición se utiliza cuando los objetos dibujados se solapan con objetos existentes.



La transformación que se aplica durante el dibujado para convertir el objeto dibujado desde el espacio de usuario a las coordenadas de espacio del dispositivo. También se pueden aplicar otras transformaciones opcionales como la traducción, rotación escalado, recortado, a través de este atributo.



El Clip que restringe el dibujado al área dentro de los bordes de la Shape se utiliza para definir el área de recorte. Se puede usar cualquier Shape para definir un clip.



La fuente se usa para convertir cadenas de texto.



Punto de Rendering que especifican las preferencias en cuanto a velocidad y calidad. Por ejemplo, podemos especificar si se debería usar antialiasing, si está disponible.

Para configurar un atributo en el contexto de rendering de Graphics2D, se usan los métodos set Attribute:

- setStroke
- setPaint
- setComposite
- setTransform
- setClip
- setFont
- setRenderingHints

Cuando configuramos un atributo, se el pasa al objeto el atributo apropiado. Por ejemplo, para cambiar el atributo paint a un relleno de gradiente azul-gris, deberíamos construir el objeto GradientPaint y luego llamar a setPaint.

```
gp = new GradientPaint(0f,0f,blue,0f,30f,green);  
g2.setPaint(gp);
```

Graphics2D contiene referencias a sus objeto atributos -- no son clonados. Si modificamos un objeto atributo que forma parte del contexto Graphics2D, necesitamos llamar al método set para notificarlo al contexto. La modificación de un atributo de un objeto durante el

renderin puede causar comportamientos impredecibles.

Métodos de rendering de Graphics2D

Graphics2D proporciona los siguientes métodos generales de dibujado que pueden ser usados para dibujar cualquier primitivo geométrico, texto o imagen:

- draw--dibuja el exterior de una forma geométrica primitiva usando los atributos stroke y paint.
- fill--dibuja cualquier forma geométrica primitiva relleno su interior con el color o patrón especificado por el atributo paint.
- drawString--dibuja cualquier cadena de texto. El atributo font se usa para convertir la fuente a glyphs que luego son rellenos con el color o patrón especificados por el atributo paint.
- drawImage--dibuja la imagen especificada.

Además, Graphics2D soporta los métodos de rendering de Graphics para formas particulares, como drawOval y fillRect.

Sistema de Coordenadas

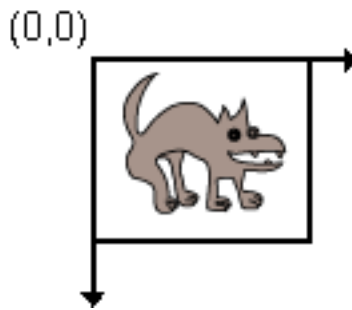
El sistema 2D de Java mantiene dos espacios de coordenadas.

- El espacio de usuario es el el espacio en que se especifican los gráficos primitivos.
- El espacio de dispositivo es el sistema de coordenadas para un diopositivo de salida, como una pantalla, una ventana o una impresora.

El espacio de usuario es un sistema de coordenadas lógicas independiente del dispositivo: el espacio de coordenadas que usan nuestros programas. Todos los geométricos pasados a las rutinas Java 2D de rendering se especifican en coordenadas de espacio de usuario.

Cuando se utiliza la transformación por defecto desde el espacio de usuario al espacio de dispositivo, el origen del espacio de usuario es la esquina superior izquierda del área de dibujo del componente. La coordena X se incrementa hacia la derecha, y la coordena Y hacia abajo, como se muestra la siguiente figura.

El espacio de dispositivo es un sistema de coordenadas dependiente del dispositivo que varía de acuerdo a la fuente del dispositivo. Aunque el sistema de coordenadas para una ventana o una pantalla podría ser muy distinto que para una impresora, estas diferencias son invisibles para los programas Java. Las conversiones necesarias entre el espacio de usuario y el espacio de dispositivo se realizan automáticamente durante el dibujado.



Formas

Las clases del paquete `java.awt.geom` definen gráficos primitivos comunes, como puntos, líneas, curvas, arcos, rectángulos y elipses.

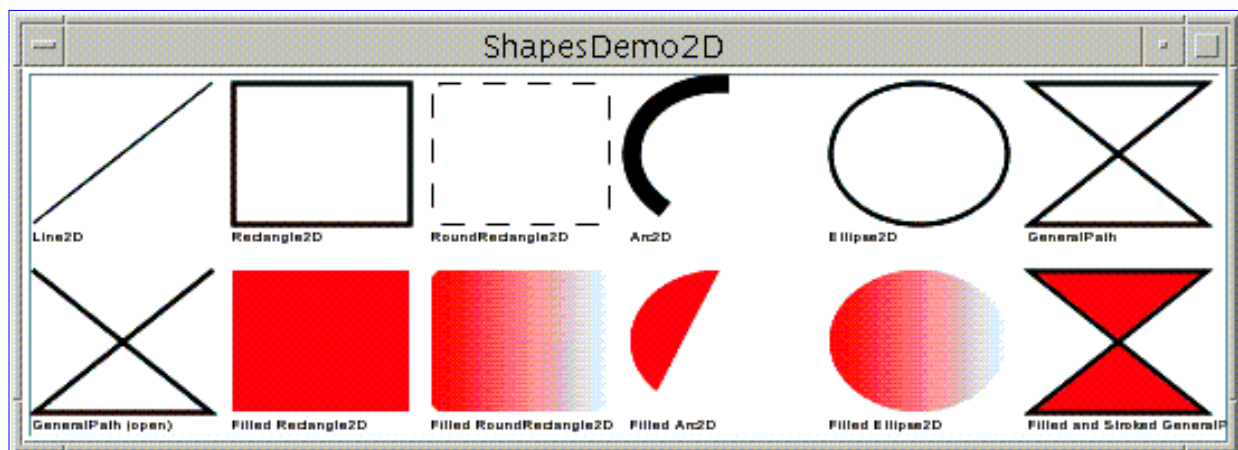
Clases en el paquete `java.awt.geom`

Arc2D	Ellipse2D	QuadCurve2D
Area	GeneralPath	Rectangle2D
CubicCurve2D	Line2D	RectangularShape
Dimension2D	Point2D	RoundRectangle2D

Excepto para `Point2D` y `Dimension2D`, cada una de las otras clases geométricas implementa el interface `Shape`, que proporciona un conjunto de métodos comunes para describir e inspeccionar objetos geométricos bi-dimensionales.

Con estas clases podemos crear de forma virtual cualquier forma geométrica y dibujarla a través de `Graphics2D` llamando al método `draw` o al método `fill`. Por ejemplo, las formas geométricas del siguiente applet están definidas usando los geométricos básicos de Java 2D.

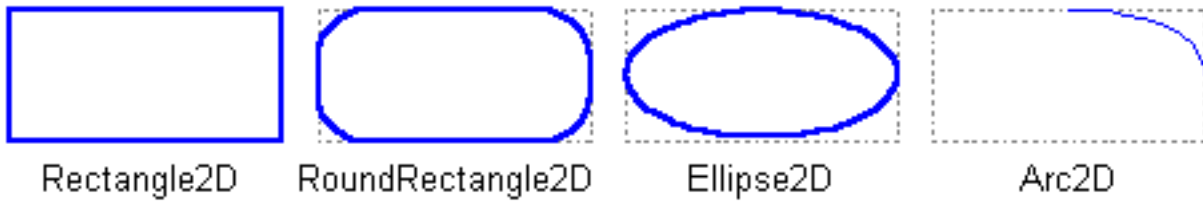
Si tienes curiosidad, el código del programa está en [ShapesDemo2D.java](#). La forma de dibujar y rellenar formas se describe en la siguiente lección [Mostrar Gráficos con Graphics2D](#).



Esta figura ha sido reducida para que quepa en la página.
Pulsa sobre la imagen para verla en su tamaño natural.

Formas Rectangulares

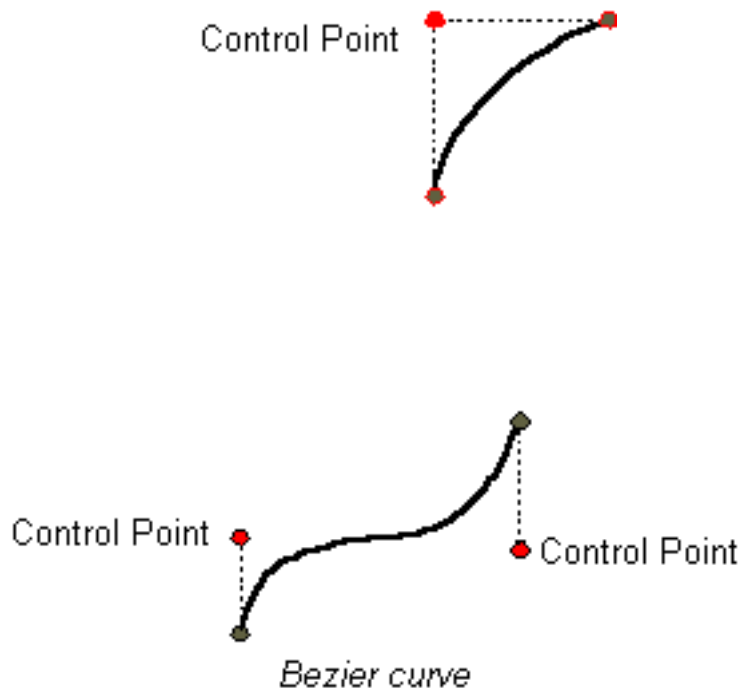
Los primitivos `Rectangle2D`, `RoundRectangle2D`, `Arc2D`, y `Ellipse2D` descienden del `RectangularShape`, que define métodos para objetos `Shape` que pueden ser descritos por una caja rectángulo. La geometría de un `RectangularShape` puede ser extrapolada desde un rectángulo que encierra completamente el exterior de la `Shape`.



QuadCurve2D y CubicCurve2D

La clase QuadCurve2D nos permite crear segmentos de curvas cuadráticas. Una curva cuadrática está definida por dos puntos finales y un punto de control.

La clase CubicCurve2D no permite crear segmentos de curvas cúbicas. Una curva cúbica está definida por dos puntos finales y dos puntos de control. Las siguientes figuras muestran ejemplos de curvas cuadráticas y cúbicas.



GeneralPath

La clase GeneralPath nos permite crear una curva arbitraria especificando una serie de posiciones a lo largo de los límites de la forma. Estas posiciones pueden ser conectadas por segmentos de línea, curvas cuadráticas o curvas cúbicas. La siguiente figura puede ser creada con 3 segmentos de línea y una curva cúbica.



Areas

Con la clase Area podemos realizar operaciones booleanas, como uniones, intersecciones y subtracciones, sobre dos objetos Shape cualesquiera. Esta técnica, nos permite crear rápidamente objetos Shape complejos sin tener que describir cada línea de segmento o cada curva.

Ozito

Texto

Cuando necesitemos mostrar texto, podemos usar uno de los componentes orientados a texto, como los componentes [Como usar Labels](#) o [Usar Componentes de Texto](#) de Swing. Cuando se utiliza un componente de texto, mucho del trabajo lo hacen por nosotros--por ejemplo, los objetos JTextComponent proporcionan soporte interno para chequeo de pulsaciones y para mostrar texto internacional.

Si queremos mostrar una cadena de texto estática, podemos dibujarla directamente a través de Graphics2D usando el método drawString. Para especificar la fuente, podemos usar el método setFont de Graphics2D.

Si queremos implementar nuestras propias rutinas de edición de texto o necesitamos más control sobre la distribución del texto que la que proporcionan los componentes de texto, podemos usar las clases del paquete java.awt.font.

Fuentes

Las formas que una fuente usa para representar los caracteres de una cadena son llamadas glyphs. Un caracter particular o una combinación de caracteres podría ser representada como uno o más glyphs. Por ejemplo, á podría ser representado por dos glyphs, mientras que la ligadura fi podría ser representada por un sólo glyph.

Se puede pensar en una fuente como en una colección de glyphs. Una sola fuente podría tener muchas caras, como una pesada, média, obliqua, gótica y regular. Todas las caras de una fuente tienen características tipográficas similares y pueden ser reconocidas como miembros de la misma familia. En otras palabras, una colección de glyphs con un estilo particular forma una cara de fuente; y una colección de caras de fuentes forma una familia de fuentes; y el conjunto de familias de fuentes forman el juego de fuentes disponibles en el sistema.

Cuando se utiliza el API 2D de Java, se especifican las fuentes usando un ejemplar de Font. Podemos determinar las fuentes disponibles en el sistema llamando al método estático

GraphicsEnvironment.getLocalGraphicsEnvironment y preguntando al GraphicsEnvironment devuelto. El método getAllFonts devuelve un array que contiene ejemplares Font para todas las fuentes disponibles en el sistema; getAvailableFontFamilyNames devuelve una lista de las familias de fuentes disponibles.

GraphicsEnvironment también describe la colección de dispositivos de dibujo de la plataforma, como pantallas e impresoras, que un programa Java puede utilizar. Esta información es usada cuando el sistema realiza

la conversión del espacio de usuario al espacio de dispositivo durante el dibujo.

Nota: En el JDK 1.1, las fuentes se describían con nombres lógicos que eran mapeados en diferentes caras de fuentes, dependiendo de las fuentes que estaban disponibles en una plataforma particular. Por razones de compatibilidad, Graphics2D también soporta la especificación de fuentes por el nombre lógico. Para obtener una lista de los nombres lógicos de la fuentes, se puede llamar a `java.awt.Toolkit.getFontList`.

Distribución de Texto

Antes de poder mostrar el texto, debe ser distribuido para que los caracteres sean representados por los glyphs apropiados en las posiciones apropiadas. Si estamos usando Swing, podemos dejar que JLabel o JTextComponent manejen la distribución de texto por nosotros. JTextComponent soporta texto bidireccional y está diseñada para manejar las necesidades de la mayoría de las aplicaciones intencionales. Para más información sobre el uso de los componentes de texto Swing, puedes ver [Usar Componentes de Texto](#).

Si no estamos usando un componente de texto Swing para mostrar el texto automáticamente, podemos usar uno de los mecanismos de Java 2D para manejar la distribución de texto.

- Si queremos implementar nuestras propias rutinas de edición de texto, podemos usar la clase TextLayout para manejar la distribución de texto, iluminación y detección de pulsación. Las facilidades proporcionadas por TextLayout manejan muchos casos comunes, incluyendo cadenas con fuentes mezcladas, lenguajes mezclados y texto bidireccional. Para más información sobre TextLayout, puedes ver [Manejar la distribución de Texto](#).
- Si queremos un control total sobre la forma y posición de nuestro texto, podemos construir nuestro propio objeto GlyphVector usando Font y renderizando cada GlyphVector a través de Graphics2D. Para más información sobre la implementación de nuestro propio mecanismo de distribución de texto, puedes ver [Implementar un Mecanismo Personalizado de Distribución de Texto](#)

Imágenes

El API 2D de Java implementa un nuevo modelo de imagen que soporta la manipulación de imágenes de resolución fija almacenadas en memoria. Una nueva clase `Image` en el paquete `java.awt.image`, `BufferedImage`, puede ser usada para manipular datos de una imagen recuperados desde un fichero o una URL. Por ejemplo, se puede usar un `BufferedImage` para implementar doble buffer -- los elementos gráficos son dibujados fuera de la pantalla en el `BufferedImage` y luego son copiados a la pantalla a través de llamadas al método `drawImage` de `Graphics2D`. Las clases `BufferedImage` y `BufferedImageOp` también permiten realizar una gran variedad de operaciones de filtrado de imágenes como blur. El modelo de imagen productor/consumidor proporcionado en las versiones anteriores del JDK sigue siendo soportado por razones de compatibilidad.

Ozito

Imprimir

Todos los gráficos del AWT y de Java 2D, incluidos los gráficos compuestos y las imágenes, pueden ser dibujadas en una impresora usando el API de Java 2D. Este API proporciona características de composición de documentos que nos permite realizar dichas operaciones como cambiar el orden de impresión de las páginas.

Dibujar en la impresora es como dibujar en la pantalla. El sistema de impresión controla cuando se dibujan las páginas, cómo lo hace el sistema gráfico de la pantalla cuando un componente se dibuja en la pantalla.

Nuestra aplicación proporciona el sistema de impresión con información sobre el documento a imprimir, y el sistema de impresión determina cuando necesitar dibujar cada página. cuando las páginas necesitan ser dibujadas, el sistema de impresión llama al método print de la aplicación con el contexto Graphics apropiado. Para usar las características del API 2D de Java en la impresión, debemos forzar el objeto Graphics a un objeto Graphics2D, igual que se hace cuando se dibuja en la pantalla.

Mostrar Gráficos con Graphics2D

Esta lección nos muestra cómo usar Graphics2D para mostrar gráficos con líneas exteriores, estilos de relleno, transformación de gráficos mientras son dibujados, restricción de dibujo en un área particular y generalmente controla la forma y aspecto de los gráficos. También aprenderemos cómo crear objetos complejos combinando algunos simples y cómo detectar cuando el usuario pulsa sobre un gráfico primitivo. Estos tópicos se describen en las siguientes secciones:

Dibujar y rellenar gráficos primitivos

Esta sección ilustra cómo seleccionar el punteado y los atributos de dibujo para controlar la línea exterior y el relleno aplicado al objeto Shape y el texto.

Transformar formas, texto e imágenes

Esta sección muestra cómo modificar la transformación por defecto para que los objetos sean trasladados, girados, escalados o sombreados mientras son dibujados.

Recortar el área de dibujo

Se puede usar la forma como path de recortado -- el área en la que tiene lugar el dibujo.

Componer Gráficos

Esta sección ilustra los distintos estilos de composición soportados por AlphaComposite y muestra cómo seleccionar el estilo de composición en el contexto Graphics2D.

Controlar la Calidad de dibujo

Esta sección describe los trucos de dibujo que soporta Graphics2D y muestra cómo especificar nuestras preferencias entre la calidad de dibujo y la velocidad.

Construir formas complejas desde gráficos primitivos

Esta sección describe cómo realizar operaciones booleanas sobre objeto Shape usando la clase Area.

Soportar Interacción del Usuario

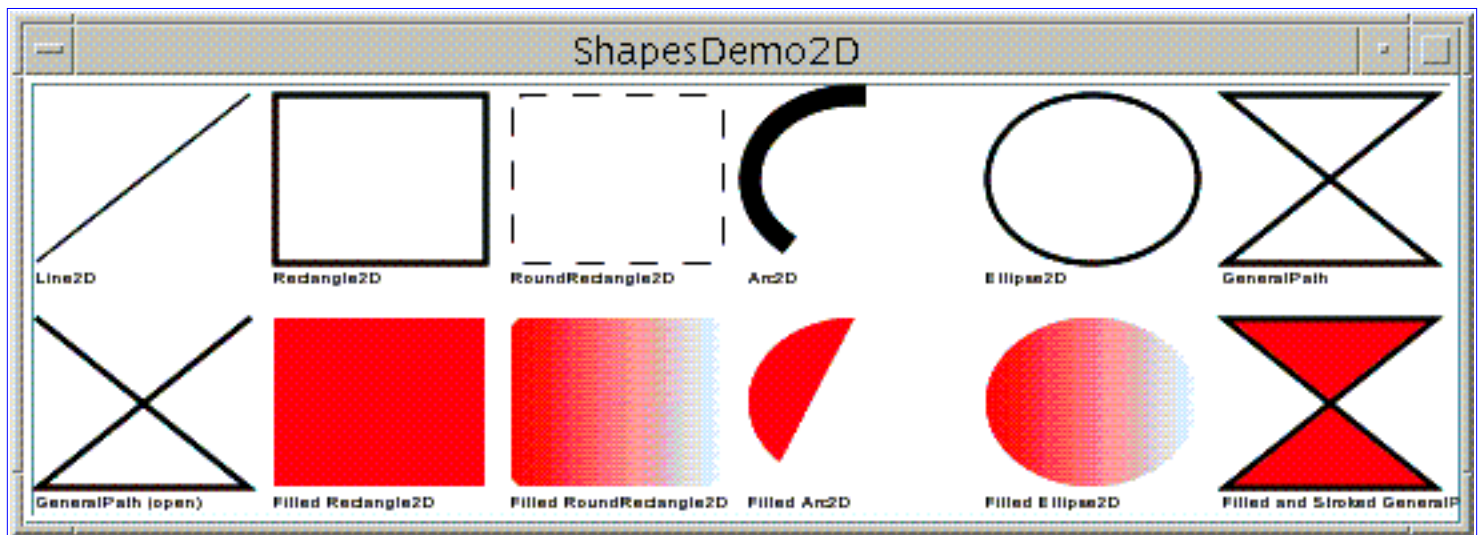
Esta sección muestra como realizar detección de pulsaciones sobre gráficos primitivos.

Ozito

Punteado y Relleno de Gráficos Primitivos

Cambiando el punteado y los atributos de dibujo en el contexto de Graphics2D, antes del dibujo, podemos fácilmente aplicar estilos divertidos de líneas y patrones de relleno para gráficos primitivos. Por ejemplo, podemos dibujar una línea punteada creando el objeto Stroke apropiado y llamando a setStroke para añadirlo al contexto Graphics2D antes de dibujar la línea. De forma similar, podemos aplicar un relleno de gradiente a un Shape creando un objeto GradientPaint y añadiendo al contexto Graphics2D llamando a setPaint antes de dibujar la Shape.

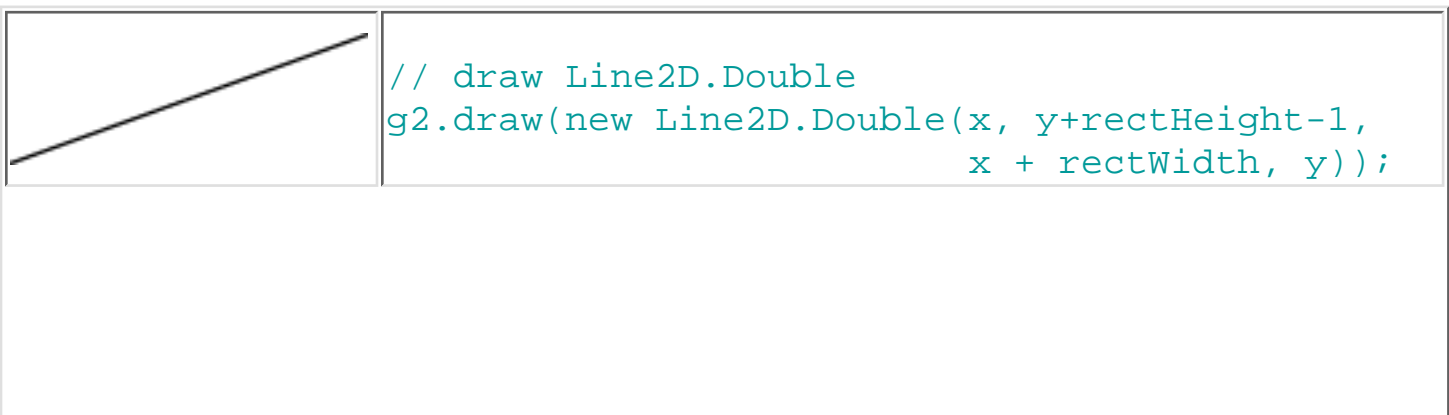
El siguiente applet demuestra cómo podemos dibujar formas geométricas usando los métodos Graphics2D draw y fill.








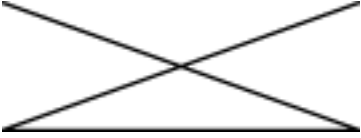



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

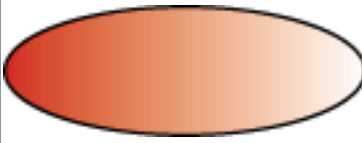

ShapesDemo2D.java contiene el código completo de este applet.

Cada una de las formas dibujadas por el applet está construidas de geometrías y está dibujada a través de Graphics2D. Las variables rectHeight y rectWidth de este ejemplo definen las dimensiones del espacio en que se dibuja cada forma, en pixels. La variables x e y cambian para cada forma para que sean dibujadas en formación de parrilla.



	<pre>// draw Rectangle2D.Double g2.setStroke(stroke); g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));</pre>
	<pre>// draw RoundRectangle2D.Double g2.setStroke(dashed); g2.draw(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));</pre>
	<pre>// draw Arc2D.Double g2.setStroke(wideStroke); g2.draw(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135, Arc2D.OPEN));</pre>
	<pre>// draw Ellipse2D.Double g2.setStroke(stroke); g2.draw(new Ellipse2D.Double(x, y, rectWidth, rectHeight));</pre>
	<pre>// draw GeneralPath (polygon) int x1Points[] = {x, x+rectWidth, x, x+rectWidth}; int y1Points[] = {y, y+rectHeight, y+rectHeight, y}; GeneralPath polygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x1Points.length); polygon.moveTo(x1Points[0], y1Points[0]); for (int index = 1; index < x1Points.length; index++) { polygon.lineTo(x1Points[index], y1Points[index]); };</pre>

	<pre> polygon.closePath(); g2.draw(polygon); </pre>
	<pre> // draw GeneralPath (polyline) int x2Points[] = {x, x+rectWidth, x, x+rectWidth}; int y2Points[] = {y, y+rectHeight, y+rectHeight, y}; GeneralPath polyline = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x2Points.length); polyline.moveTo (x2Points[0], y2Points[0]); for (int index = 1; index < x2Points.length; index++) { polyline.lineTo(x2Points[index], y2Points[index]); }; g2.draw(polyline); </pre>
	<pre> // fill Rectangle2D.Double (red) g2.setPaint(red); g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight)); </pre>
	<pre> // fill RoundRectangle2D.Double g2.setPaint(redtowhite); g2.fill(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10)); </pre>
	<pre> // fill Arc2D g2.setPaint(red); g2.fill(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135, Arc2D.OPEN)); </pre>

	<pre>// fill Ellipse2D.Double g2.setPaint(redtowhite); g2.fill (new Ellipse2D.Double(x, y, rectWidth, rectHeight));</pre>
	<pre>// fill and stroke GeneralPath int x3Points[] = {x, x+rectWidth, x, x+rectWidth}; int y3Points[] = {y, y+rectHeight, y+rectHeight, y}; GeneralPath filledPolygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x3Points.length); filledPolygon.moveTo(x3Points[0], y3Points[0]); for (int index = 1; index < x3Points.length; index++) { filledPolygon.lineTo(x3Points[index], y3Points[index]); } filledPolygon.closePath(); g2.setPaint(red); g2.fill(filledPolygon);</pre>

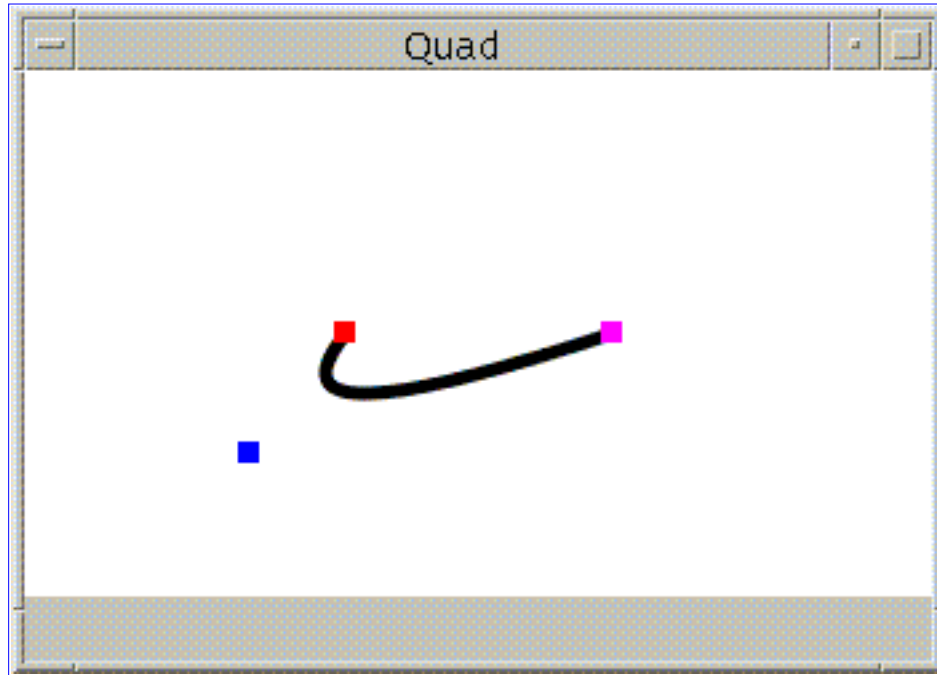
Observa que este ejemplo usa implementaciones de doble precisión de las clases geométricas. Donde sea posible, las implementaciones de los float y doble precisión de cada geométrico están proporcionados por clases internas.

Dibujar Curvas

Los applets Cubic y Quad demuestran como crear curvas cúbicas y cuadráticas usando CubicCurve2D y QuadCurve2D. Estos applets también demuestran como se dibujan las curvas con respecto al posicionamiento de los puntos de control permitiendonos interactivamente mover tanto los puntos de control como los puntos finales.

Ejemplo: Quad

El applet Quad demuestra una curva cuadrática, que es un segmento de curva que tiene dos puntos finales y un único punto de control. El punto de control determina la forma de la curva controlando tanto el punto de control como los vectores tangenciales de los puntos finales.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Quad.java](#) contiene el código completo de este applet.

Primero se crea una nueva curva cuadrática con dos puntos finales y un punto de control y las posiciones de los puntos se seleccionan con respecto al tamaño de la ventana.

```
QuadCurve2D.Double quad = new QuadCurve2D.Double();

Point2D.Double start, end, control;
start = new Point2D.Double();
one = new Point2D.Double();
control = new Point2D.Double();

quad.setCurve(start, one, control);

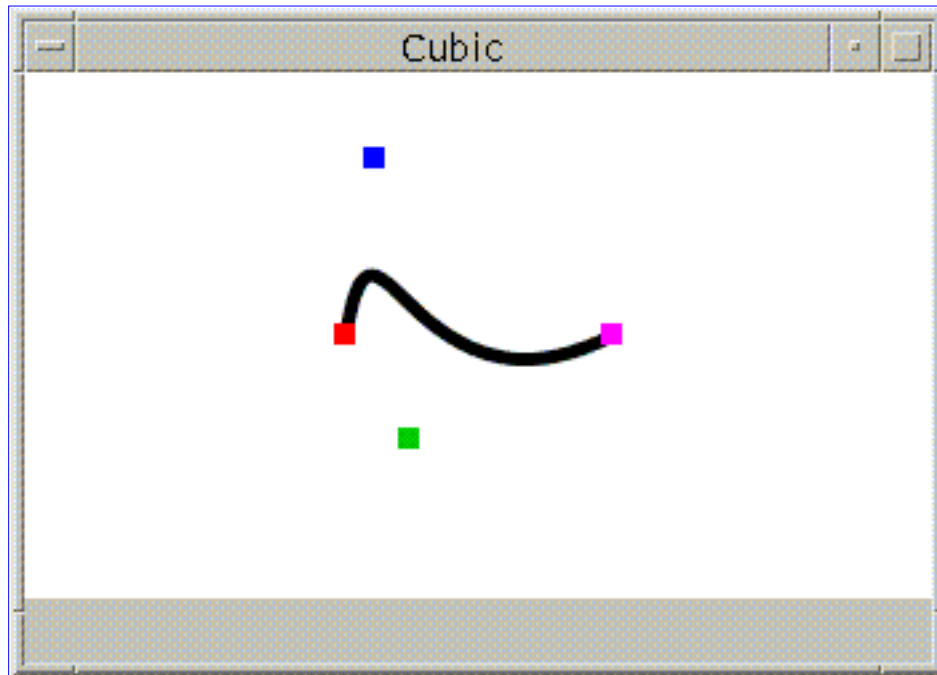
start.setLocation(w/2-50, h/2);
end.setLocation(w/2+50, h/2);
control.setLocation((int)(start.x)+50, (int)(start.y)-50);
```

Cada vez que el usuario mueva uno de los puntos, la curva se reseteará.

```
quad.setCurve(start, control, end);
```

Ejemplo: Cubic

El ejemplo Cubic demuestra una curva cúbica, que es un segmento de curva que tiene dos puntos finales y dos puntos de control. Cada punto de control determina la forma de la curva mediante el control de uno de los vectores tangenciales de un punto final. En el ejemplo Cubic, las cruces coloreadas se dibujan donde se encuentran los puntos de control y los puntos finales. El punto de control azul controla el punto final rojo y el punto de control verde controla el punto final magenta.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Cubic.java](#) contiene el código completo de este applet.

Una curva cúbica se crea con dos puntos finales y dos puntos de control y las localizaciones de los puntos se seleccionan con respecto al tamaño de la ventana.


```

CubicCurve2D.Double cubic = new CubicCurve2D.Double();

Point2D.Double start, end, one, two;
start = new Point2D.Double();
one = new Point2D.Double();
two = new Point2D.Double();
end = new Point2D.Double();

cubic.setCurve(start, one, two, end);

...

start.setLocation(w/2-50, h/2);
end.setLocation(w/2+50, h/2);
one.setLocation((int)(start.x)+25, (int)(start.y)-25);
two.setLocation((int)(end.x)-25, (int)(end.y)+25);

```

Como en el ejemplo Quad, la curva es reseteada cada vez que se mueven los pulsos.

```

cubic.setCurve(start, one, two, end);

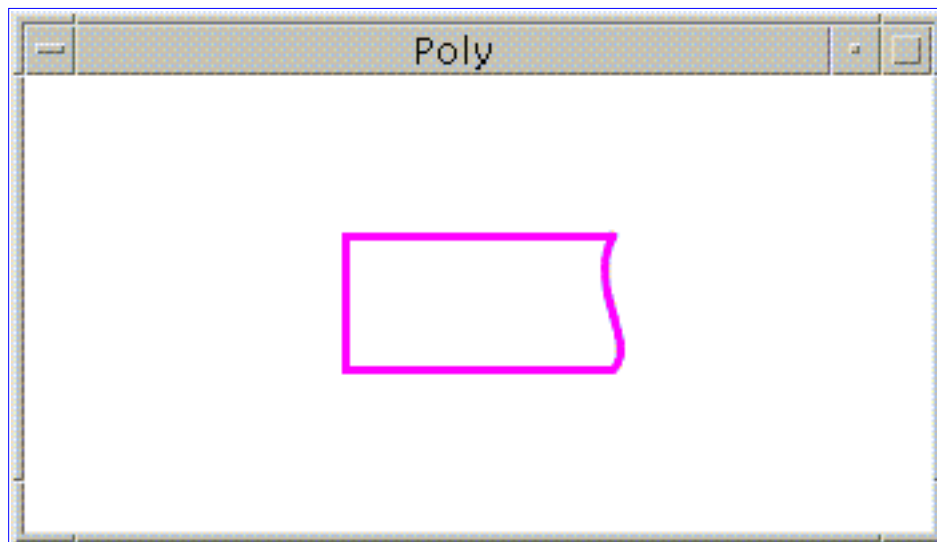
```

Dibujar formas arbitrarias

El ejemplo ShapesDemo usa GeneralPath para hacer polígonos en forma de cristales, pero también podemos usar GeneralPath para hacer formas arbitrarias tanto con líneas rectas como curvas.

Ejemplo: Odd_Shape

El ejemplo Odd_Shape usa GeneralPath para crear formas arbitrarias en la sección [Formas](#).



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Odd_Shape.java](#) contiene el código completo de este applet.

El siguiente código crea un nuevo GeneralPath y añade el primer punto al path.

```
GeneralPath oddShape = new GeneralPath();  
...
```

```
x = w/2 + 50;  
y = h/2 - 25;
```

```
x2 = x;  
y2 = y;
```

```
oddShape.moveTo(x, y);
```

Después de añadir el primer punto al path, se añaden tres líneas rectas.

```
x -= 100;  
oddShape.lineTo(x, y);  
y += 50;  
oddShape.lineTo(x, y);  
x += 100;  
oddShape.lineTo(x, y);
```

Finalmente, se añade una curva cúbica.

```
x += 10;  
y -= 10;  
x1 = x - 20;  
y1 = y - 20;  
oddShape.curveTo(x, y, x1, y1, x2, y2);
```

Definir Estilos de línea divertidos y Patrones de relleno.

Probablemente habrás observado en el ejemplo anterior algunas de las formas tiene línea punteadas o están rellenas con gradientes de dos colores. Usando las clases Stroke y Paint de Java 2D, podemos fácilmente definir estilos de línea divertidos y patrones de relleno

Estilos de Línea

Los estilos de línea está definida por el atributo stroke en el contexto Graphics2D. Para seleccionar el atributo stroke podemos crear un objeto BasicStroke y pasarlo dentro del método Graphics2D setStroke.

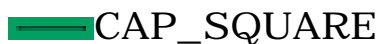
Un objeto BasicStroke contiene información sobre la anchura de la línea, estilo de uniones, estilos finales, y estilo de punteado. Esta información es usada cuando un Shape es dibujado con el método draw.

La anchura de línea es la longitud de la línea medida perpendicularmente a su trayectoria. La anchura de la línea se especifica como un valor float en las unidades de coordenadas de usuario, que es equivalente a 1/72 pulgadas cuando se utiliza la transformación por defecto.

El estilo de unión es la decoración que se aplica cuando se encuentran dos segmentos de línea. BasicStroke soporta tres estilos de unión: :



El estilo de finales es la decoración que se aplica cuando un segmento de línea termina. BasicStroke soporta tres estilos de finalización: :



El estilo de punteado define el patrón de las secciones opacas y transparentes aplicadas a lo largo de la longitud de la línea. Este estilo es definido por un array de punteada y una fase de punteado. El array de punteado define el patrón de punteado. Los elementos alternativos en el array representan la longitud del punteado y el espacio entre punteados en unidades de coordenadas de usuario.. El elemento 0 representa el primer punteado, el elemento 1 el primer espacio, etc. La fase de

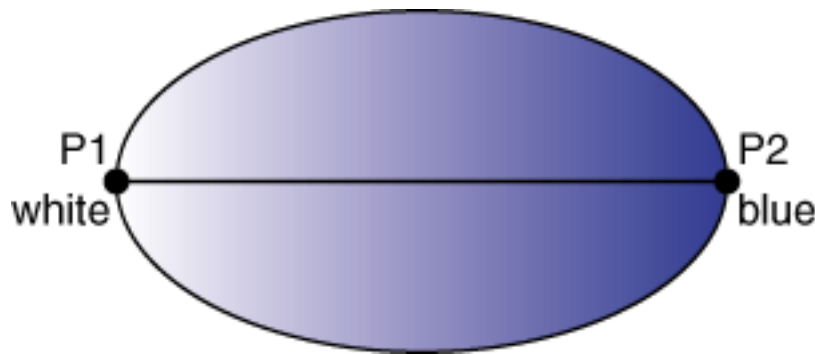
punteado es un desplazamiento en el patrón de punteado, también especificado en unidades de coordenadas de usuario. La fase de punteado indica que parte del patrón de punteado se aplica al principio de la línea.

Patrón de Relleno

Los patrones de rellenos están definidos por el atributo paint en el contexto Graphics2D. Para seleccionar el atributo paint, se crea un ejemplar de un objeto que implemente el interface Paint y se pasa dentro del método Graphics2D setPaint.

Tres clases implementan el interface Paint: Color, GradientPaint, y TexturePaint. GradientPaint y TexturePaint son nuevas en el JDK 1.2.

Para crear un GradientPaint, se especifica una posición inicial y un color y una posición final y otro color. El gradiente cambia proporcionalmente desde un color al otro a lo largo de la línea que conecta las dos posiciones.



El patrón para una TexturePaint esta definida por un BufferedImage. Para crear un TexturePaint, se especifica una imagen que contiene el patrón y un rectángulo que es usado para replicar y anclar el patrón.



Pattern Image



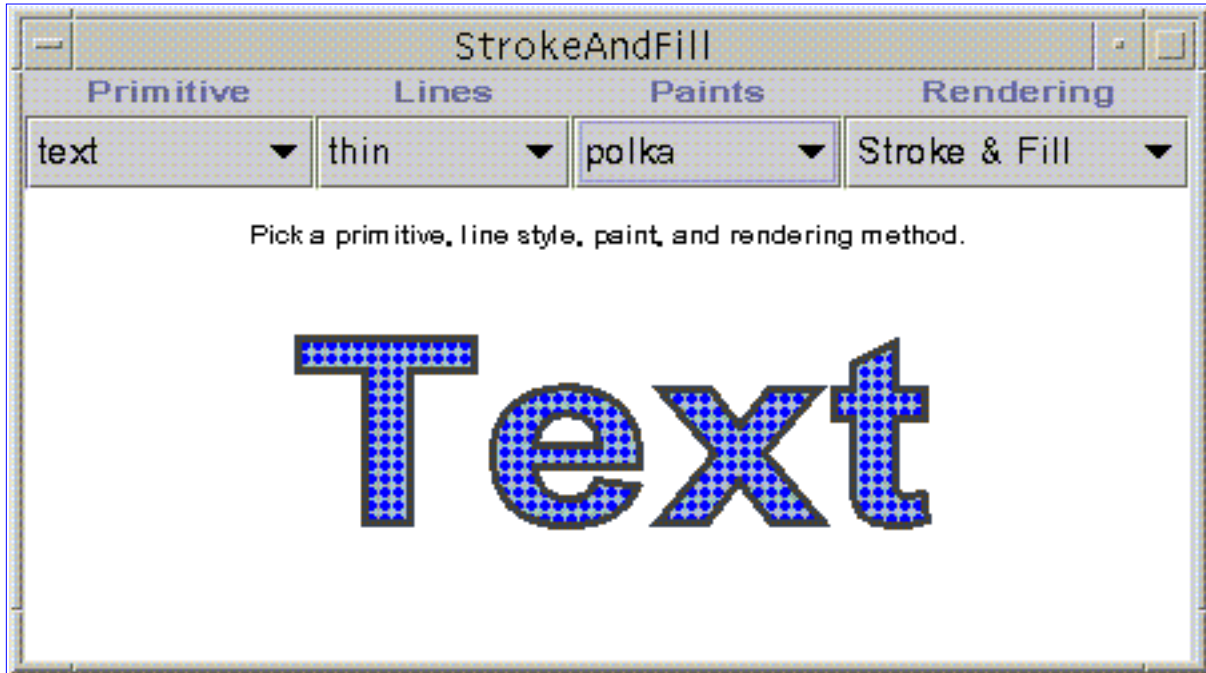
Rectangle Defining
Repetition Frequency



Large Rectangle Filled with
Resulting TexturePaint

Ejemplo: StrokeAndFill

El applet StrokeAndFill permite al usuario seleccionar un gráfico primitivo, un estilo de línea, un estilo de dibujo y o bien puntear el exterior del objeto, rellenarlo con el dibujo seleccionado, o puntear el objeto en blanco y rellenar el dibujo con el dibujo seleccionado.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[StrokeAndFill.java](#) contiene el código completo de este applet.

Los primitivos son inicializados e introducidos en un array de objeto Shape. El siguiente código crea un Rectangle y un Ellipse2D.Double y los introduce en el array shapes.

```
shapes[0] = new Rectangle(0, 0, 100, 100);
shapes[1] = new Ellipse2D.Double(0.0, 0.0, 100.0, 100.0);
```

Para crear un objeto Shape desde una cadena de texto, primero debemos crear un objeto TextLayout desde el texto de la cadena.

```
TextLayout textTl = new TextLayout("Text",
    new Font("Helvetica", 1, 96),
    new FontRenderContext(null, false, false));
```

Las siguientes líneas transforman el TextLayout para que sea

centrado en el origen y luego introduce el objeto Shape resultante de la llamada a getOutline dentro del array shapes.

```
AffineTransform textAt = new AffineTransform();
textAt.translate(0,
    (float)textTl.getBounds().getHeight());
shapes[2] = textTl.getOutline(textAt);
```

Podemos elegir un primitivo accediendo al índice apropiado dentro del array shapes.

```
Shape shape =
    shapes[Transform.primitive.getSelectedIndex()];
```

Cómo se realiza el dibujo dependen de las opciones elegidas.

- Cuando el usuario elige stroke, se llama a Graphics2D.draw para realizar el dibujo, Su se elige text como primitivo, las líneas son recuperadas y el dibujo se hace con el método draw.
- Cuando el usuario elige fill, se llama a Graphics2D.fill o Graphics2D.drawString para realizar el dibujado.
- Cuando el usuario elige stroke and fill, se llama a fill o drawString para rellenar el Shape, y luego se llama a draw para dibujar la línea exterior.

Nota: Para rellenar y puntear un gráfico primitivo, necesitamos hacer dos llamadas separadas a métodos: fill o drawString para rellenar el interior, y draw para dibujar el exterior.

Los tres estilos de línea usados en este ejemplo -- ancho, estrecho y punteado -- son ejemplares de BasicStroke.

```
// Sets the Stroke.
...
case 0 : g2.setStroke(new BasicStroke(3.0f)); break;
case 1 : g2.setStroke(new BasicStroke(8.0f)); break;
case 2 : float dash[] = {10.0f};
        g2.setStroke(new BasicStroke(3.0f,
            BasicStroke.CAP_BUTT,
            BasicStroke.JOIN_MITER,
            10.0f, dash, 0.0f));
        break;
```

El estilo de punteado de este ejemplo tiene 10 unidades de punteado alternados con 10 unidades de espacio. El principio

del patrón del punteado se aplica al principio de la línea -- la fase de punteado es 0.0.

En este ejemplo se usan tres estilos de dibujo -- sólido, gradiente y polka. El dibujo de color sólido es un ejemplar de `Color`, el gradiente un ejemplar de `GradientPaint`, y el patrón un ejemplar de `TexturePaint`.

```
// Sets the Paint.
...
case 0 : g2.setPaint(Color.blue); break;
case 1 : g2.setPaint(new GradientPaint(0, 0,
                                     Color.lightGray,
                                     w-250, h, Color.blue, false));
        break;
case 2 : BufferedImage bi = new BufferedImage(5, 5,
                                     BufferedImage.TYPE_INT_RGB);
        Graphics2D big = bi.createGraphics();
        big.setColor(Color.blue);
        big.fillRect(0, 0, 5, 5);
        big.setColor(Color.lightGray);
        big.fillOval(0, 0, 5, 5);
        Rectangle r = new Rectangle(0,0,5,5);
        g2.setPaint(new TexturePaint(bi, r));
        break;
```

Transformar Formas, Texto e Imágenes

Podemos modificar el atributo transform en el contexto Graphics2D para mover, rotar, escalar y modificar gráficos primitivos mientras son dibujados. El atributo transform está definido por un ejemplar de AffineTransform.

Graphics2D proporciona varios métodos para cambiar el atributo transform. Podemos construir un nuevo AffineTransform y cambiar el atributo transform de Graphics2D llamando al método setTransform.

AffineTransform define los siguientes métodos para hacer más sencilla la construcción de nuevas transformaciones:

- getRotateInstance
- getScaleInstance
- getShearInstance
- getTranslateInstance

De forma alternativa podemos usar uno de los métodos de transformación de Graphics2D para modificar la transformación actual. Cuando se llama a uno de esos métodos de conveniencia, la transformación resultante se concatena con la transformación actual y es aplicada durante el dibujo:

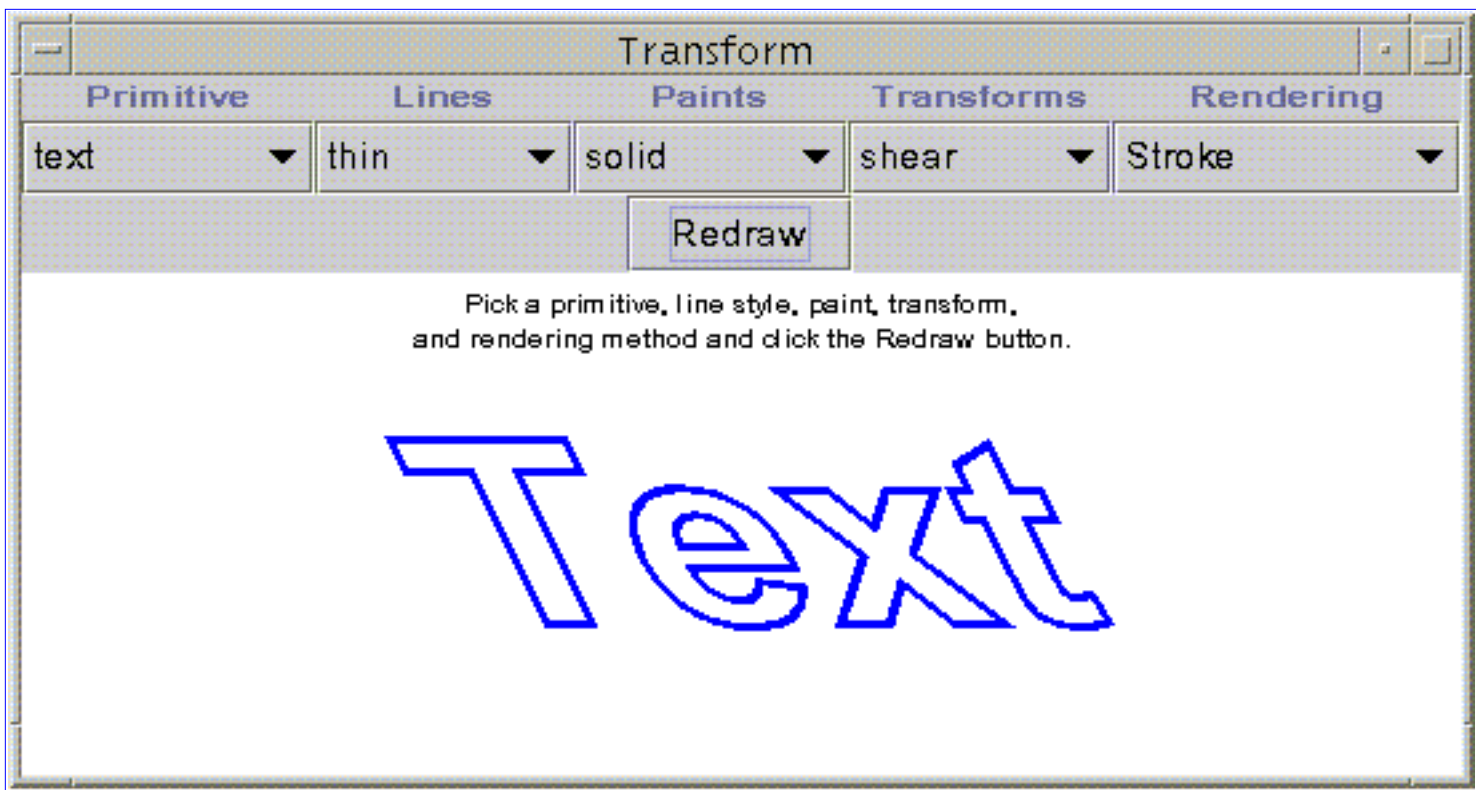
- rotate--para especificar un ángulo de rotación en radianes.
- scale--para especificar un factor de escala en direcciones x e y.
- shear--para especificar un factor de compartición en direcciones x e y
- translate--para especificar un desplazamiento de movimiento en direcciones x e y

También podemos construir directamente un AffineTransform y concatenarlo con la transformación actual llamando al método transform .

El método drawImage también está sobrecargado para permitirnos especificar un AffineTransform que es aplicada a la imagen a dibujar. Especificar un transform cuando se llama a drawImage no afecta al atributo transform de Graphics2D.

Ejemplo: Transform

El siguiente programa es el mismo que StrokeandFill, pero también permite al usuario elegir una transformación para aplicarla al objeto selecciona cuando se dibuje.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Transform.java](#) contiene el código completo de este applet.

Cuando se elige una opción de transformación, se modifica un ejemplar de `AffineTransform` y es concatenado con una transformación de movimiento que mueve la `Shape` hacia el centro de la ventana. La transformación resultante se pasa al método `setTransform` para seleccionar el atributo `transform` de `Graphics2D`

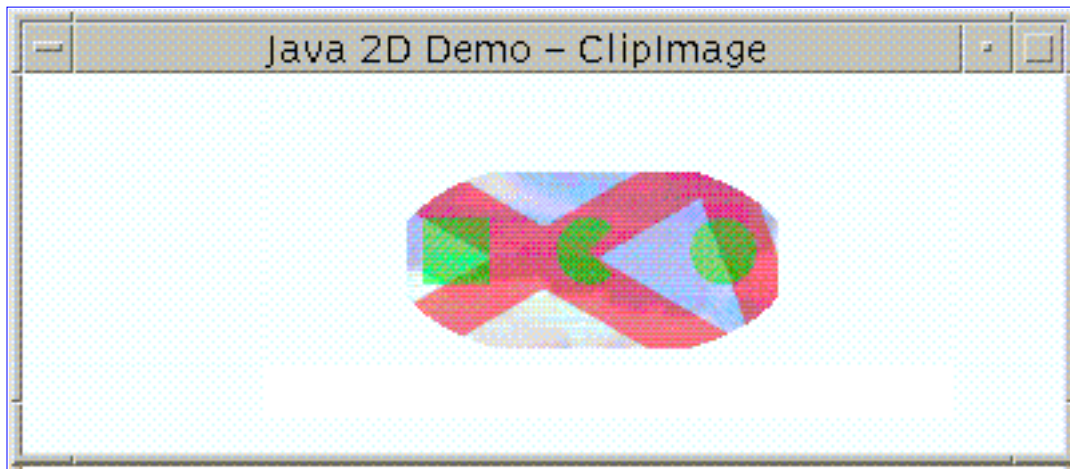
```
switch (Transform.trans.getSelectedIndex()){
case 0 : at.setToIdentity();
        at.translate(w/2, h/2); break;
case 1 : at.rotate(Math.toRadians(45)); break;
case 2 : at.scale(0.5, 0.5); break;
case 3 : at.shear(0.5, 0.0); break;
...
AffineTransform toCenterAt = new AffineTransform();
toCenterAt.concatenate(at);
toCenterAt.translate(-(r.width/2), -(r.height/2));
g2.setTransform(toCenterAt);
```

Recortar la Región de Dibujo

Cualquier Shape puede usarse como un path de recortado que restringe las porciones del área de dibujo que serán renderizadas. El path de recortado forma parte del contexto Graphics2D; para seleccionar el atributo clip, se llama a Graphics2D.setClip y se pasa a la Shape que define el path de recortado que queramos usar. Podemos reducir el path de recortado llamando al método clip y pasándolo en otra Shape; el atributo clip se configura a la intersección entre el clip actual y el Shape especificado.

Ejemplo: ClipImage

Este ejemplo anima un path de recortado para revelar diferentes porciones de una imagen.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[ClipImage.java](#) contiene todo el código de este applet. El applet requiere el fichero de imagen [clouds.jpg](#).

El path de recortado está definido por la intersección de una elipse y un rectángulo cuyas dimensiones son aleatorias. La elipse se pasa al método setClip, y luego se llama al método clip para seleccionar el path de recortado a la intersección entre la elipse y el rectángulo.

```
private Ellipse2D ellipse = new Ellipse2D.Float();
private Rectangle2D rect = new Rectangle2D.Float();
...
ellipse setFrame(x, y, ew, eh);
g2.setClip(ellipse);
rect.setRect(x+5, y+5, ew-10, eh-10);
g2.clip(rect);
```

Ejemplo: Starry

Un área de recortado también puede ser creada desde una cadena de texto existente. El siguiente ejemplo crea un `TextLayout` con la cadena `The Starry Night`. Luego, obtiene una línea exterior del `TextLayout`. El método `TextLayout.getOutline` devuelve un objeto `Shape` y un `Rectangle` creado a partir de los límites del objeto `Shape`. Los límites contienen todos los pixels que layout puede dibujar. El color en el contexto gráfico se selecciona a azul y se dibuja la figura exterior de la forma, como ilustran la siguiente imagen y el fragmento de código.

The Starry Night

```
FontRenderContext frc = g2.getFontRenderContext();
Font f = new Font("Helvetica", 1, w/10);
String s = new String("The Starry Night");
TextLayout tl = new TextLayout(s, f, frc);
AffineTransform transform = new AffineTransform();
Shape outline = textTl.getOutline(null);
Rectangle r = outline.getBounds();
transform = g2.getTransform();
transform.translate(w/2-(r.width/2), h/2+(r.height/2));
g2.transform(transform);
g2.setColor(Color.blue);
g2.draw(outline);
```

Luego, se selecciona un área de recortado en el contexto gráfico usando el objeto `Shape` creado a partir de `getOutline`. La imagen `starry.gif`, que es una pintura famosa de Van Gogh, `The Starry Night`, se dibuja dentro de área de recortado que empieza en la esquina inferior izquierda del objeto `Rectangle`.

```
g2.setClip(outline);
g2.drawImage(img, r.x, r.y, r.width, r.height, this);
```

The Starry Night

Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.






[Starry.java](#) contiene el código completo de este programa. El applet



requiere el fichero de imagen [Starry.gif](#).

Ozito

Componer Gráficos

La clase AlphaComposite encapsula varios estilos de composición, que determinan cómo se dibujan los objeto solapados. Un AlphaComposite también puede tener un valor alpha que especifica el grado de transparencia: $\alpha = 1.0$ es totalmente opaco, $\alpha = 0.0$ es totalmente transparente. AlphaComposite soporta la mayoría de los estandares de composición como se muestra en la siguiente tabla.

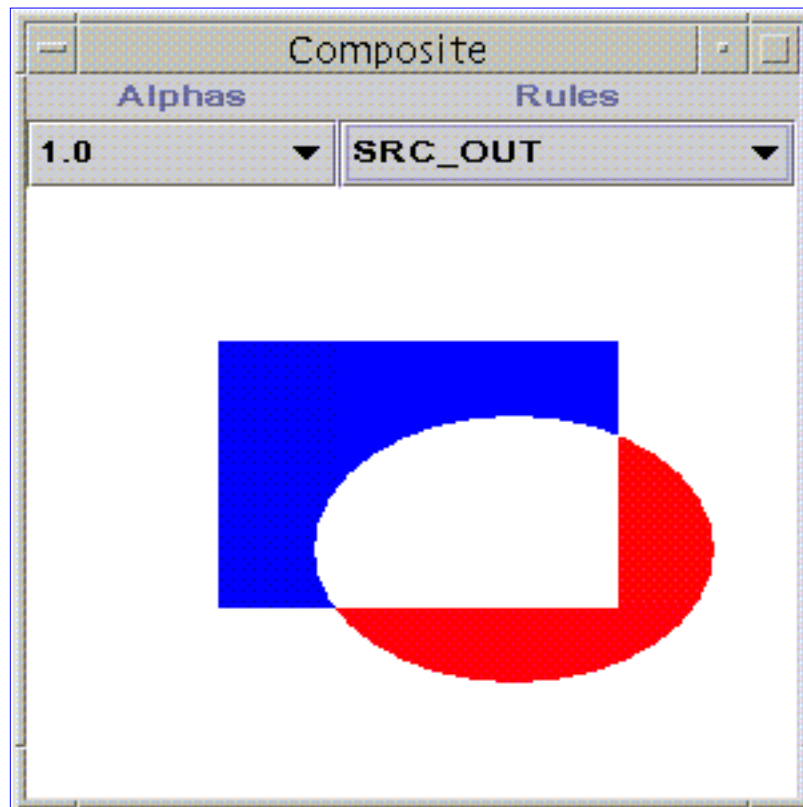
Source-over (SRC_OVER)  Source Destination	Si los pixels del objeto que está siendo renderizado (la fuente) tienen la misma posición que los pixels renderizadoa previamente (el destino), los pixels de la fuente se renderizan sobre los pixels del destino.
Source-in (SRC_IN)  Source-in	Si los pixels de la fuente y el destino se solapan, sólo se renderizarán los pixels que haya en el área solapada.
Source-out (SRC_OUT)  Source-out	Si los pixels de la fuente y el destino se solapan, sólo se renderizarán los pixels que haya fuera del área solapada. Los pixels que haya en el área solapada se borrarán.
Destination-over (DST_OVER)  Destination-over	Si los pixels de la fuente y del destino se solapan, sólo renderizarán los pixels de la fuente que haya fuera del área solapada. Los pixels que haya en el área solapada no se cambian.
Destination-in (DST_IN)  Destination-in	Si los pixels de la fuente y del destino se solapan, el alpha de la fuente se aplica a los pixels del área solapada del destino. Si el $\alpha = 1.0$, los pixels del área solapada no cambian; si alpha es 0.0 los pixels del área solapada se borrarán.

Destination-out (DST_OUT)  Destination-out	Si los pixels de la fuente y del destino se solapan, el alpha de la fuente se aplica a los pixels del área solapada del destino. Si el alpha = 1.0, los pixels del área solapada no cambian; si alpha es 0.0 los pixels del área solapada se borrarán.
Clear (CLEAR)  Clear	Si los pixels de la fuente y del destino se solapan, los pixels del área solapada se borrarán.

To change the compositing style used by Graphics2D, you create an AlphaComposite object and pass it into the setComposite method.

Ejemplo: Composite

Este programa ilustra los efectos de varias combinaciones de estilos de composición y valores de alpha.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Composite.java](#) contiene el código completo de este applet.

Se ha construido un nuevo objeto AlphaComposite al llamando a

AlphaComposite. getInstance y especifican las reglas de composición deseadas.

```
AlphaComposite ac =  
    AlphaComposite.getInstance(AlphaComposite.SRC);
```

Cuando se selecciona una regla de composición o un valor alpha, se llama de nuevo a AlphaComposite.getInstance, y el nuevo AlphaComposite se asigna a ac. El alpha selecciona se aplica al valor alpha de cada pixel y se le pasa un segundo parámetro a AlphaComposite.getInstance.

```
ac = AlphaComposite.getInstance(getRule(rule), alpha);
```

El atributo composite se modifica pasando el objeto AlphaComposite a Graphics 2D setComposite. Los objetos son renderizados dentro de un BufferedImage y más tarde son copiados en la pantalla, por eso el atributo composite es configurado al contexto Graphics2D para el BufferedImage:

```
BufferedImage buffImg = new BufferedImage(w, h,  
                                           BufferedImage.TYPE_INT_ARGB);  
Graphics2D gbi = buffImg.createGraphics();  
...  
gbi.setComposite(ac);
```

Controlar la Calidad de Dibujo

Podemos usar el atributo 'rendering hint' de Graphics2D para especificar si queremos que los objetos sean dibujados tan rápido como sea posible o si preferimos que se dibujen con la mayor calidad posible.

Para seleccionar o configurar el atributo 'rendering hint' en el contexto, Graphics2D podemos construir un objeto RenderingHints y pasarlo dentro de Graphics2D setRenderingHints. Si sólo queremos seleccionar un hint, podemos llamar a Graphics2D setRenderingHint y especificar la pareja clave-valor para el hint que queremos seleccionar. (Estas parejas están definidas en la clase RenderingHints.)

Nota: No todas las plataformas soportan la modificación del modo de dibujo, por eso el especificar los hints de dibujo no garantiza que sean utilizados.

RenderingHints soporta los siguientes tipos de hints:

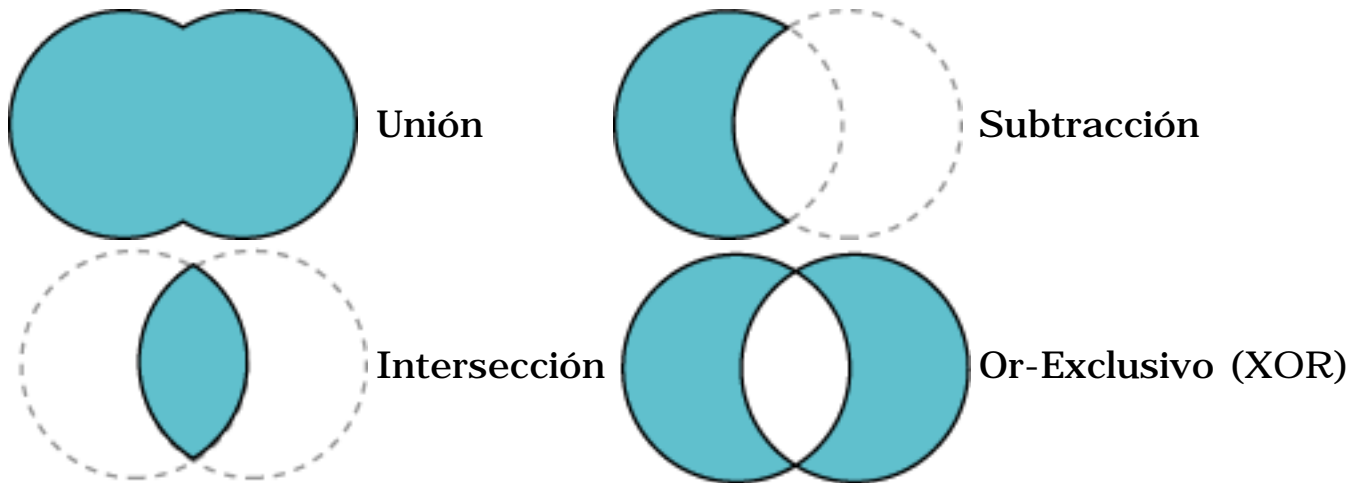
- Alpha interpolation--por defecto, calidad, o velocidad.
- Antialiasing--por defecto, on, u off
- Color rendering--por defecto, calidad, o velocidad
- Dithering--por defecto, activado o desactivado
- Fractional metrics--por defecto, on u off
- Interpolation--vecino más cercano, bilinear, o bicúbico
- Rendering--por defecto, calidad, o velocidad
- Text antialiasing--por defecto, on u off.

Cuando se selecciona un hint por defecto, se usa el sistema de dibujo por defecto de la plataforma.

Construir Formas Complejas desde Geométricos Primitivos

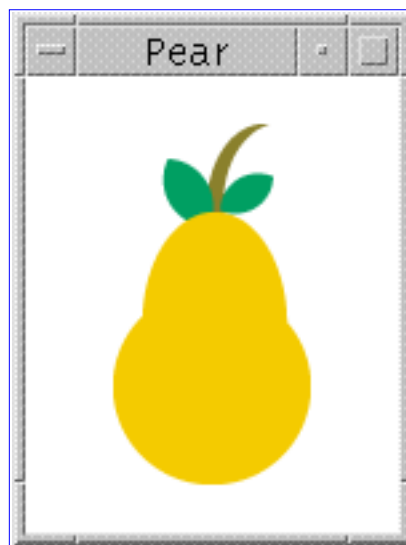
Construir un área geométrica (CAG) es el proceso de crear una nueva forma geométrica realizando operaciones con las ya existentes. En el API Java 2D un tipo especial de Shape llamado Area soporta operaciones booleanas. Podemos construir un Area desde cualquier Shape .

Areas soporta la siguientes operaciones booleanas:



Ejemplo: Areas

En este ejemplo, los objetos Area construyen una forma de pera partiendo de varias elipses.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[Pear.java](#) contiene el código completo de este applet.

Las hojas son creadas realizando una interesección entre dos círculos

solapados.

```
leaf = new Ellipse2D.Double();
...
leaf1 = new Area(leaf);
leaf2 = new Area(leaf);
...
leaf.setFrame(ew-16, eh-29, 15.0, 15.0);
leaf1 = new Area(leaf);
leaf.setFrame(ew-14, eh-47, 30.0, 30.0);
leaf2 = new Area(leaf);
leaf1.intersect(leaf2);
g2.fill(leaf1);
...
leaf.setFrame(ew+1, eh-29, 15.0, 15.0);
leaf1 = new Area(leaf);
leaf2.intersect(leaf1);
g2.fill(leaf2);
```

Los círculos solapados también se usan para cosntruir el rabo mediante una operación de sustracción.

```
stem = new Ellipse2D.Double();
...
stem.setFrame(ew, eh-42, 40.0, 40.0);
st1 = new Area(stem);
stem.setFrame(ew+3, eh-47, 50.0, 50.0);
st2 = new Area(stem);
st1.subtract(st2);
g2.fill(st1);
```

El cuerpo de la pera está construido mediante una operación unión de un círculo y un óvalo.

```
circle = new Ellipse2D.Double();
oval = new Ellipse2D.Double();
circ = new Area(circle);
ov = new Area(oval);
...
circle.setFrame(ew-25, eh, 50.0, 50.0);
oval.setFrame(ew-19, eh-20, 40.0, 70.0);
circ = new Area(circle);
ov = new Area(oval);
circ.add(ov);
g2.fill(circ);
```

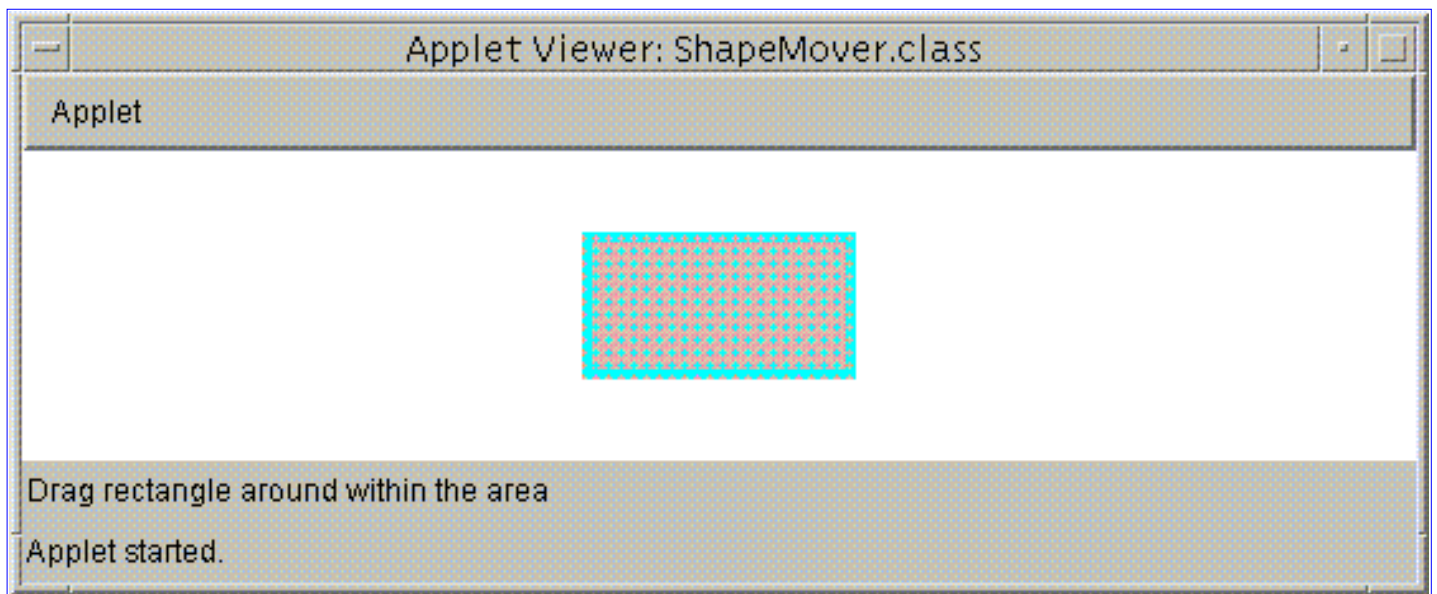
Soportar Interacción con el Usuario

Para permitir que el usuario interactúe con los graficos que hemos dibujado, necesitamos poder determinar cuando el usuario pulsa sobre uno de ellos. El método `Graphics2D hit` proporciona una forma para determinar fácilmente si ha ocurrido una pulsación de ratón sobre una `Shape` particular. De forma alternativa podemos obtener la posición del click de ratón y llamar a `contains` sobre la `Shape` para determinar si el click ocurrió dentro de los límites de la `Shape`.

Si estamos usando texto primitivo, podemos realizar una simple comprobación obteniendo la línea exterior de la `Shape` que corresponde al texto y luego llamando a `hit` o `contains` con esa `Shape`. El soporte de edición de texto requiere una comprobación mucho más sofisticada. Si queremos permitir que el usuario edite el texto, generalmente deberíamos usar uno de los componentes de texto editable de `Swing`. Si estamos trabajando con texto primitivo y estamos usando `TextLayout` para manejar la forma y posición del texto, también podemos usar `TextLayout` para realizar la comprobación para la edición de texto. Para más información puedes ver [Java 2D Programmer's Guide](#).

Ejemplo: ShapeMover

Este applet permite al usuario arrastrar la `Shape` por la ventana del applet. La `Shape` es redibujada en cada nueva posición del ratón para proporcionar información al usuario mientras la arrastra.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[ShapeMover.java](#) contiene el código completo de este applet.

Se llama al método `contains` para determinar si el cursor está dentro de los límites del rectángulo cuando se pulsa el botón. Si es así, se actualiza la posición del rectángulo.

```

public void mousePressed(MouseEvent e){
    last_x = rect.x - e.getX();
    last_y = rect.y - e.getY();
    if(rect.contains(e.getX(), e.getY())) updateLocation(e);
    ...

public void updateLocation(MouseEvent e){
    rect.setLocation(last_x + e.getX(), last_y + e.getY());
    ...
    repaint();
}

```

Podrías haber observado que redibujar la Shape en cada posición del ratón es muy lento, porque rectángulo relleno es renderizado cada vez que se mueve, Usando el doble buffer podemos eliminar este problema. Si estamos usando Swing, el dibujo usará doble buffer automáticamente; si no es así tendremos que cambiar todo el código de renderizado. El código para una versión swing de este programa es [SwingShapeMover.java](#). Para ejecutar la versión Swing, visita [SwingShapeMover](#).

Si no estamos usando Swing, el [Ejemplo: BufferedShapeMover](#) en la siguiente lección nos muestra cómo podemos implementar el doble buffer usando un BufferedImage. Podemos dibujar en un BufferedImage y luego copiar la imagen en la pantalla.

Ejemplo: HitTestSample

Esta aplicación ilustra la comprobación de pulsaciones dibujando el cursor por defecto siempre que el usuario pulse sobre el TextLayout, como se muestra en la siguiente figura.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

[HitTestSample.java](#) contiene el código completo de este applet.

El método mouseClicked usa TextLayout.hitTestChar para devolver un objeto java.awt.font.TextHitInfo que contiene la posición del click (el índice de inserción) en el objeto TextLayout.

La informacin devuelta por los métodos de TextLayout, getAscent, getDescent y getAdvance se utiliza paraa calcular la posición del origen del objeto TextLayout para que esté centrado tanto horizontal como verticalmente.

...

```
private Point2D computeLayoutOrigin() {
    Dimension size = getPreferredSize();
    Point2D.Float origin = new Point2D.Float();

    origin.x = (float) (size.width - textLayout.getAdvance()) / 2;
    origin.y =
        (float) (size.height - textLayout.getDescent()
            + textLayout.getAscent())/2;
    return origin;
}
```

...

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    setBackground(Color.white);
    Graphics2D graphics2D = (Graphics2D) g;
    Point2D origin = computeLayoutOrigin();
    graphics2D.translate(origin.getX(), origin.getY());

    // Draw textLayout.
    textLayout.draw(graphics2D, 0, 0);

    // Retrieve caret Shapes for insertionIndex.
    Shape[] carets = textLayout.getCaretShapes(insertionIndex);

    // Draw the carets.  carets[0] is the strong caret and
    // carets[1] is the weak caret.
    graphics2D.setColor(STRONG_CARET_COLOR);
    graphics2D.draw(carets[0]);
    if (carets[1] != null) {
        graphics2D.setColor(WEAK_CARET_COLOR);
        graphics2D.draw(carets[1]);
    }
}
```

...

```
private class HitTestMouseListener extends MouseAdapter {

    /**
```

```
* Compute the character position of the mouse click.
*/
public void mouseClicked(MouseEvent e) {

    Point2D origin = computeLayoutOrigin();

    // Compute the mouse click location relative to
    // textLayout's origin.
    float clickX = (float) (e.getX() - origin.getX());
    float clickY = (float) (e.getY() - origin.getY());

    // Get the character position of the mouse click.
    TextHitInfo currentHit = textLayout.hitTestChar(clickX, clickY);
    insertionIndex = currentHit.getInsertionIndex();

    // Repaint the Component so the new caret(s) will be displayed.
    hitPane.repaint();
}
```

Trabajar con Texto y Fuentes

Esta lección muestra como determinar las fuentes disponibles en nuestro sistema, crear fuentes con atributos particulares, derivar nuevas fuentes modificando atributos de alguna fuente existente, y posicionar múltiples líneas de texto dentro de un componente.

Estos tópicos se explican en las siguientes secciones:

Crear y Derivar Fuentes

Esta sección ilustra cómo usar el GraphicsEnvironment para determinar las fuentes disponibles en nuestro sistema, cómo crear un objeto Font y como cambiar los atributos de la fuente de una cadena de texto.

Dibujar Múltiples Líneas de Texto

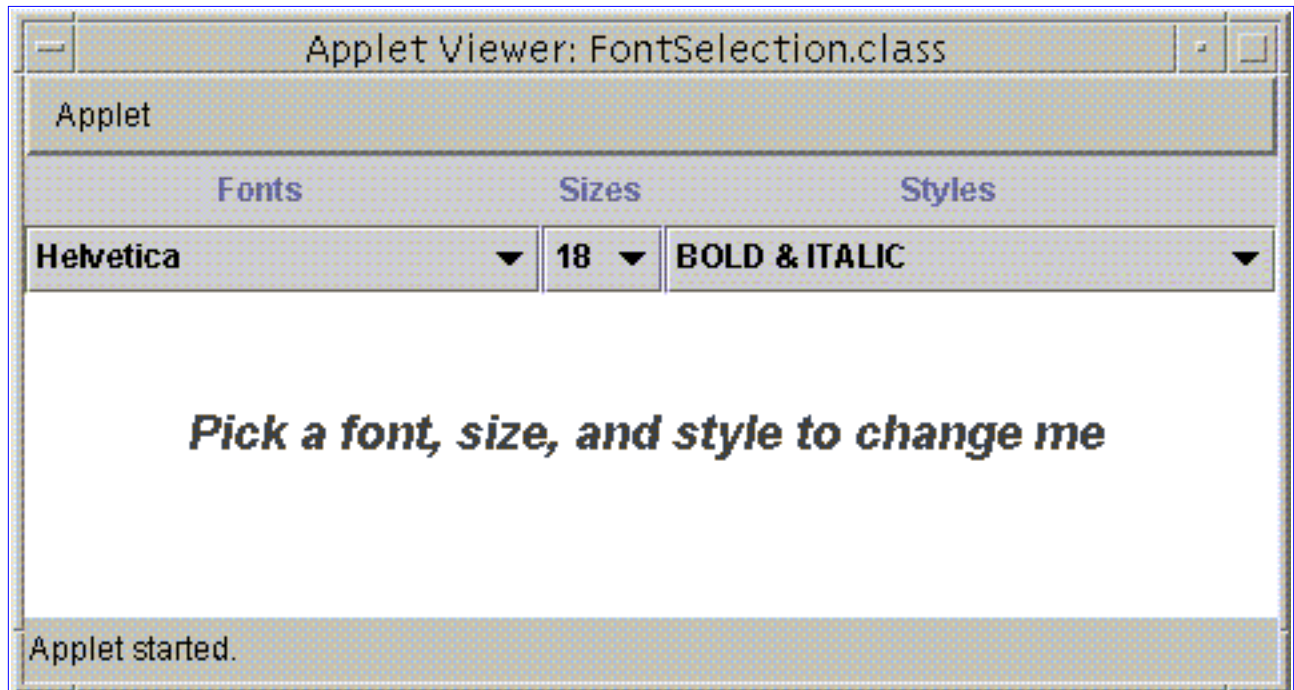
Esta sección nos muestra cómo posicionar u dibujar un párrafo de texto con estilo usando TextLayout y LineBreakMeasurer.

Crear y Derivar Fuentes

Podemos mostrar una cadena de texto con cualquier fuente disponible en nuestro sistema, en cualquier estilo y tamaño que elijamos. Para determinar las fuentes disponibles en nuestro sistema, podemos llamar al método `GraphicsEnvironment.getAvailableFontFamilyNames`. Este método devuelve un array de strings que contiene los nombres de familia de las fuentes disponibles. Cualquiera de las cadenas, junto con un argumento tamaño y otro de estilo, pueden ser usados para crear un nuevo objeto `Font`. Después de crear un objeto `Font`, podemos cambiar su nombre de familia, su tamaño o su estilo para crear una fuente personalizada.

Ejemplo: FontSelection

El siguiente applet nos permite cambiar la fuente, el tamaño y el estilo del texto dibujado.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

El código completo del applet está en [FontSelection.java](#).

El método `getAvailableFontFamilyNames` de `GraphicsEnvironment` devuelve los nombres de familia de todas las fuentes disponibles en nuestro sistema:

```
GraphicsEnvironment gEnv =  
    GraphicsEnvironment.getLocalGraphicsEnvironment();  
String envfonts[] = gEnv.getAvailableFontFamilyNames();
```

```
Vector vector = new Vector();
for ( int i = 1; i < envfonts.length; i++ ) {
    vector.addElement(envfonts[i]);
}
```

El objeto Font inicial s ecrea con estilo Font.PLAIN y tamaño 10. Los otros estilos disponibles son ITALIC, BOLD y BOLD+ITALIC.

```
Font thisFont;
...
```

```
thisFont = new Font("Arial", Font.PLAIN, 10);
```

Un nuevo Font se crea a partir de un nombre de fuente, un estilo y un tamaño.

```
public void changeFont(String f, int st, String si){
    Integer newSize = new Integer(si);
    int size = newSize.intValue();
    thisFont = new Font(f, st, size);
    repaint();
}
```

Para usar la misma familia de fuentes, pero cambiando uno o los dos atributos de estilo y tamaño, podemos llamar a uno de los métodos deriveFont.

Para controlar la fuente utilizada para renderizar texto, podemos seleccionar el atributo font en el contexto Graphics2D antes de dibujarlo. Este atributo se selecciona pasando un objeto Font al método setFont. En este ejemplo, el atributo font se configura para usar un objeto font recientemente construido y luego se dibuja la cadena de texto en el centro del componente usando la fuente especificada.

En el método paint, el atributo font del contexto Graphics2D se configura como el nuevo Font. La cadena se dibuja en el centro del componente con la nueva fuente.

```
g2.setFont(thisFont);
String change = "Pick a font, size, and style to change me";
FontMetrics metrics = g2.getFontMetrics();
int width = metrics.stringWidth( change );
int height = metrics.getHeight();
g2.drawString( change, w/2-width/2, h/2-height/2 );
```

Nota: debido al bug # [4155852](#), FontSelection podría no funcionar de forma apropiada con todos los nombres de fuentes devueltos por la llamada a getFontFamilyNames. La

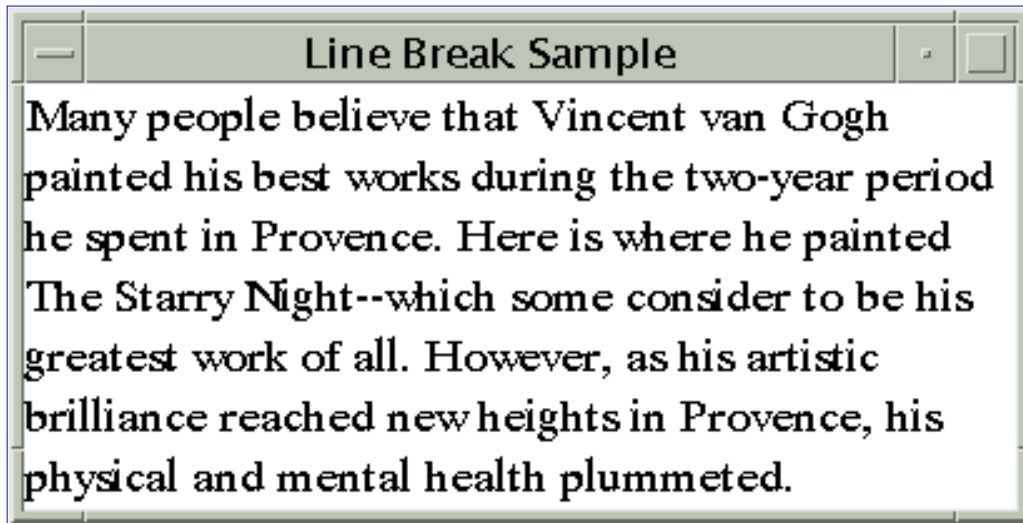
forma podría no corresponder con cambios en el tamaño o el estilo y el texto podría no mostrarse cuando se seleccionan algunos nombres de fuentes. En general, Courier y Helvetica funcionan bien. Mientras tanto, compruébalo periódicamente para ver si se han resuelto estos problemas.

Dibujar Múltiples Líneas de Texto

Si tenemos un párrafo de texto con estilo que queremos que quepa dentro de una anchura específica, podemos usar `LineBreakMeasurer`, que permite que el texto con estilo se rompa en líneas que caben dentro de un espacio visual. Como hemos aprendido en [Mostrar Gráficos con Graphics2D](#), un objeto `TextLayout` representa datos de caracteres con estilo, que no se pueden cambiar, pero también permite acceder a la información de distribución. Los métodos `getAscent` y `getDescent` de `TextLayout` devuelven información sobre la fuente usada para posicionar las líneas en el componente. El texto se almacena como un `AttributedCharacterIterator` para que los atributos de fuente y tamaño de punto puedan ser almacenados con el texto

Ejemplo: LineBreakSample

El siguiente applet posiciona un párrafo de texto con estulo dentro de un componente, usando `LineBreakMeasurer`, `TextLayout` y `AttributedCharacterIterator`.



Esta es una imagen del GUI del applet. Para ajecutar el appler, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

El código completo del applet está en [LineBreakSample.java](#).

El siguiente código crea un bucle con la cadena `vanGogh`. El inicio y final del bucle se recupera y se crea una nueva línea.

```
AttributedCharacterIterator paragraph = vanGogh.getIterator();
paragraphStart = paragraph.getBeginIndex();
paragraphEnd = paragraph.getEndIndex();

lineMeasurer = new LineBreakMeasurer(paragraph,
                                     new FontRenderContext(null, false, false));
```

El tamaño de la ventana se utiliza para determinar dónde se debería romper la línea y se crea un objeto `TextLayout` por cada línea del párrafo.

```

Dimension size = getSize();
float formatWidth = (float) size.width;
float drawPosY = 0;
lineMeasurer.setPosition(paragraphStart);

while (lineMeasurer.getPosition() < paragraphEnd) {
    TextLayout layout = lineMeasurer.nextLayout(formatWidth);

    // Move y-coordinate by the ascent of the layout.
    drawPosY += layout.getAscent();

    /* Compute pen x position.  If the paragraph is
       roght-to-left, we want to align the TextLayouts
       to the right edge of the panel.
    */
    float drawPosX;
    if (layout.isRIGHTToLEFT()) {
        drawPosX = 0;
    }
    else {
        drawPosX = formatWidth - layout.getAdvance();
    }

    // Draw the TextLayout at (drawPosX, drawPosY).
    layout.draw(graphics2D, drawPosX, drawPosY);

    // Move y-coordinate in preparation for next layout.
    drawPosY += layout.getDescent() + layout.getLeading();
}

```

Manipular y Mostrar Imágenes

Esta lección muestra cómo realizar operaciones de filtrado con `BufferedImage` y cómo usar un `BufferedImage` como un buffer fuera de pantalla.

Modo Inmediato con `BufferedImage`

Esta sección describe el modelo de modo inmediato implementado en el API Java 2D y explica cómo `BufferedImage` permite la manipulación de datos de imágenes.

Filtrado y `BufferedImage`

Esta sección muestra cómo usar las clases `BufferedImageOp` para realizar operaciones de filtrado sobre `BufferedImage`.

Usar un `BufferedImage` para doble buffer

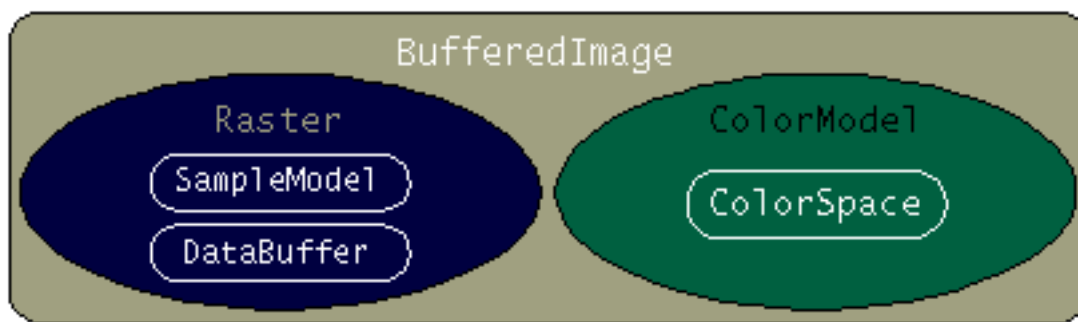
Esta sección nos enseña cómo usar un `BufferedImage` como un buffer fuera de pantalla para aumentar el rendimiento de las imágenes.

Modo Inmediato con BufferedImage

El modelo de imágenes modo inmediato permite manipular y mostrar imágenes de pixels mapeados cuyos datos están almacenados en memoria. Podemos acceder a los datos de la imagen en una gran variedad de formatos y usar varios tipos de operaciones de filtrado para manipular los datos.

BufferedImage es la clase clave del API del modo-inmediato. Esta clase maneja una imagen en memoria y proporciona métodos para almacenar, interpretar y dibujar cada dato de pixel. Un BufferedImage puede ser renderizado en un contexto Graphics o en un contexto Graphics2D.

Un BufferedImage es esencialmente un Image un buffer de datos accesible. Un BufferedImage tiene un ColorModel y un Raster de los datos de la imagen.



El ColorModel proporciona una interpretación de color de los datos de los pixels de la imagen. El Raster representa las coordenadas rectangulares de la imagen, mantiene los datos de la imagen en memoria, y proporciona un mecanismo para crear múltiples subimágenes de un sólo buffer de imagen. El Raster también proporciona métodos para acceder a pixels específicos dentro de la imagen. Para más información sobre como manipular directamente los datos de los pixels y escribir filtros para objetos BufferedImage, puedes ver el capítulo Imaging de [Java 2D Programmer's Guide](#).

Filtrar un BufferedImage

El API Java 2D define varias operaciones de filtrado para objeto `BufferedImage`. Cada operación de proceso de imágenes está incluida en una clase que implementa el interface `BufferedImageOp`. La manipulación de imágenes se realiza en el método `filter`. La clase `BufferedImageOp` en el API Java 2D soporta

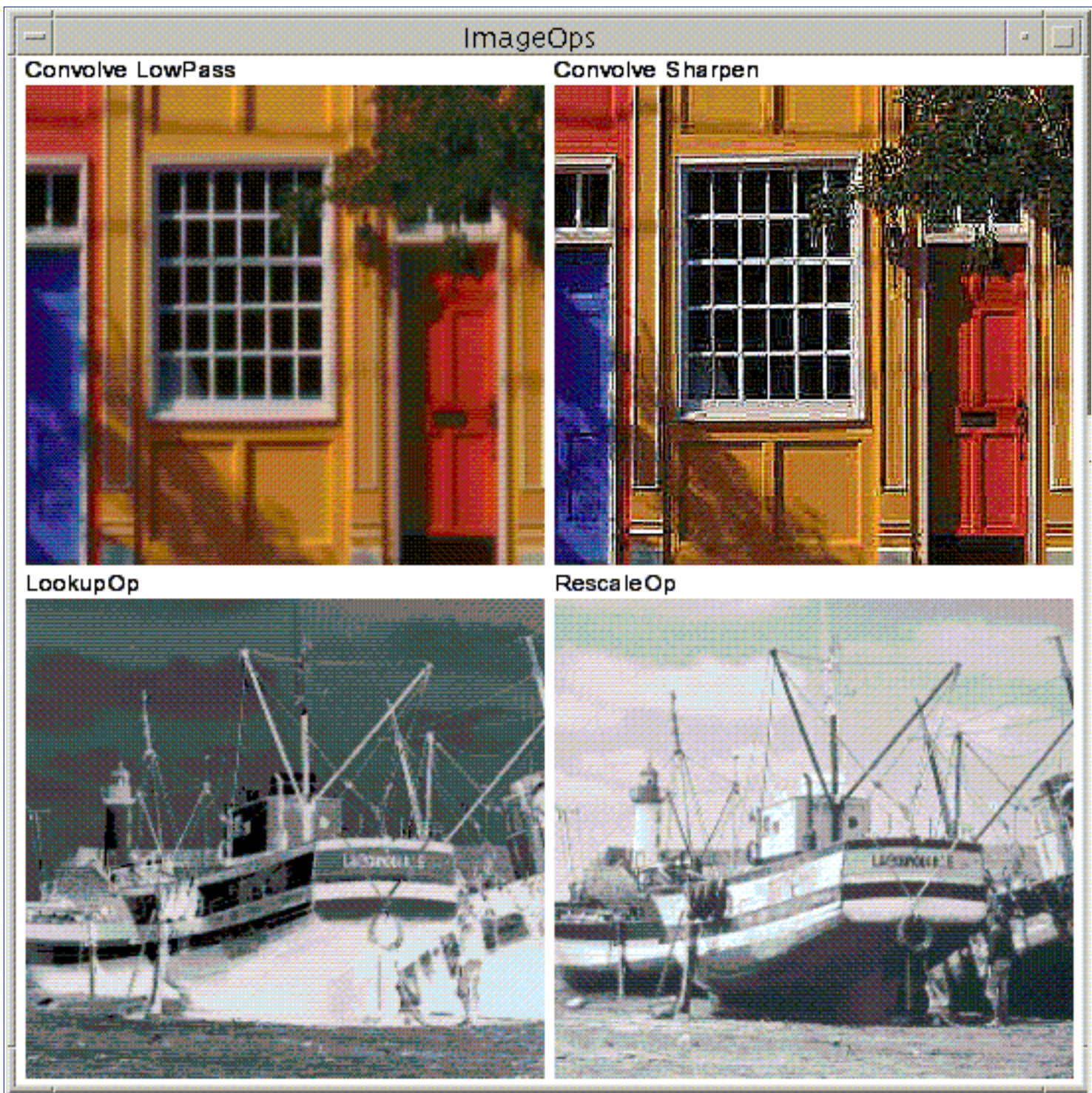
- Transformación afin.
- Escalado.
- Modificación de Aspecto.
- Combinación Linear de Bandas.
- Conversión de color.
- Convolución.

Para filtrar un `BufferedImage` usando una de las clases de operación de imagen, debemos

1. Construir un ejemplar de una de las clases `BufferedImageOp`: `AffineTransformOp`, `BandCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, o `RescaleOp`.
2. Llamar al método de operación `filter`, pasando en el `BufferedImage` que queremos filtrar y el `BufferedImage` donde queremos almacenar el resultado.

Ejemplo: ImageOps

El siguiente applet ilustra el uso de cuatro operaciones de filtrado de imágenes: `low-pass`, `sharpen`, `lookup`, y `rescale`.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

El código completo del applet está en [ImageOps.java](#). El applet usa estos dos ficheros de imagen: [bld.jpg](#) y [boat.gif](#).

El filtro sharpen se realiza usando un ConvolveOp. Convolución es el proceso de hacer más pesado el valor de cada pixel en una imagen con los valores de los pixels vecinos. La mayoría de los algoritmos de filtrado espacial están basados en las operaciones de convolución.

Para construir y aplicar este tipo de filtrado al BufferedImage, este

ejemplo usa un código similar al del siguiente fragmento.

```
public static final float[] SHARPEN3x3 = {  
    0.f, -1.f, 0.f,  
    -1.f, 5.0f, -1.f,  
    0.f, -1.f, 0.f};  
  
BufferedImage dstbimg = new  
    BufferedImage(iw,ih,BufferedImage.TYPE_INT_RGB);  
Kernel kernel = new Kernel(3,3,SHARPEN3x3);  
ConvolveOp cop = new ConvolveOp(kernel,  
    ConvolveOp.EDGE_NO_OP,  
    null);  
  
cop.filter(srcbimg,dstbimg);
```

El objeto Kernel define matemáticamente cómo se ve afectada la salida de cada pixels en su área inmediata. La definición del Kernel determine el resultado del filtro. Para más información sobre cómo trabaja el kernel con ConvolveOp puedes ver la sección 'Image Processing and Enhancement' en [Java 2D Programmer's Guide](#)

Usar un BufferedImage para Doble Buffer

Cuando un gráfico es complejo o se usa repetidamente, podemos reducir el tiempo que tarda en mostrarse renderizándolo primero en un buffer fuera de pantalla y luego copiando el buffer en la pantalla. Esta técnica, llamada doble buffer, se usa frecuentemente para animaciones.

Nota: cuando dibujamos sobre un componente Swing, éste utiliza automáticamente el doble buffer.

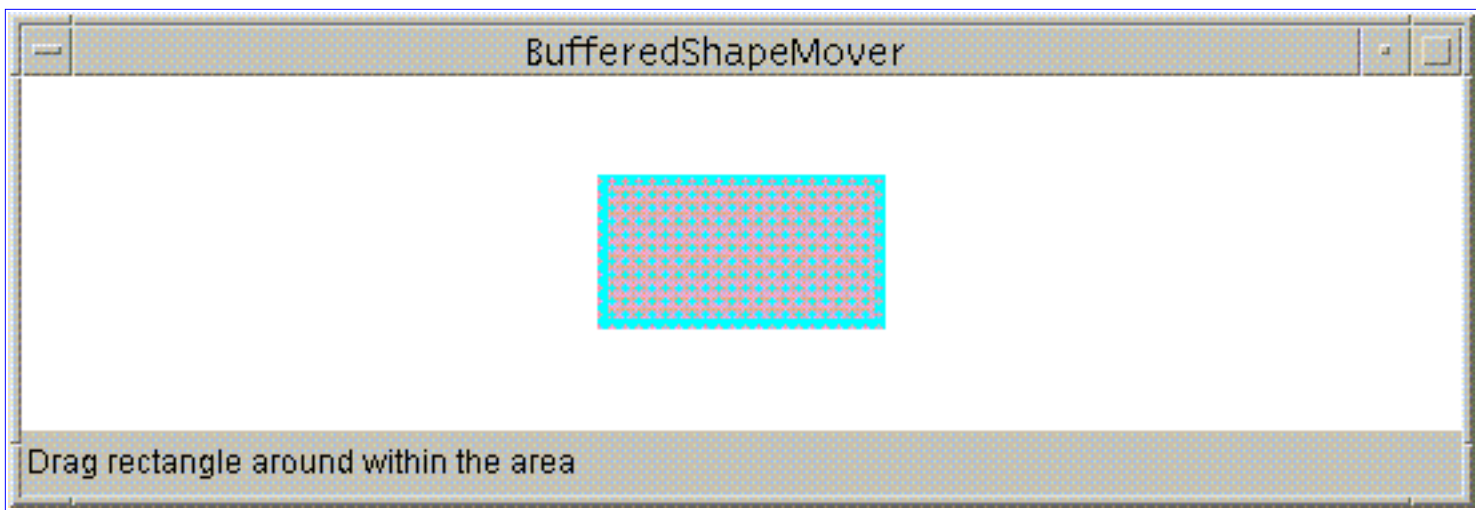
Un BufferedImage puede usarse fácilmente como un buffer fuera de pantalla. Para crear un BufferedImage cuyo espacio, color, profundidad y distribución de pixels corresponden exactamente la ventana en la que son dibujados, se llama al método Component createImage. Si necesitamos control sobre tipo de la imagen fuera de la pantalla, la transparencia, podemos construir directamente un objeto BufferedImage y usarlo como un buffer.

Para dibujar dentro de una imagen almacenada, se llama al método BufferedImage.createGraphics para obtener el objeto Graphics2D; luego se llama a los métodos de dibujo apropiados del Graphics2D. Todo el API de dibujo de Java 2D puede usarse cuando se dibuja sobre un BufferedImage que está siendo utilizado como un buffer fuera de pantalla.

Cuando estemos listos para copiar el BufferedImage en la pantalla, simplemente llamamos al método drawImage sobre el Graphics2D de nuestro componente y pasarlo en BufferedImage.

Ejemplo: BufferedShapeMover

El siguiente applet permite al usuario arrastrar un rectángulo sobre la ventana del applet. En lugar de dibujar el rectángulo en cada posición del cursor, para proporcionar información al usuario, se usa un BufferedImage como buffer fuera de la pantalla. Cuando se arrastra el rectángulo, es renderizado dentro del BufferedImage en cada nueva posición y BufferedImage se copia en la pantalla.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

El código completo del applet está en [BufferedShapeMover.java](#).

Aquí está el código usado para renderizar en el BufferedImage y mostrar la imagen en la pantalla:

```
public void updateLocation(MouseEvent e){
    rect.setLocation(last_x + e.getX(),
                     last_y + e.getY());

    ...
    repaint();
    ...
    // In the update method...
    if(firstTime) {
        Dimension dim = getSize();
        int w = dim.width;
        int h = dim.height;
        area = new Rectangle(dim);
        bi = (BufferedImage)createImage(w, h);
        big = bi.createGraphics();
        rect.setLocation(w/2-50, h/2-25);
        big.setStroke(new BasicStroke(8.0f));
        firstTime = false;
    }

    // Clears the rectangle that was previously drawn.
    big.setColor(Color.white);
    big.clearRect(0, 0, area.width, area.height);

    // Draws and fills the newly positioned rectangle
    // to the buffer.
    big.setPaint(strokePolka);
```

```
big.draw(rect);  
big.setPaint(fillPolka);  
big.fill(rect);  
  
// Draws the buffered image to the screen.  
g2.drawImage(bi, 0, 0, this);  
}
```

Imprimir

Esta lección nos enseña cómo usar el API `Print` de Java para imprimir desde nuestras aplicaciones Java. Aprenderemos cómo dibujar los contenidos de nuestros componentes a la impresora en lugar de al dispositivo de pantalla y como componer documentos de múltiples páginas. Esta lección asume que has leído la primera lección de esta ruta, [Introducción al API 2D de Java](#), y que estás familiarizado con el uso del contexto de dibujo `Graphics2D`.

Introducción a la Impresión en Java

Esta sección nos ofrece una introducción al soporte de impresión del AWT y el API 2D de Java y describe el modelo de impresión de Java.

Imprimir los Contenidos de un Componente

Esta sección nos enseña cómo crear un `PrinterJob` y cómo usar un `Printable` para imprimir los contenidos de un componente.

Mostrar el diálogo 'Page Setup'

Esta sección describe el diálogo estándar de configuración de página y nos enseña cómo usarla para permitir que el usuario configure un trabajo de impresión.

Imprimir una Colección de Páginas

Esta sección enseña cómo configurar un `Book` para imprimir una colección de páginas que no tienen ni el mismo tamaño ni orientación.

Introducción a la Impresión en Java

El sistema controla totalmente el proceso de impresión, al igual que controla cómo y cuándo pueden dibujar los programas, Nuestras aplicaciones proporcionan información sobre el documento a imprimir, y el sistema de impresión determina cuando necesita renderizar cada página.

Este modelo de impresión permite soportar una amplio rango de impresoras y de sistemas. Incluso permite al usuario imprimir en una impresora de bitmaps desde un ordenador que no tiene suficiente memoria o espacio en disco para contener el mapa de bits de una página completa. En esta situación el sistema de impresión le pedirá a nuestra aplicación que renderize la página de forma repetida para que pueda ser imprimida como una serie de pequeñas imágenes.

Para soportar impresión, una aplicación necesita realizar dos tareas:

- Job control--manejar el trabajo de impresión
- Imaging--renderizar las páginas a imprimir

Job Control

Aunque el sistema controla todo el proceso de impresión, nuestra aplicación tiene que obtener la oportunidad de configurar un `PrinterJob`. El `PrinterJob`, el punto clave del control del proceso de impresión, almacena las propiedades del trabajo de impresión, controla la visión de los diálogos de impresión y se usa para inicializar la impresión.

Para dirigir el `PrinterJob` a través del proceso de impresión, nuestra aplicación necesita

1. Obtener un `PrinterJob` llamando a `PrinterJob.getPrinterJob`
2. Decirle al `PrinterJob` dónde está el código de dibujo llamando a `setPrintable` o `setPageable`
3. Si se desea, mostrar los diálogos de configuración de página e impresión llamando a `pageDialog` y `printDialog`
4. Iniciar la impresión llamando a `print`

El dibujo de páginas está controlado por el sistema de impresión a través de llamadas al código de imágenes de la aplicación.

Imaging

Nuestra aplicación debe poder renderizar cualquier página cuando el sistema de impresión lo pida. Este código de renderizado está contenido en el método `print` de un `page painter`--una clase que implementa el interface `Printable`. Nosotros implementamos `print` para renderizar el contenido de la página usando un `Graphics` o un `Graphics2D`. Podemos

usar un único 'page painter' para renderizar todas las páginas de un 'print job' o diferentes 'page painters' para los diferentes tipos de páginas. Cuando el sistema de impresión necesita renderizar una página, llama al método print del 'page painter' apropiado.

Cuando se usa un único 'page painter', al 'print job' se le llama un printable job. Usar un 'printable job' es la forma más sencilla de soportar impresión. Las operaciones de impresión más complejas que usan múltiples 'page painters' son conocidas como pageable jobs. En un 'pageable job' se usa un ejemplar de una clase que implemente el interface Pageable para manejar los 'page painters'.

Printable Jobs

En un 'printable job' todas las páginas usan el mismo 'page painter' y PageFormat, que define el tamaño y orientación de la página a imprimir. Se le pide al 'page painter' que renderice cada página en orden indexado, empezando en la página con índice 0. Al 'page painter' se le podría pedir que renderizará una página varias veces antes de pedir la siguiente página. Por ejemplo, si un usuario imprimir las páginas 2 y 3 de un documento, se le pide al 'page painter' que renderice las páginas con índices 0, 1 y 2 incluso aunque las dos primeras no sean impresas.

Si se presenta un diálogo de impresión, no se mostrará el número de páginas, porque esa información no está disponible para el sistema de impresión. El 'page painter' informa al sistema de impresión cuando se alcanza el final del documento.

Pageable Jobs

Los 'pageable jobs' son útiles si nuestra aplicación construye una representación explícita de un documento, página por página. En un 'pageable job' diferentes páginas pueden usar diferentes 'page painters' y PageFormats. El sistema de impresión puede pedir a los 'page painters' que rendericen las páginas en cualquier orden, incluso puede saltarse algunas. Por ejemplo si un usuario imprimir las páginas 2 y 3 de un documento, sólo se le pedirá al 'page painter' que renderice las páginas con los índices 1 y 2.

Los distintos 'page painters' de un 'pageable job' son coordinados por una clase que implementa el interface Pageable, como un Book. Un Book representa una colección de página que pueden usar diferentes 'page painter' y que

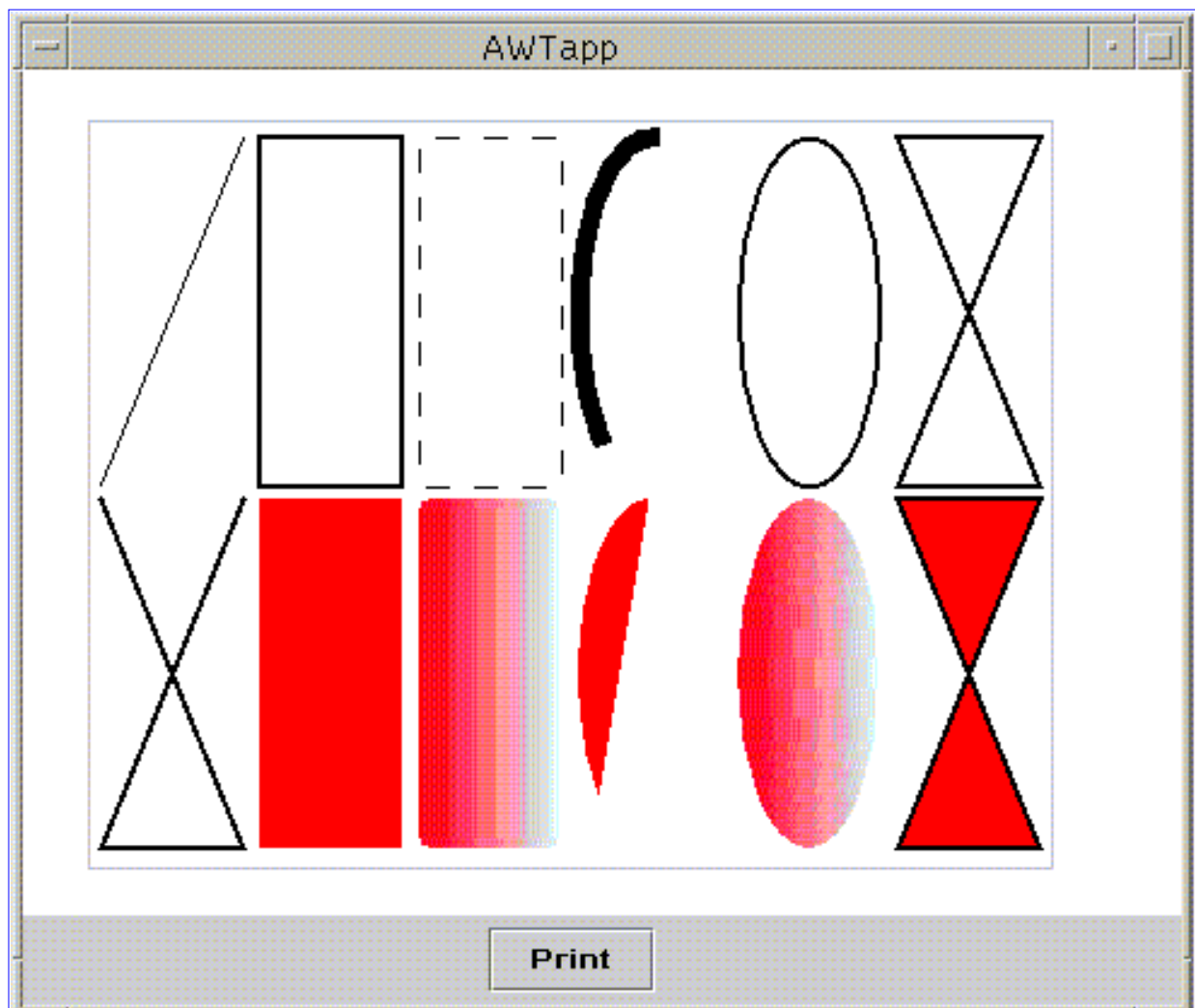
pueden variar en tamaño y orientación, También podemos usar nuestra propia implementación del interface Pageable si Book no cumple con las necesidades de nuestra aplicación.

Imprimir los Contenidos de un Componente

Cualquier cosa que podamos dibujar en la pantalla también puede ser imprimida. Podemos fácilmente usar un 'printable job' para imprimir el contenido de un componente.

Ejemplo: ShapesPrint

En este ejmplo usamos el mismo código de dibujo para mostrar e imprimir los contenidos de un componente. Cuando el usuario pulsa sobre el botón print, se crea un 'print job' y se llama a printDialog para mostrar el diálogo de impresión. Si el usuario continúa con el trabajo, el proceso de impresión se inicia, y el sistema de impresión llama a print cuando sea necesario renderizar el trabajo a la impresora.



Esta figura ha sido reducida para que quepa en la página.
Pulsa sobre la imagen para verla a su tamaño natural.

ShapesPrint es el 'page painter'. Sus método print llama a drawShapes para realizar el dibujo del 'print job'. (También se llama al método drawShapes por parte de paintComponent para dibujar en la

pantalla.)

```
public class ShapesPrint extends JPanel
    implements Printable, ActionListener {
    ...
    public int print(Graphics g, PageFormat pf, int pi)
        throws PrinterException {
        if (pi >= 1) {
            return Printable.NO_SUCH_PAGE;
        }
        drawShapes((Graphics2D) g);
        return Printable.PAGE_EXISTS;
    }
    ...
    public void drawShapes(Graphics2D g2) {
        Dimension d = getSize();
        int gridWidth = 400/6;
        int gridHeight = 300/2;
        int rowspacing = 5;
        int columnspacing = 7;
        int rectWidth = gridWidth - columnspacing;
        int rectHeight = gridHeight - rowspacing;
        ...

        int x = 85;
        int y = 87;
        ...
        g2.draw(new Rectangle2D.Double(x,y,rectWidth,rectHeight));
        ...
    }
}
```

El código de control del trabajo está en el método ShapesPrint actionPerformed.

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() instanceof JButton) {
        PrinterJob printJob = PrinterJob.getPrinterJob();
        printJob.setPrintable(this);
        if (printJob.printDialog()) {
            try {
                printJob.print();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Puedes encontrar el código completo de este programa en [ShapesPrint.java](#).

Ozito

Mostrar el Diálogo de configuración de Página

Podemos permitir que el usuario especifique las características de la página, como el tamaño del papel y la orientación, mostrando el diálogo de Configuración de Página. La información de la página se almacena en un objeto `PageFormat`. Al igual que el diálogo de Impresión, el diálogo de Configuración de Página se muestra llamando un método sobre el objeto `PrinterJob`, `pageDialog`.

El diálogo de Configuración de Página se inicializa usando el `PageFormat` pasado al método `pageDialog`. Si el usuario pulsa sobre el botón OK del diálogo, se clona `PageFormat`, alterado para reflejar las selecciones del usuario, y luego retorna. Si el usuario cancela el diálogo, `pageDialog` devuelve el original, sin modificar `PageFormat`.

`ShapesPrint` podría fácilmente ser modificado para mostrar un diálogo de configuración de página añadiendo una llamada a `pageDialog` después de obtener el `PrinterJob`.

```
// Get a PrinterJob
PrinterJob job = PrinterJob.getPrinterJob();
// Ask user for page format (e.g., portrait/landscape)
PageFormat pf = job.pageDialog(job.defaultPage());
```

Imprimir una Colección de Páginas

Cuando se necesite más control sobre las páginas individuales en un trabajo de impresión, podemos usar un 'pageable job' en lugar de un 'printable job'. La forma más sencilla de manejar un 'pageable job' es utilizar la clase Book, que representa una colección de páginas.

Ejemplo: SimpleBook

El programa SimpleBook usa un Book para manejar dos 'page painters': PaintCover se utiliza para la cubierta, y PaintContent para la página de contenido. La cubierta se imprime en modo apaisado, mientras que el contenido se imprime en modo vertical.

Una vez creado el Book, las páginas se añaden con el método append. Cuando se añade una página a un Book, se necesita especificar el Printable y el PageFormat para usar con cada página.

```
// In the program's job control code...
// Get a PrinterJob
PrinterJob job = PrinterJob.getPrinterJob();

// Create a landscape page format
PageFormat landscape = job.defaultPage();
landscape.setOrientation(PageFormat.LANDSCAPE);

// Set up a book
Book bk = new Book();
bk.append(new PaintCover(), job.defaultPage());
bk.append(new PaintContent(), landscape);

// Pass the book to the PrinterJob
job.setPageable(bk);
```

Se llama al método setPageable sobre PrinterJob para decirle al sistema de control que utilice el Book para localizar el código de dibujo adecuado.

Puedes encontrar el programa completo en [SimpleBook.java](#).

Resolver problemas comunes con Gráficos 2D

Problema: Puedo ejecutar applets Java2D con appletviewer, pero no funcionan con mi navegador. La consola Java del navegador dice: `defn not found for java/awt/Graphics2D`.

- Necesitas descargar el Java Plugin 1.2 para ejecutar Swing y applets 2D en un navegador. Puedes descargar el plugin aquí:
<http://java.sun.com/products/plugin/index.html>

Necesitarás ajustar tus ficheros HTML para apuntar hacia el pugin. Aquí tienes una página con varios ejemplos, incluido un ejemplo 2D en la parte inferior:

<http://java.sun.com/products/plugin/1.2/demos/applets.html>

Problema: ¿Cómo puedo escribir sobre una imagen anterior? Nuestro problema es que nuestro applet muestrea una imagen de un mapa, pero cuando se dibuja una línea sobre el mapa, la línea sobrescribe el mapa.

- Deberías intentar dibujar tu imagen dentro de un `BufferedImage`. Luego, dibuja el `BufferedImage` dentro de un contexto `Graphics2D` y luego dibuje la línea sobre el contexto `Graphics2D` un ejemplo de código está en

[Map_Line.java](#)

Sólo debes sustituir el nombre de tu imagen por `images/bld.jpg`.

Problema: ¿Cómo creo un `BufferedImage` desde un fichero gif o jpeg?

- Para crear un `BufferedImage` desde un gif o jpeg from a gif or jpeg, debes cargar tu fichero gif o jpeg en un objeto `Image` y luego dibujar el `Image` en el objeto `BufferedImage`. El siguiente fragmento de código ilustra esto:

```
Image img = getImage("picture.gif");
int width = img.getWidth(this);
int height = img.getHeight(this);
```

```
BufferedImage bi = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
Graphics2D biContext = bi.createGraphics();
biContext.drawImage(img, 0, 0, null);
```

`getImage` es un método `Applet`. Si tiene una aplicación, puedes usar:

```
Image img = Toolkit.getDefaultToolkit().getImage("picture.gif");
```

`BufferedImage.TYPE_INT_RGB` es uno de los muchos tipos de `BufferedImage`. Para más información, puedes ver:

<http://java.sun.com/products/java-media/2D/forDevelopers/2Dapi/java/awt/image/BufferedImage.html>

Necesitas crear un contexto `Graphics2D` para el `BufferedImage` usando el método `createGraphics`. Luego, puedes usar el método `drawImage` de la clase `Graphics2D` para dibujar la imagen dentro del buffer.

Problema: No puedo compilar el código fuente de `StrokeAndFill.java` y `Transform.java` con `jdk1.2beta4`.

- La implementación de `TextLayout.getOutline` se cambió entre la `beta4` y el JDK actual. La nueva implementación sólo toma un `AffineTransform` como argumento. Necesitas descargar el nuevo JDK para ejecutar el ejemplo.

Problema: ¿Existe alguna forma de especificar una fórmula para una línea y dibujar un gráfico de acuerdo a ella?

- Usar los segmentos de línea podría ser la forma más sencilla. Pudes representar los segmentos de línea rellenando un GeneralPath con ellos, o implementando Shape y PathIterator y leyendo los segmentos de línea 'bajo demanda' para guardar el almacenamiento intermedio del objeto GeneralPath. Observa que podrías analizar tu fórmula para determinar si corresponde con curvas cúbicas o cuadráticas.

Problema: ¿Cómo puedo añadir texto a un campo gráfico en una cierta posición?

- En el JDK 1.2 se añadió una clase llamada Graphics2D (ahora llamado Java 2 SDK). Esta clase descende de Graphics. Hay dos métodos drawString en Graphics2D que puedes utilizar. Si quieres rotar el texto, deberías usar Graphics2D en vez de Graphics por lo que podrás realizar rotaciones y otras transformaciones en tu contexto Graphics2D.

El ejemplo Transform en el tutorial de 2D no usa drawString para dibujar texto. Lo que sucede es que se crea un TextLayout desde la cadena "Text." El TextLayout nos permite crear un objeto Shape a partir del String obteniendo su forma exterior. Introducimos esta Shape en el array de shapes, junto con las formas del rectángulo y de la elipse. Cuando dibujamos o rellenamos el Shape seleccionado del array de shapes, llamamos a g2.draw(Shape) o a g2.fill(Shape).

Podrías usar drawString para dibujar el texto en el contexto Graphics2D y luego llamar a g2.rotate (ángulo de rotación). Esto rotará todo lo que hayamos introducido dentro del contexto Graphics2D. Por eso, podríamos resetear el contexto g2 cada vez que querramos transformar una imagen particular o una parte del texto en el contexto de forma separada de otras cadenas que hayan sido renderizadas en el contexto g2.

Problema: He leído su comentario en la parte inferior de [Crear y Derivar fuentes](#) sobre el bug 4155852. Este bug ha sido cerrado sin ninguna acción. ¿Es cierto que no se puede aplicar un estilo a una fuente como Arial?

El problema es que la correspondencia fuente-a-estilo no funciona de forma apropiada para fuentes físicas (como Arial o Palatino). Sólo se pueden aplicar estilos a las fuentes lógicas en este momento (como Dialog o SansSerif).

Como atajo hasta que se corrija el bug, podrías hacer lo siguientes:

```
Font f = new Font("Palatino Bold", Font.PLAIN, 12);
```

en lugar de :

```
Font f = new Font("Palatino", Font.BOLD, 12);
```


Conectividad y Seguridad del Cliente

El entorno Java es altamente considerado en parte por su capacidad para escribir programas que utilizan e interactúan con los recursos de Internet y la World Wide Web. De hecho, los navegadores que soportan Java utilizan esta capacidad del entorno Java hasta el extremo de transportar y ejecutar applets a través de la red.

Indice de Contenidos:

- Introducción al Trabajo en Red
 - [Trabajo en Red Básico](#)
 - [Lo qué podrías conocer ya sobre el trabajo en Red en Java](#)
- Trabajar con URLs
 - [¿Qué es una URL?](#)
 - [Crear una URL](#)
 - [Compartir una URL](#)
 - [Leer Directamente desde una URL](#)
 - [Conectar con una URL](#)
 - [Leer y Escribir utilizando una Conexión URL](#)
- [Todo sobre los Sockets](#)
 - [¿Qué es un Socket?](#)
 - [Leer y Escribir utilizando un Socket](#)
 - [Escribir el lado del servidor de un Socket](#)
- Todos sobre los Datagramas
 - [¿Qué es un Datagrama?](#)
 - [Escribir un Datagrama Cliente y Servidor](#)
- [Proporcionar su propio Controlador de Seguridad](#)
 - [Introducción a los Controladores de Seguridad](#)
 - [Escribir un Controlador de Seguridad](#)
 - [Instalar su Controlador de Seguridad](#)
 - [Decidir qué métodos sobrescribir del SecurityManager](#)
- **Cambios en el** [JDK 1.1](#)

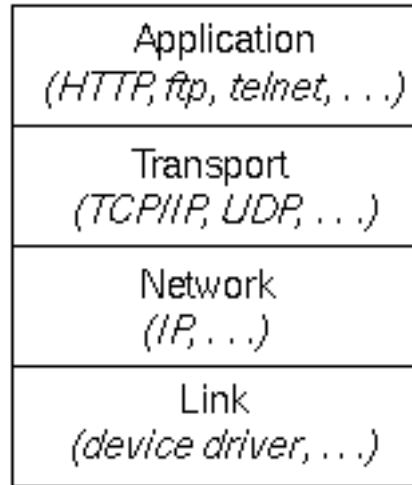
Consideraciones de Seguridad: Observa que la comunicación a través de la red está sujeta a la aprobación del controlador de seguridad actual. Los programas de ejemplo contenidos en las lecciones sobre URLs, sockets y Datagramas de esta ruta son aplicaciones solitarias, que por defecto no tienen controlador de

seguridad. Si quieres convertir estas aplicaciones en applets podría ser posible que no se comunicarán a través de la red, dependiendo del navegador o visualizados en el que se esté ejecutando. Puedes ver [Entender las Capacidades y las Restricciones de un Applet](#) para más información sobre las restricciones de seguridad de los applets

Ozito

Trabajo en Red Básico

Los ordenadores que se ejecutan en Internet comunican unos con otros utilizando los protocolos TCP y UDP, que son protocolos de 4 capas.



Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación. Típicamente, no se necesita trabajar con las capas TCP y UDP -- en su lugar se puede utilizar las clases del paquete java.net. Estas clases proporcionan comunicación de red independiente del sistema. Sin embargo, necesitas entender la diferencia entre TCP y UDP para decidir que clases Java deberían utilizar tus programas.

Cuando dos aplicación se quieren comunicar una con otra de forma fiable, establecen una conexión y se envían datos a través de la conexión. Esto es parecido a hacer una llamada de teléfono --se establece una comunicación cuando se marca el número de teléfono y la otra persona responde. Se envían y reciben datos cuando se habla por el teléfono y se escucha lo que le dice la otra persona. Al igual que la compañía telefónica, TCP garantiza que los datos enviados por una parte de la conexión realmente llegan a la otra parte y en el mismo orden en el que fueron enviados (de otra forma daría un error).

Definición: TCP es un protocolo basado en conexión que proporciona un flujo fiable de datos entre dos ordenadores.

Las aplicaciones que requieren fiabilidad, canales punto a punto para comunicarse, utilizan TCP para ello. Hyper Text Transfer Protocol (HTTP), File Transfer Protocol (ftp), y Telnet (telnet) son ejemplos de aplicaciones que requieren un canal de comunicación fiable. El orden en que los datos son enviados y recibidos a través de la Red es crítico para el éxito de estas aplicaciones -- cuando se utiliza HTTP para leer desde una URL, los datos deben recibirse en el mismo orden en que fueron enviados, de otra forma tendrás un fichero HTML revuelto, un fichero Zip corrupto o cualquier otra información no válida.

Para muchas aplicaciones esta garantía de fiabilidad es crítica para el éxito de la transferencia de información desde un punto de la conexión al otro. Sin embargo, otras formas de comunicación no necesitan esta comunicación tan estricta y de hecho lo que hace es estorbar porque la conexión fiable anula el servicio.

Considera, por ejemplo, un servicio de hora que envía la hora actual a sus clientes cuando estos lo piden. Si el cliente pierde un paquete, ¿tiene sentido volver a enviar el paquete? No porque la hora que recibiría el cliente ya no sería exacta. Si el cliente hace dos peticiones y recibe dos paquetes del servidor en distinto orden, realmente no importa porque el cliente puede imaginarse que los paquetes no están en orden y pedir otro. El canal fiable, aquí no es necesario, causando una degradación del rendimiento, y podría estorbar a la utilidad del servicio.

Otro ejemplo de servicio que no necesita un canal de fiabilidad garantizada es el comando ping. El único objetivo del comando ping es comprobar la comunicación entre dos programas a través de la red. De hecho, ping necesita conocer las caídas o los paquetes fuera de orden para determinar lo buena o mala que es la conexión. Así un canal fiable invalidaría este servicio.

El protocolo UDP proporciona una comunicación no garantizada entre dos aplicaciones en la Red. UDP no está basado en la conexión como TCP. UDP envía paquetes de datos, llamados datagramas de una aplicación a la otra. Enviar datagramas es como enviar una carta a través del servicio de correos: el orden de envío no es importante y no está garantizado, y cada mensaje es independiente de los otros.

Definición: UDP es un protocolo que envía paquetes de datos independientes, llamados datagramas desde un ordenador a otro sin garantías sobre su llegada. UDP no está basado en la conexión como TCP.

Puertos

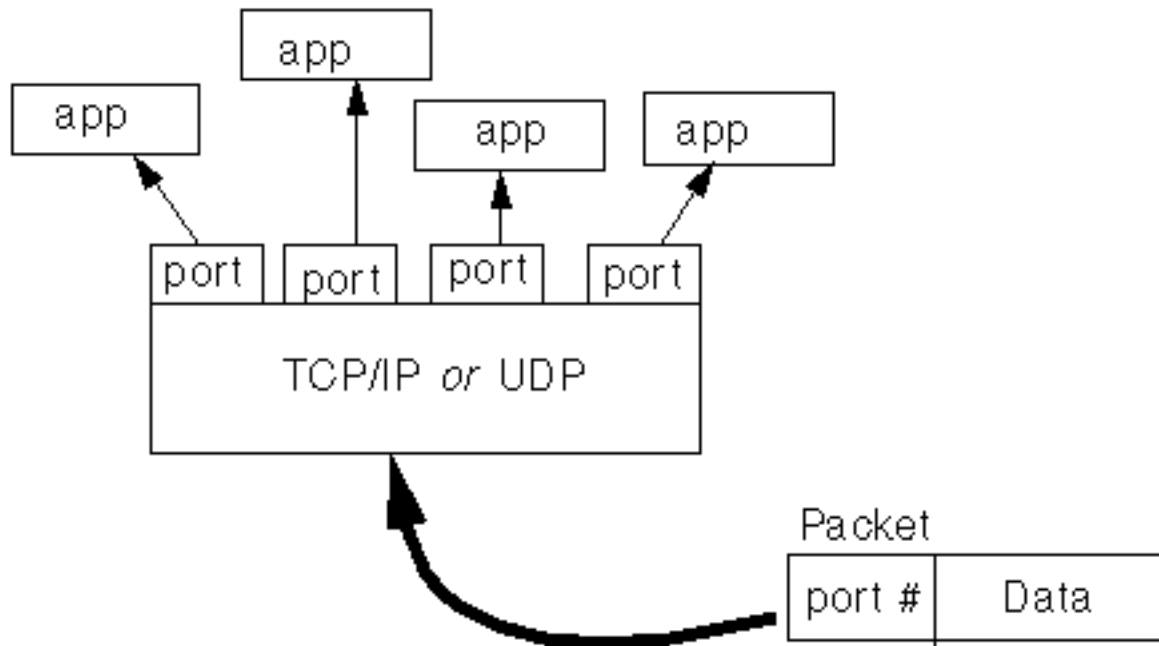
Generalmente hablando, un ordenador tiene una sola conexión física con la Red. Todos los datos destinados a un ordenador particular llegan a través de la conexión. Sin embargo, los datos podrían ser utilizados por diferentes aplicaciones ejecutándose en el ordenador. ¿Entonces cómo sabe el ordenador a qué aplicación enviarle los datos? A través del uso de los puertos.

Los datos transmitidos por internet están acompañados por una información de dirección que identifica el ordenador y el puerto al que están destinados. El ordenador está identificado por su dirección IP de 32 bits, esta dirección se utiliza para enviar los datos al ordenador correcto en la red. Los puertos están identificados por un número de 16 bits, que TCP y UDP utilizan para enviar los datos a la aplicación correcta.

En aplicaciones basadas en la conexión, una aplicación establece una

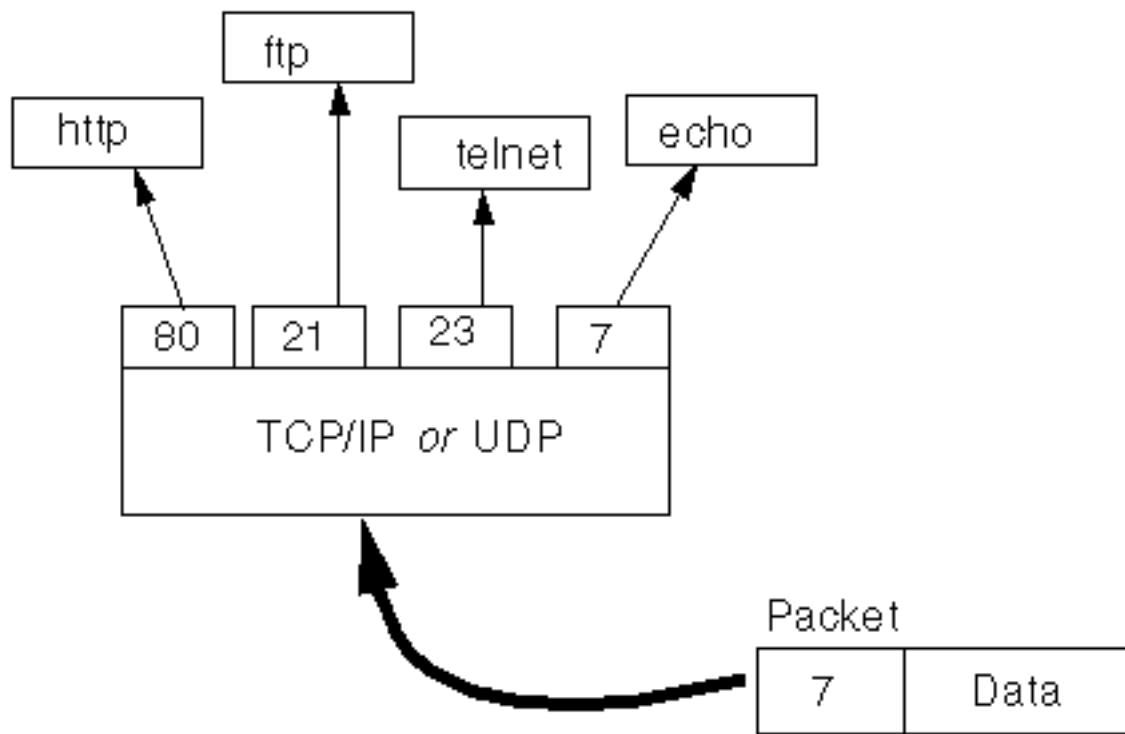
conexión con otra aplicación uniendo un socket a un número de puerto. Esto tiene el efecto de registrar la aplicación con el sistema para recibir todos los datos destinados a ese puerto. Dos aplicaciones no pueden utilizar el mismo puerto: intentar acceder a un puerto que ya está utilizado dará un error.

En comunicaciones basadas en datagramas, los paquetes de datagramas contienen el número de puerto del destinatario.



Definición: Los protocolos TCP y UDP utilizan puertos para dirigir los datos de entrada a los procesos particulares que se están ejecutando en un ordenador.

Los números de puertos tienen un rango de 0 a 65535 (porque los puertos están representados por un número de 16 bits). Los puertos entre los números 0 - 1023 están restringidos -- están reservados para servicios bien conocidos como HTTP, FTP y otros servicios del sistema. Tus aplicaciones no deberían intentar unirse a estos puertos. Los puertos que están reservados para los servicios bien conocidos como HTTP y FTP son llamados puertos bien conocidos.



A través de las clases del paquete `java.net`, los programas Java pueden utilizar TCP o UDP para comunicarse a través de Internet. Las clases `URL`, `URLConnection`, `Socket`, y `SocketServer` utilizan el TCP para comunicarse a través de la Red. Las clases `DatagramPacket` y `DatagramServer` utilizan UDP.

Lo que ya podrías Conocer sobre el Trabajo en Red en Java

Las palabras trabajo en red lanzan el temor en los corazones de muchos programadores. Temor no! Utilizando las capacidades proporcionadas por el entorno Java es muy sencillo. De hecho, podrías haber utilizado la red sin saberlo!

Cargar Applets desde la Red

Si accedes a un navegador que soporta Java, indudablemente habrás ejecutado muchos applets. Los applets que has ejecutado están referenciados por una etiqueta especial en el fichero HTML -- la etiqueta `<APPLET>`. Los applets pueden situarse en cualquier lugar, en una máquina local o en cualquier lugar de Internet. La posición del applet es completamente invisible para ti, el usuario. Sin embargo, la posición del applet está codificada dentro de la etiqueta `<APPLET>`. El navegador, decodifica esta información, localiza el applet y lo ejecuta. Si el applet está en otra máquina distinta a la tuya, el navegador debe descargar el applet antes de ejecutarlo.

Esto es el acceso de alto nivel que tienes en Internet desde el entorno de desarrollo de Java. Alguién ha utilizado su tiempo en escribir un navegador que hace todo el trabajo sucio de conexión a la red, y obtener los datos de ella, y que te permite ejecutar applets de cualquier lugar del mundo.

Para más información:

Las lecciones de [Escribir Applets](#) describen como escribir applets Java desde la A hasta la Z.

Cargar Imágenes desde URLs

Si te has aventurado a escribir tus propios applets y aplicaciones Java, podrías haber ejecutado una clase del paquete `java.net` llamada `URL`. Esta clase representa un Uniform Resource Locator, que es la dirección de algún recurso en la red. Tus applets y aplicaciones pueden utilizar una URL para referenciar e incluso conectarse a recursos de la red. Por ejemplo, para cargar una imagen desde la red, un programa Java debe primero crear una URL que contenga la dirección de la imagen.

Esta es la siguiente interacción de alto nivel que puedes tener con Internet -- tus programas Java obtienen una dirección de algo que quieren, crean una URL para ello, y utilizan alguna función existente en el entorno de desarrollo de Java que hace el trabajo sucio de conectar con la red y recuperar el recurso.

Para más información:

[Cargar Imágenes](#) muestra cómo cargar una imagen en tu programa Java (tanto en applets como en aplicaciones) cuando se tiene su URL. Antes de poder cargar la imagen debe crear un objeto URL con la dirección del recurso.

[Trabajar con URLs](#), la siguiente lección en esta ruta, proporciona una completa explicación sobre las URLs, incluyendo cómo pueden tus programas conectar con ellas y leer y escribir datos desde esa conexión.

¿Qué es una URL?

Si has navegado por la World Wide Web, indudablemente habrás oído el término URL y habrás utilizado URLs para acceder a varias páginas HTML de la Web. Entonces, ¿qué es exactamente una URL? Bien, lo siguiente es una sencilla, pero formal definición de URL:

Definición: URL es un acrónimo que viene de Uniform Resource Locator y es una referencia (una dirección) a un recurso de Internet.

Algunas veces es más sencillo (aunque no enteramente acertado) pensar en una URL como el nombre de un fichero en la red porque la mayoría de las URLs se refieren a un fichero o alguna máquina de la red. Sin embargo, deberías recordar que las URLs pueden apuntar a otros recursos de la red, como consultas a bases de datos, o salidas de comandos.

Lo siguiente es un ejemplo de una URL:

<http://java.sun.com/>

Esta URL particular direcciona la Web de Java hospedada por Sun Microsystems. La URL anterior, como otras URLs, tiene dos componentes principales:

- El identificador de protocolo
- El nombre del recurso

En el ejemplo, http es el identificador de protocolo y //java.sun.com/ es el nombre del recurso.

El identificador de protocolo indica el nombre del protocolo a utilizar para buscar ese recurso. El ejemplo utiliza el protocolo Hyper Text Transfer Protocol (HTTP), que es utilizado típicamente para servir documentos de hipertexto. HTTP es sólo uno de los diferentes protocolos utilizados para acceder a los distintos tipos de recursos de la red. Otros protocolos incluyen File Transfer Protocol (ftp), Gopher (gopher), File (file), y News (news).

El nombre del recurso es la dirección completa al recurso. El formato del nombre del recurso depende completamente del protocolo utilizado, pero la mayoría de los formatos de nombres de recursos contienen uno o más de los siguientes componentes:

nombre del host

 nombre de la máquina donde reside el recurso.

nombre de fichero

 el path al fichero dentro de la máquina

número de puerto

 el número de puerto a conectar (normalmente es opcional)

referencia

una referencia a una posición marcada dentro del recurso; normalmente identifica una posición específica dentro de un fichero (normalmente es opcional)

Para muchos protocolos, el nombre del host y el nombre del fichero son obligatorios y el número de puerto y la referencia son opcionales. Por ejemplo, el nombre de recursos para una URL HTTP debería especificar un servidor de la red (el nombre del host) y el path al documento en esa máquina (nombre de fichero), y también puede especificar un número de puerto y una referencia. En la URL mostrada anteriormente, java.sun.com es el nombre del host y la barra inclinada '/' es el path para el nombre del fichero /index.html.

Cuando construyas una URL, pon primero el indentificador de protocolo, seguido por dos puntos (:), seguido por el nombre del recurso, de esta forma:

protocoloID:nombredeRecursos

El paquete java.net contiene una clase llamada URL que utilizan los programas Java para representar una dirección URL. Tus programas Java pueden construir un objeto URL, abrir una conexión con el, leer y escribir datos a través de esa conexión. Las páginas siguientes de esta lección le enseñan cómo trabajar con objetos URL en programas Java.

También puedes ver

[java.net.URL](#)

Nota sobre la terminología: El término URL puede ser ambiguo -- se puede referir al concepto de la dirección de algo en Internet o a un objeto URL en tus programas Java. Normalmente el significado es claro dependiendo del contexto de la sentencia, o si no importa que la sentencia se aplique a ambos. Sin embargo, donde los significados de URL deban diferenciarse uno del otro, este texto utiliza Dirección URL para indicar el concepto de una dirección de Internet y objeto URL para referirse a un ejemplar de la clase URL en tu programa.

Ozito

Crear una URL

La forma más sencilla de crear un objeto URL es crearlo desde una Cadena que represente la forma "Leible" de la dirección URL. Esta es la forma típica que otra persona utilizaría para decirte una URL. Por ejemplo, para decirle la dirección de Gamelan que contiene una lista de sitios sobre Java, podríamos dártela de la siguiente forma:

<http://www.gamelan.com/>

En tu progrma Java, puede utilizar una cadena que contenga el texto anterior para crear un objeto URL:

```
URL gamelan = new URL("http://www.gamelan.com/");
```

El objeto URL anterior representa una URL absoluta. Una URL absoluta contiene toda la información necesaria para alcanzar el recurso en cuestión. También puedes crear objetos URL desde una dirección URL relativa.

Crear una URL relativa a otra

Una URL relativa sólo contiene la información suficiente para alcanzar el recurso en relación a (o en el contexto de) otra URL.

Las epecificaciones de las URL relativas se utilizan frecuentemente en ficheros HTML. Por ejemplo, supon que has escrito un fichero HTML llamado HomePage.html. Dentro de esta página, hay enlaces a otras páginas, graficos.html y Preferencias.html, que están en la misma máquina y en el mismo directorio que HomePage.html. Estos enlaces a graficos.html y Preferencias.html desde HomePage.html podrían especificarse sólo como nombres de ficheros, de la siguiente forma:

```
<a href="graficos.html">graficos</a>
<a href="preferencias.html">Preferencias</a>
```

Estas direcciones URL son URLs relativas. Esto es, las URL estás especificadas en relación al fichero en el que están contenidas HomePage.html.

En tus programas java, puedes crear un objeto URL desde una especificación URL relativa. Por ejemplo, supon que ya has creado una URL para "http://www.gamelan.com/" en tu programa, y que sabes los nombres de varios ficheros en esa site(Gamelan.network.html, y Gamelan.animation.html). Puedes crear URLs para cada fichero de Gamelan simplemente especificando el nombre del fichero en el contexto de la URL original de Gamelan. Los nombres de ficheros son URLs relativas y están en relación a la URL original de Gamelan.

```
URL gamelan = new URL("http://www.gamelan.com/");
URL gamelanNetwork = new URL(gamelan, "Gamelan.network.html");
```

Este código utiliza un constructor de la clase URL que permite crear un objeto URL desde un objeto URL (la base) y una URL relativa.

Este constructor también es útil para crear URL llamados anclas (también conocidos como referencias) dentro de un fichero. Por ejemplo, supon que el fichero "Gamelan.network.html" tiene una referencia llamada BOTTOM que está al final del fichero. Puedes utilizar el constructor de URL relativa para crear una URL como esta:

```
URL gamelanNetworkBottom = new URL(gamelanNetwork, "#BOTTOM");
```

La forma general de este constructor de URL es:

```
URL(URL URLbase, String URLrelativa)
```

El primer argumento es un objeto URL que especifica la base de la neva URL, y el segundo argumento es una cadena que especifica el resto del nombre del recurso

relativo a la base. Si URLbase es null, entonces este constructor trata URLrelativa como si fuera una especificación de una URL absoluta. Y al revés, si relativeURL es una especificación de URL absoluta, entonces el constructor ignora baseURL.

Otros Constructores de URL

La clase URL proporciona dos constructores adicionales para crear un objeto URL. Estos constructores son útiles cuando trabajan con URLs como URLs HTTP, que tienen los componentes del nombre del host, el nombre del fichero, el número de puerto y una referencia en la parte del nombre del recurso de la URL. Estos dos constructores son útiles cuando no se tiene una cadena que contiene la especificación completa de la URL, pero si conocen algunos componentes de la URL.

Por ejemplo, si has diseñado un navegador con un panel similar al explorador de ficheros que le permite al usuario utilizar el ratón para seleccionar el protocolo, el nombre del host, el número del puerto, el nombre del fichero, puedes construir una URL a partir de estos componentes. El primer constructor crea una URL desde un protocolo, un nombre de host y un nombre de fichero. El siguiente código crea una URL del fichero Gamelan.network.html en la site de Gamelan:

```
URL gamelan = new URL("http", "www.gamelan.com", "/Gamelan.network.html");
```

Esto es equivalente a

`URL("http://www.gamelan.com/Gamelan.network.html")`. El primer argumento es el protocolo, el segundo es el nombre del host y el último argumento es el path del fichero. Observa que el nombre del fichero contiene la barra inclinada (/) al principio. Esto indica que el nombre de fichero está especificado desde la raíz del host.

El último constructor de URL añade el número de puerto a la lista de los argumentos utilizados por el constructor anterior.

```
URL gamelan = new URL("http", "www.gamelan.com", 80, "/Gamelan.network.html");
```

Esto crea un objeto URL con la siguiente dirección URL:

```
http://www.gamelan.com:80/Gamelan.network.html
```

Si construyes una URL utilizando uno de estos constructores, puedes obtener una cadena que contiene la dirección URL completa, utilizando el método `toString()` de la clase URL o el método `toExternalForm()` equivalente.

MalformedURLException

Cada uno de los cuatro constructores de URL lanza una `MalformedURLException` si los argumentos del constructor son nulos o el protocolo es desconocido. Típicamente, se querrá capturar y manejar esta excepción. Así normalmente deberías introducir tu constructor de URL en un par try/catch.

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    . . .
    // Aquí va el código del manejador de excepciones
    . . .
}
```

Puede ver [Manejar Errores Utilizando Excepciones](#) para obtener información sobre el manejo de excepciones.

Nota: Las URLs son objetos de "una sólo escritura". Una vez que has creado un objeto URL no se

puede cambiar ninguno de sus atributos (protocolo, nombre del host, nombre del fichero ni número de puerto).

Ozito

Analizar una URL

La clase URL proporciona varios métodos que permiten preguntar a los objetos URL. Puede obtener el protocolo, nombre de host, número de puerto, y nombre de fichero de una URL utilizando estos métodos accesorios:

`getProtocol()`

Devuelve el componente identificador de protocolo de la URL.

`getHost()`

Devuelve el componente nombre del host de la URL.

`getPort()`

Devuelve el componente número del puerto de la URL. Este método devuelve un entero que es el número de puerto. Si el puerto no está seleccionado, devuelve -1.

`getFile()`

Devuelve el componente nombre de fichero de la URL.

`getRef()`

Obtiene el componente referencia de la URL.

Nota: Recuerda que no todas las direcciones URL contienen estos componentes. La clase URL proporciona estos métodos porque las URLs de HTTP contienen estos componentes y quizás son las URLs más utilizadas. La clase URL está centrada de alguna forma sobre HTTP.

Se pueden utilizar estos métodos `getXXX()` para obtener información sobre la URL sin importar el constructor que se haya utilizado para crear el objeto URL.

La clase URL, junto con estos métodos accesorios, libera de tener que analizar la URL de nuevo! Dando a cualquier cadena la especificación de una URL, y sólo creando un nuevo objeto URL y llamando a uno de sus métodos accesorios para la información que se necesite. Este pequeño programa de ejemplo crea una URL partiendo de una especificación y luego utiliza los métodos accesorios del objeto URL para analizar la URL:

```
import java.net.*;
import java.io.*;

class ParseURL {
    public static void main(String[] args) {
        URL aURL = null;
        try {
            aURL = new URL("http://java.sun.com:80/tutorial/intro.html#DOWNLOADING");
            System.out.println("protocol = " + aURL.getProtocol());
            System.out.println("host = " + aURL.getHost());
            System.out.println("filename = " + aURL.getFile());
            System.out.println("port = " + aURL.getPort());
            System.out.println("ref = " + aURL.getRef());
        } catch (MalformedURLException e) {
            System.out.println("MalformedURLException: " + e);
        }
    }
}
```

Aquí tienes la salida mostrada por el programa:

```
protocol = http
host = java.sun.com
filename = /tutorial/intro.html
port = 80
ref = DOWNLOADING
```


Leer Directamente desde una URL

Después de haber creado satisfactoriamente una URL, se puede llamar al método `openStream()` de la clase `URL` para obtener un canal desde el que poder leer el contenido de la URL. El método retorna un objeto [java.io.InputStream](#) por lo que se puede leer normalmente de la URL utilizando los métodos normales de `InputStream`. [Canales de Entrada y Salida](#) describe las clases de I/O proporcionadas por el entorno de desarrollo de Java y enseña cómo utilizarlas.

Leer desde una URL es tan sencillo como leer de un canal de entrada. El siguiente programa utiliza `openStream()` para obtener un stream de entrada a la URL `"http://www.yahoo.com/"`. Lee el contenido del canal de entrada y lo muestra en la pantalla.

```
import java.net.*;
import java.io.*;

class OpenStreamTest {
    public static void main(String[] args) {
        try {
            URL yahoo = new URL("http://www.yahoo.com/");
            DataInputStream dis = new DataInputStream(yahoo.openStream());
            String inputLine;

            while ((inputLine = dis.readLine()) != null) {
                System.out.println(inputLine);
            }
            dis.close();
        } catch (MalformedURLException me) {
            System.out.println("MalformedURLException: " + me);
        } catch (IOException ioe) {
            System.out.println("IOException: " + ioe);
        }
    }
}
```

Cuando ejecutes el programa, deberás ver los comandos HTML y el contenido textual del fichero HTML localizado en `"http://www.yahoo.com/"` desplazándose por su ventana de comandos.

O podrías ver el siguiente mensaje de error:

```
IOException: java.net.UnknownHostException: www.yahoo.com
```

El mensaje anterior indica que se podría tener seleccionado un proxy y por eso el programa no puede encontrar el servidor `www.yahoo.com`. (Si es necesario, preguntale a tu administrador por el proxy de su servidor.)

Conectar con una URL

Si has creado satisfactoriamente una URL, puedes llamar al método `openConnection()` de la clase `URL` para conectar con ella. Cuando hayas conectado con una URL habrá inicializado un enlace de comunicación entre un programa Java y la URL a través de la red. Por ejemplo, puedes abrir una conexión con el motor de búsqueda de Yahoo con el código siguiente:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    yahoo.openConnection();
} catch (MalformedURLException e) {           // nueva URL() fallada
    . . .
} catch (IOException e) {                   // openConnection() fallada
    . . .
}
```

Si es posible, el método `openConnection()` crea un nuevo objeto `URLConnection` (si no existe ninguno apropiado), lo inicializa, conecta con la URL y devuelve el objeto `URLConnection`. Si algo va mal -- por ejemplo, el servidor de Yahoo está apagado -- el método `openConnection()` lanza una `IOException`.

Ahora que te has conectado satisfactoriamente con la URL puedes utilizar el objeto `URLConnection` para realizar algunas acciones como leer o escribir a través de la conexión. La [siguiente sección](#) de esta lección te enseña cómo leer o escribir a través de un objeto `URLConnection`.

También puedes ver

java.net.URLConnection

Leer y Escribir a través de un objeto URLConnection

Si has utilizado satisfactoriamente `openConnection()` para inicializar comunicaciones con una URL, tendrás una referencia a un objeto `URLConnection`. La clase `URLConnection` contiene muchos métodos que permiten comunicarse con la URL a través de la red. `URLConnection` es una clase centrada sobre HTTP -- muchos de sus métodos son útiles sólo cuando trabajan con URLs HTTP. Sin embargo, la mayoría de los protocolos URL permite leer y escribir desde una conexión por eso esta página enseña como leer y escribir desde una URL a través de un objeto `URLConnection`.

Leer desde un objeto URLConnection

El siguiente programa realiza la misma función que el mostrado en [Leer Directamente desde una URL](#). Sin embargo, mejor que abrir directamente un stream desde la URL, este programa abre explícitamente una conexión con la URL, obtiene un stream de entrada sobre la conexión, y lee desde el stream de entrada:

```
import java.net.*;
import java.io.*;

class ConnectionTest {
    public static void main(String[] args) {
        try {
            URL yahoo = new URL("http://www.yahoo.com/");
            URLConnection yahooConnection = yahoo.openConnection();
            DataInputStream dis = new
DataInputStream(yahooConnection.getInputStream());
            String inputLine;

            while ((inputLine = dis.readLine()) != null) {
                System.out.println(inputLine);
            }
            dis.close();
        } catch (MalformedURLException me) {
            System.out.println("MalformedURLException: " + me);
        } catch (IOException ioe) {
            System.out.println("IOException: " + ioe);
        }
    }
}
```

La salida de este programa debería ser idéntica a la salida del programa que abría directamente el stream desde la URL. Puedes utilizar cualquiera de estas dos formas para leer desde una URL. Sin embargo, algunas veces leer desde una `URLConnection` en vez de leer directamente desde una URL podría ser más útil ya que se puede utilizar el objeto `URLConnection` para otras tareas (como escribir sobre la conexión URL) al mismo tiempo.

De nuevo, si en vez de ver la salida del programa, se viera el siguiente mensaje error:

```
IOException: java.net.UnknownHostException: www.yahoo.com
```

Podrías tener activado un proxy y el programa no podría encontrar el servidor de `www.yahoo.com`.

Escribir a una URLConnection

Muchas páginas HTML contienen forms -- campos de texto y otros objeto GUI que le permiten introducir datos en el servidor. Después de teclear la información requerida e iniciar la petición pulsando un botón, el navegador que se utiliza escribe los datos en la URL a través de la red. Después de que la otra parte de la conexión (normalmente un script

cgi-bin) en el servidor de datos, los procesa, y le envía de vuelta una respuesta, normalmente en la forma de una nueva página HTML. Este esenario es el utilizado normalmente por los motores de búsqueda.

Muchos scripts cgi-bin utilizan el POST METHOD para leer los datos desde el cliente. Así, escribir sobre una URL frecuentemente es conocido como posting a URL. Los scripts del lado del servidor utilizan el método POST METHOD para leer desde su entrada estandard.

Nota: Algunos scripts cgi-bin del lado del servidor utilizan el método GET METHOD para leer sus datos. El método POST METHOD es más rápido haciendo que GET METHOD esté obsoleto porque es más versátil y no tiene limitaciones sobre los datos que pueden ser enviados a través de la conexión.

Tus programas Java también pueden interactuar con los scripts cgi-bin del lado del servidor. Sólo deben poder escribir a una URL, así proporcionan los datos al servirdor. Tu programa puede hacer esto siguiendo los siguientes pasos:

1. Crear una URL.
2. Abrir una conexión con la URL.
3. Obtener un stream de salida sobre la conexión. Este canal de entrada está conectado al stream de entrada estandard del script cgi-bin del servidor.
4. Escribir en el stream de salida.
5. Cerrar el stram de salida.

Hassan Schroeder, un miembro del equipo de Java, escribió un script cgi-bin, llamado [backwards](#), y está disponible en la Web site de, [java.sun.com](#). Puedes utilizar este script para probar el siguiente programa de ejemplo. Si por alguna razón no puedes obtenerlo de nuestra Web; puedes poner el script en cualquier lugar de la red, llamándolo backwards, y prueba el programa localmente.

El script de nuestra Web lee una cadena de la entrada estandard, invierte la cadena, y escribe el resultado en la salida estandard. El script requiere una entrada de la siguiente forma: string=string_to_reverse, donde string_to_reverse es la cadena cuyos caracteres van a mostrarse en orden inverso.

Aquí tienew un programa de ejemplo que ejecuta el script backwards a través de la red utilizando un URLConnection:

```
import java.io.*;
import java.net.*;

public class ReverseTest {
    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.err.println("Usage:  java ReverseTest string_to_reverse");
                System.exit(1);
            }
            String stringToReverse = URLEncoder.encode(args[0]);

            URL url = new URL("http://java.sun.com/cgi-bin/backwards");
            URLConnection connection = url.openConnection();

            PrintStream outStream = new PrintStream(connection.getOutputStream());
            outStream.println("string=" + stringToReverse);
            outStream.close();

            DataInputStream inStream = new
```

```

DataInputStream(connection.getInputStream());
    String inputLine;

    while ((inputLine = inStream.readLine()) != null) {
        System.out.println(inputLine);
    }
    inStream.close();
} catch (MalformedURLException me) {
    System.err.println("MalformedURLException: " + me);
} catch (IOException ioe) {
    System.err.println("IOException: " + ioe);
}
}
}

```

Examinemos el programa y veamos como trabaja. Primero, el programa procesa los argumentos de la línea de comandos:

```

if (args.length != 1) {
    System.err.println("Usage:  java ReverseTest string_to_reverse");
    System.exit(1);
}
String stringToReverse = URLEncoder.encode(args[0]);

```

Estas líneas aseguran que el usuario proporciona uno y sólo un argumento de la línea de comandos del programa y lo codifica. El argumento de la línea de comandos es la cadena a invertir por el script cgi-bin backwards. El argumento de la línea de comandos podría tener espacios u otros caracteres no alfanuméricos. Estos caracteres deben ser codificados porque podrían suceder varios procesos en la cadena en el lado del servidor. Esto se consigue mediante la clase URLEncoder.

Luego el programa crea el objeto URL -- la URL para el script backwards en java.sun.com.

```
URL url = new URL("http://java.sun.com/cgi-bin/backwards");
```

El programa crea una URLConnection y abre un stream de salida sobre esa conexión. El stream de salida está filtrado a través de un PrintStream.

```
URLConnection connection = url.openConnection();
PrintStream outStream = new PrintStream(connection.getOutputStream());

```

La segunda línea anterior llama al método getOutputStream() sobre la conexión. Si no URL no soporta salida, este método lanza una UnknownServiceException. Si la URL soporta salida, este método devuelve un stream de salida que está conectado al stream de entrada estandard de la URL en el lado del servidor -- la salida del cliente es la entrada del servidor.

Luego, el programa escribe la información requerida al stream de salida y cierra el stream:

```
outStream.println("string=" + stringToReverse);
outStream.close();

```

Esta línea escribe en el canal de salida utilizando el método println(). Como puedes ver, escribir datos a una URL es tan sencillo como escribir datos en un stream. Los datos escritos en el stream de salida en el lado del cliente son la entrada para el script backwards en el lado del servidor. El programa ReverseTest construye la entrada en la forma requerida por el script mediante la concatenación string= para codificar la cadena.

Frecuentemente, como en este ejemplo, cuando escribe en una URL está pasando información al script cgi-bin que lee la información que usted escribe, realiza alguna acción y luego envía la información de vuelta mediante la misma URL. Por lo que querrás leer desde la URL después de haber escrito en ella. El programa ReverseTest los hace de esta forma:

```

DataInputStream inStream = new DataInputStream(connection.getInputStream());
String inputLine;

```

```
while (null != (inputLine = inStream.readLine())) {  
    System.out.println(inputLine);  
}  
inStream.close();
```

Cuando ejecutes el programa ReverseTest utilizando Invierteme como argumento, deberías ver esta salida:

```
Invierteme  
  reversed is:  
emetreivnI
```

Todo sobre los Sockets

Utilizas URLs y URLConnections para comunicarte a través de la red a un nivel relativamente alto y para un propósito específico: acceder a los recursos de Internet. Algunas veces tus programas requieren una comunicación en la red a un nivel más bajo, por ejemplo, cuando quieras escribir una aplicación cliente-servidor.

En aplicaciones cliente-servidor, el servidor proporciona algún servicio, como procesar consultas a bases de datos o enviar los precios actualizados del stock. El cliente utiliza el servicio proporcionado por el servidor para algún fin, mostrar los resultados de la consulta a la base de datos, o hacer recomendaciones de pedidos a un inversor. La comunicación que ocurre entre el cliente y el servidor debe ser fiable -- los datos no pueden caerse y deben llegar al lado del cliente en el mismo orden en el que fueron enviados.

TCP proporciona un canal de comunicación fiable punto a punto, lo que utilizan para comunicarse las aplicaciones cliente-servidor en Internet. Las clases Socket y ServerSocket del paquete java.net proporcionan un canal de comunicación independiente del sistema utilizando TCP.

¿Qué es un Socket?

Un socket es un punto final en un enlace de comunicación de dos vías entre dos programas que se ejecutan en la red. Las clases Socket son utilizadas para representar conexiones entre un programa cliente y otro programa servidor. El paquete java.net proporciona dos clases -- Socket y ServerSocket -- que implementan los lados del cliente y del servidor de una conexión, respectivamente.

Leer y Escribir a través de un Socket

Esta página contiene un pequeño ejemplo que ilustra cómo un programa cliente puede leer y escribir a través de un socket.

Escribir desde el Lado del Servidor a través de un Socket

La página anterior mostró un ejemplo de cómo escribir en un programa cliente que interactúa con servidor existente mediante un objeto Socket. Esta página muestra cómo se puede escribir un programa que implemente el otro lado de la conexión -- un programa servidor.

¿Qué es un Socket?

Una aplicación servidor normalmente escucha a un puerto específico esperando una petición de conexión de un cliente. Cuando llega una petición de conexión, el cliente y el servidor establecen una conexión dedicada sobre la que poder comunicarse. Durante el proceso de conexión, el cliente es asignado a un número de puerto, y ata un socket a ella. El cliente habla al servidor escribiendo sobre el socket y obtiene información del servidor cuando lee de él. Similarmente, el servidor obtiene un nuevo número de puerto local (necesita un nuevo puerto para poder continuar escuchando para petición de conexión del puerto original.) El servidor también ata un socket a este puerto local y comunica con él mediante la lectura y escritura sobre él.

El cliente y el servidor deben ponerse de acuerdo sobre el protocolo -- esto es, debe ponerse de acuerdo en el lenguaje para transferir la información de vuelta a través del socket.

Definición: Un socket es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan a través de la red.

El paquete `java.net` del entorno de desarrollo de Java proporciona una clase -- `Socket` -- que representa un final de una comunicación de dos vías entre un programa java y otro programa de la red. La clase `Socket` implementa el lado del servidor de un enlace de dos vías. Si estás escribiendo software de servidor, también estarás interesado en la clase `ServerSocket` que implementa el lado del servidor en un enlace de dos vías. Esta lección muestra cómo utilizar las clases `Socket` y `ServerSocket`.

Si estás; intentando conectar con la World Wide Web, la clase `URL` y las clases relacionadas con esta (`URLConnection`, `URLEncoder`) son más indicadas para lo que estás haciendo. Las URLs son conexiones de nivel relativamente alto para la Web y utilizan los sockets como parte de su implementación interna. Puedes ver [Trabajar con URLs](#) para más información sobre como conectarse con la Web mediante URLs.

También puedes ver

[java.net.ServerSocket](#)
[java.net.Socket](#)

Leer y Escribir a través de un Socket

El siguiente programa es un ejemplo sencillo de cómo establecer una conexión entre un programa cliente y otro servidor utilizando sockets. La clase Socket del paquete java.net es una implementación independiente de la plataforma de un cliente para un enlace de comunicación de dos vías entre un cliente y un servidor. La clase Socket se sitúa en la parte superior de una implementación dependiente de la plataforma, ocultando los detalles de los sistemas particulares a un programa Java. Utilizando la clase java.net.Socket en lugar de tratar con código nativo, los programas Java pueden comunicarse a través de la red de una forma independiente de la plataforma.

Este programa cliente, EchoTest, conecta con el Echo del servidor (en el port 7) mediante un socket. El cliente lee y escribe a través del socket. EchoTest envía todo el texto tecleado en su entrada standard al Echo del servidor, escribiendole el texto al socket. El servidor repite todos los caracteres recibidos en su entrada desde el cliente de vuelta a través del socket al cliente. El programa cliente lee y muestra los datos pasados de vuelta desde el servidor.

```
import java.io.*;
import java.net.*;

public class EchoTest {
    public static void main(String[] args) {
        Socket echoSocket = null;
        DataOutputStream os = null;
        DataInputStream is = null;
        DataInputStream stdIn = new DataInputStream(System.in);

        try {
            echoSocket = new Socket("taranis", 7);
            os = new DataOutputStream(echoSocket.getOutputStream());
            is = new DataInputStream(echoSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis");
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: taranis");
        }

        if (echoSocket != null && os != null && is != null) {
            try {
                String userInput;

                while ((userInput = stdIn.readLine()) != null) {
                    os.writeBytes(userInput);
                    os.writeByte('\n');
                    System.out.println("echo: " + is.readLine());
                }
                os.close();
                is.close();
                echoSocket.close();
            } catch (IOException e) {
                System.err.println("I/O failed on the connection to: taranis");
            }
        }
    }
}
```



```
}
```

Paseemos a través del programa e investiguemos las cosas interesantes.

Las siguientes tres líneas de código dentro del primer bloque try del método main() son críticos -- establecen la conexión del socket entre el cliente y el servidor y abre un canal de entrada y un canal de salida sobre el socket:

```
echoSocket = new Socket("taranis", 7);  
os = new DataOutputStream(echoSocket.getOutputStream());  
is = new DataInputStream(echoSocket.getInputStream());
```

La primera línea de esta secuencia crea un nuevo objeto Socket y lo llama echoSocket. El constructor Socket utilizado aquí (hay otros tres) requiere el nombre de la máquina y el número de puerto al que quiere conectarse. El programa de ejemplo utiliza el host taranis, que es el nombre de una máquina (hipotética) de nuestra red local. Cuando teclees y ejecutes este programa en tu máquina, deberías cambiar este nombre por una máquina de tu red. Asegurate de que el nombre que utiliza tienes el nombre IP totalmente cualificado de la máquina a la que te quieres conectar. El segundo argumento es el número de puerto. El puerto número 7 es el puerto por el que escucha el Echo del servidor.

La segunda línea del código anterior abre un canal de etnrada sobre el socket, y la tercera línea abre un canal de salida sobre el mismo socket. EchoTest sólo necesita escribir en el stream de salida y leer del stream de entrada para comunicarse a través del socket con el servidor. El resto del programa hace esto. Si no estás familiarizado con los streams de entrada y salida, podrías querer leer [Streams de Entrada y Salida](#).

La siguiente sección de código lee desde el stream de entranda estandard de EchoTest (donde el usuario teclea los datos) una línea cada vez. EchoTest escribe inmediatamente la entada seguida por un carácter de nueva línea en el stream de salida conectado al socket.

```
String userInput;  
  
while ((userInput = stdIn.readLine()) != null) {  
    os.writeBytes(userInput);  
    os.writeByte('\n');  
    System.out.println("echo: " + is.readLine());  
}
```

La última línea del bucle while lee una línea de información desde el stream de entrada conectado al socket. El método readLine() se bloquea hasta que el servidor haya devuelto la información a EchoTest. Cuando readline() retorna, EchoTest imprime la información en la salida estandard.

Este bloque continua -- EchoTest lee la entrada del usuairo, la envía al servidor Echo, obtiene una respuesta desde el servidor y la muestra -- hasta que el usuario teclee un carácter de final de entrada.

Cuando el usuario teclea un carácter de fin de entrada, el bucle while termina y el programa continúa ejecutando las siguientes líneas de código:

```
os.close();  
is.close();  
echoSocket.close();
```

Estas línea de código caen en la categoría de limpieza del hogar. Un programa con buen comportamienteo, se limpia a sí mismo y este programa tiene buen comportamiento. Estas tres líneas de código cierran las streams de entrada y salida conectados al socket, y cierra la conexión del socket con el servidor. El orden es importante -- debe cerrar los streams conectados a un socket antes de cerrar éste.

Este programa cliente tiene un comportamiento correcto y sencillo porque el servidor Echo implementa un protocolo sencillo. El cliente envía texto al servidor, y el servidor lo devuelve. Cuando tus programas clientes hablen con servidores más complicados como un servidor http, tu programa cliente también será más complicado. Si embargo, las cosas básicas son las que has visto en este programa:

1. Abrir un socket.
2. Abrir un stream de entrada y otro de salida hacia el socket.
3. Leer y escribir a través del socket de acuerdo al protocolo del servidor.
4. Cerrar los Streams.
5. Cerrar el socket.

Sólo el paso 3 será diferente de un cliente a otro, dependiendo del servidor. Los otros pasos permanecen inalterables.

También puede ver

java.net.Socket

Escribir el Lado del Servidor de un Socket

Esta sección le muestra cómo escribir el lado del servidor de una conexión socket, con un ejemplo completo cliente-servidor. El servidor en el pareja cliente/servidor sirve bromas "Knock Knock". Las bromas Knock Knock son las favoritas por los niños pequeños y que normalmente son vehículos para malos juegos de palabras. Son de esta forma:

Servidor: "Knock knock!"

Cliente: "¿Quién es?"

Servidor: "Dexter."

Cliente: "¿Qué Dexter?"

Servidor: "La entrada de Dexter con ramas de acebo."

Cliente: "Gemido."

El ejemplo consiste en dos programas Java independientes ejecutandose: el programa cliente y el servidor. El programa cliente está implementado por una sola clase KnockKnockClient, y está basado en el ejemplo [EchoTest](#) de la [página anterior](#). El programa servidor está implementado por dos clases: KnockKnockServer y KKState. KnockKnockServer contiene el método main() para el program servidor y realiza todo el trabajo duro, de escuchar el puerto, establecer conexiones, y leer y escribir a través del socket. KKState sirve la bromas: sigue la pista de la broma actual, el estado actual (enviar konck knock, enviar pistas, etc...) y servir varias piezas de texto de la broma dependiendo del estado actual. Esta página explica los detalles de cada clase en estos programas y finalmente le muestra cómo ejecutarlas.

El servidor Knock Knock

Esta sección pasa a través del código que implemente el programa servidor Knock Knock, Aquí tienes el código fuente completo de la clase [KnockKnockServer.class](#). El programa servidor empieza creando un nuevo objeto ServerSocket para escuchar en un puerto específico. Cuando escriba un servidor, debería elegir un puerto que no estuviera ya dedicado a otro servicio, KnockKnockServer escucha en el puerto 4444 porque sucede que el 4 es mi número favorito y el puerto 4444 no está siendo utilizado por ninguna otra cosa en mi entorno:

```
try {  
    serverSocket = new ServerSocket(4444);  
} catch (IOException e) {  
    System.out.println("Could not listen on port: " + 4444 + ", " + e);  
    System.exit(1);  
}
```

ServerSocket es una clase java.net que proporciona una implementación independientes del sistema del lado del servidor de una conexión cliente/servidor. El constructor de ServerSocket lanza una excepción por alguna razón (cómo que el puerto ya está siendo utilizado) no puede escuchar en el puerto especificado. En este caso, el KnockKnockServer no tiene elección pero sale.

Si el servidor se conecta con éxito con su puerto, el objeto ServerSocket se crea y el servidor continua con el siguiente paso, que es aceptar una conexión desde el cliente.

```
Socket clientSocket = null;  
try {  
    clientSocket = serverSocket.accept();  
} catch (IOException e) {  
    System.out.println("Accept failed: " + 4444 + ", " + e);  
    System.exit(1);  
}
```

El método accept() se bloquea (espera) hasta que un cliente empiece y pida una conexión el puerto (en este caso 4444) que el servidor está escuchando. Cuando el método accept() establece la conexión con éxito con el cliente, devuelve un objeto Socket que apunta a un puerto local nuevo. El servidor puede continuar con el cliente sobre este nuevo Socket en un

puerto diferente del que estaba escuchando originalmente para las conexiones. Por eso el servidor puede continuar escuchando nuevas peticiones de clientes a través del puerto original del `ServerSocket`. Esta versión del programa de ejemplo no escucha más peticiones de clientes. Sin embargo, una versión modificada de este programa, porpocionada más [adelante](#), si lo hace.

El código que hay dentro del siguiente bloque `try` implemente el lado del servidor de una comunicación con el cliente. Esta sección del servidor es muy similar al lado del cliente (que vió en el ejemplo de la página anterior y que verá más adelante en el ejemplo de la clase `KnockKnockClient`):

- Abre un stream de entrada y otro de salida sobre un socket.
- Lee y escribe a través del socket.

Empecemos con las primeras 6 líneas:

```
DataInputStream is = new DataInputStream(
    new BufferedInputStream(clientSocket.getInputStream()));
PrintStream os = new PrintStream(
    new BufferedOutputStream(clientSocket.getOutputStream(), 1024), false);
String inputLine, outputLine;
KKState kks = new KKState();
```

Las primeras dos líneas del código abren un stream de entrada sobre el socket devuelto por el método `accept()`. Las siguiente dos líneas abren un stream de salida sobre el mismo socket. La siguiente línea declara y crea un par de strings locales utilizadas para leer y escribir sobre el socket. Y finalmente, la última línea crea un objeto `KKState`. Este es el objeto que sigue la pista de la broma actual, el estado actual dentro de una broma, etc.. Este objeto implementa el protocolo -- el lenguaje que el cliente y el servidor deben utilizar para comunicarse.

El servidor es el primero en hablar, con estas líneas de código:

```
outputLine = kks.processInput(null);
os.println(outputLine);
os.flush();
```

La primera línea de código obtiene del objeto `KKState` la primera línea que el servidor le dice al cliente. Por ejemplo lo primero que el servidor dice es "Knock! Knock!".

Las siguientes dos líneas escriben en el stream de salida conectado al socket del cliente y vacía el stream de salida. Esta secuencia de código inicia la conversación entre el cliente y el servidor.

La siguiente sección de código es un bucle que lee y escribe a través del socket enviando y recibiendo mensajes entre el cliente y el servidor mientras que tengan que decirse algo el uno al otro. Como el servidor inicia la conversación con un "Knock! Knock!", el servidor debe esperar la respuesta del cliente. Así el bucle `while` itera y lee del stream de entrada. El método `readLine()` espera hasta que el cliente respondan algo escribiendo algo en el stream de salida (el stream de entrada del servidor). Cuando el cliente responde, el servidor pasa la respuesta al objeto `KKState` y le pide a éste una respuesta adecuada. El servidor inmediatamente envía la respuesta al cliente mediante el stream de salida conectado al socket, utilizando las llamadas a `println()` y `flush()`. Si la respuesta del servidor generada por el objeto `KKState` es "Bye.", indica que el cliente dijo que no quería más bromas y el bucle termina.

```
while ((inputLine = is.readLine()) != null) {
    outputLine = kks.processInput(inputLine);
    os.println(outputLine);
    os.flush();
    if (outputLine.equals("Bye. "))
        break;
}
```

La clase `KnockKnockServer` es un servidor de buen comportamiento, ya que las últimas líneas

de esta sección realizan la limpieza cerrando todas los streams de entrada y salida, el socket del cliente, y el socket del servidor.

```
os.close();
is.close();
clientSocket.close();
serverSocket.close();
```

El Protocolo Knock Knock

La clase [KKState](#) implementa el protocolo que deben utilizar el cliente y el servidor para comunicarse. Esta clase sigue la pista de dónde están el cliente y el servidor en su comunicación y sirve las respuestas del servidor a las setencias del cliente. El objeto `KKState` contiene el texto de todos las bromas y se asegura de que el servidor ofrece la respuesta adecuada a las frases del cliente. No debería decir el cliente "¿Qué Dexter?" cuando el servidor dice "Knock! Knock!".

Todos las parejas cliente-servidor deden tener algún protocolo en el que hablar uno con otro, o el significado de los datos que se pasan unos a otro. El protocolo que utilicen sus clientes y servidores dependen enteramente de la comunicación requerida por ellos para realizar su tarea.

El Cliente Knock Knock

La clase [KnockKnockClient](#) implementa el programa cliente que habla con `KnockKnockServer`. `KnockKnockClient` está basado en el programa [EchoTest](#) de la página [anterior](#) y debería serte familiar. Pero echemos un vistazo de todas formas para ver lo que sucede en el cliente, mientras tenemos en mente lo que sucedía en el servidor.

Cuando arranca el program cliente, el servidor debería estar ya ejecutándose y escuchando el puerto esperando un petición de conexión por parte del cliente.

```
kkSocket = new Socket("taranis", 4444);
os = new PrintStream(kkSocket.getOutputStream());
is = new DataInputStream(kkSocket.getInputStream());
```

Así, lo primero que hace el programa cliente es abrir un socket sobre el puerto en el que está escuchando el servidor en la máquina en la que se está ejecutando el servidor. El programa ejemplo `KnockKnockClient` abre un socket sobre el puerto 4444 que el mismo por el que está escuchando el servidor. `KnockKnockClient` utiliza el nombre de host `taranis`, que es el nombre de una máquina (hipotética) en tu red local. Cuando teclees y ejecutes este programa en tu máquina, deberías cambiar este nombre por el de una máquina de tu red. Esta es la máquina en la ejecutará `KnockKnockServer`.

Luego el cliente abre un stream de entrada y otro de salida sobre el socket.

Luego comienza el bucle que implementa la comunicación entre el cliente y el servidor. El servidor habla primero, por lo que el cliente debe escuchar, lo que hace leyendo desde el stream de entrada adosado al socket. Cuando el servidor habla, si dice "Bye,", el cliente sale del bucle. De otra forma muestra el texto en la salida estandard, y luego lee la respuesta del usuario, que la teclea en al entrada estandard. Después de que el usuario teclee el retorno de carro, el cliente envía el texto al servidor a través del stream de salida adosado al socket.

```
while ((fromServer = is.readLine()) != null) {
    System.out.println("Server: " + fromServer);
    if (fromServer.equals("Bye."))
        break;
    while ((c = System.in.read()) != '\n') {
        buf.append((char)c);
    }
    System.out.println("Client: " + buf);
```

```
    os.println(buf.toString());
    os.flush();
    buf.setLength(0);
}
```

La comunicación termina cuando el servidor pregunta si el cliente quiere escuchar otra broma, si el usuario dice no, el servidor dice "Bye."

En el interés de una buena limpieza, el cliente cierra sus streams de entrada y salida y el socket:

```
os.close();
is.close();
kkSocket.close();
```

Ejecutar los Programas

Primero se debe arrancar el programa servidor. Haz esto ejecutando el programa servidor utilizando el intérprete de Java, como lo haría con cualquier otro programa. Recuerda que debes ejecutarlo en la máquina que el programa cliente especifica cuando crea el socket.

Luego ejecutas el programa cliente. Observa que puedes ejecutarlo en cualquier máquina de tu red, no tiene porque ejecutarse en la misma máquina que el servidor.

Si es demasiado rápido, podría arrancar el cliente antes de que el cliente tuviera la oportunidad de inicializarse y empezar a escuchar el puerto. Si esto sucede verás el siguiente mensaje de error cuando intentes arrancar el programa cliente:

```
Exception: java.net.SocketException: Connection refused
```

Si esto sucede, intenta ejecutar el programa cliente de nuevo.

Verás el siguiente mensaje de error si se te olvidó cambiar el nombre del host en el código fuente del programa KnockKnockClient.

```
Trying to connect to unknown host: java.net.UnknownHostException: taranis
```

Modifica el programa KnockKnockClient y proporciona un nombre de host válido en tu red. Recompila el programa cliente e intentalo de nuevo.

Si intentas arrancar un segundo cliente mientras el primero está conectado al servidor, el segundo colgará. La [siguiente sección](#) le cuenta como soportar múltiples clientes.

Cuando obtengas una conexión entre el cliente y el servidor verás esto en tu pantalla:

```
Server: Knock! Knock!
```

Ahora, deberás responder con :

```
Who's there?
```

El cliente repite lo que has tecleado y envía el texto al servidor. El servidor responde con la primera línea de uno de sus varias bromas Knock Knock de su repertorio. Ahora tu pantalla debería contener esto (el texto que escribiste; está en negrita):

```
Server: Knock! Knock!
```

```
Who's there?
```

```
Client: Who's there?
```

```
Server: Turnip
```

Ahora deberías responderle con:

```
Turnip who?
```

De nuevo, el cliente repite lo que has tecleado y envía el texto al servidor. El servidor responde con la línea graciosa. Ahora tu pantalla debería contener esto (el texto que escribiste; está en negrita):

```
Server: Knock! Knock!
```

```
Who's there?
```

```
Client: Who's there?
```

Server: Turnip

Turnip who?

Client: Turnip who?

Server: Turnip the heat, it's cold in here! Want another? (y/n)

Si quieres oír otra broma teclea "y", si no, teclee "n". Si tecleas "y", el servidor empieza de nuevo con "Knock! Knock!". Si tecleas "n" el servidor dice "Bye.", haciendo que tanto el cliente como el servidor terminen.

Si en cualquier momento cometes un error al teclear, el objeto KKState lo captura, el servidor responde con un mensaje similar a este, y empieza la broma otra vez:

Server: You're supposed to say "Who's there?"! Try again. Knock! Knock!

El objeto KKState es particular sobre la ortografía y la puntuación, pero no sobre las letras mayúsculas y minúsculas.

Soportar Múltiples Clientes

El ejemplo KnockKnockServer fue diseñado para escuchar y manejar una sola petición de conexión. Sin embargo, pueden recibirse varias peticiones sobre el mismo puerto y consecuentemente sobre el mismo ServeSocket. Las peticiones de conexiones de clientes se almacenan en el puerto, para que el servidor pueda aceptarlas de forma secuencial. Sin embargo, puede servir las simultáneamente a través del uso de threads -- un thread para procesar cada conexión de cliente.

El flujo lógico básico en este servidor sería como este:

```
while (true) {  
    aceptar un a conexión;  
    crear un thread para tratar a cada cliente;  
end while
```

El thread lee y escribe en la conexión del cliente cuando sea necesario.

Intenta esto: Modifica el KnockKnockServer para que pueda servir a varios clientes al mismo tiempo. Aquí tienes nuestra solución, que está compuesta en dos clases: [KKMultiServer](#) y [KKMultiServerThread](#). KKMultiServer hace un bucle continuo escuchando peticiones de conexión desde los clientes en un ServerSocket. Cuando llega una petición KKMultiServer la acepta, crea un objeto KKMultiServerThread para procesarlo, manejando el socket devuelto por accept(), y arranca el thread. Luego el servidor vuelve a escuchar en el puerto las peticiones de conexión. El objeto KKMultiServerThread comunica con el cliente con el que está leyendo y escribiendo a través del socket. Ejecute el nuevo servidor Knock Knock y luego ejecute varios clientes sucesivamente.

¿Qué es un Datagrama?

Los clientes y servidores que se comunican mediante un canal fiable (como una URL o un socket) tienen un canal punto a punto dedicado entre ellos (o al menos la ilusión de uno). Para comunicarse, establecen una conexión, transmiten los datos y luego cierran la conexión. Todos los datos enviados a través del canal se reciben en el mismo orden en el que fueron enviados. Esto está garantizado por el canal.

En contraste, las aplicaciones que se comunican mediante datagramas envían y reciben paquetes de información completamente independientes. Estos clientes y servidores no tienen y no necesitan un canal punto a punto dedicado. El envío de los datos a su destino no está garantizado, ni su orden de llegada.

Definición: Un datagrama es un mensaje autocontenido independiente enviado a través de la red, cuya llegada, momento de llegada y contenido no está garantizado.

El paquete `java.net` contiene dos clases para ayudarte a escribir programas Java que utilicen datagramas para enviar y recibir paquetes a través de la red: `DatagramSocket` y `DatagramPacket`. Su aplicación envía y recibe `DatagramPackets` a través de un `DatagramSocket`.

También puedes ver

[java.net.DatagramPacket](#)
[java.net.DatagramSocket](#)

Escribir un Datagrama Cliente y Servidor

El ejemplo generado en esta sección está comprendido por dos aplicaciones: un cliente y un servidor. El servidor recibe continuamente paquetes de datagramas a través de un socket datagramas. Cada paquete recibido por el servidor indica una petición del cliente de una cita famosa. Cuando el servidor recibe un datagrama, le responde enviando un datagrama que contiene un texto de sólo una línea que contiene "la cita del momento" al cliente.

La aplicación cliente de este ejemplo es muy sencilla -- envía un datagrama al servidor que indica que le gustaría recibir una cita del momento. Entonces el cliente espera a que el servidor le envíe un datagrama en respuesta.

Dos clases implementan la aplicación servidor: QuoteServer y QuoteServerThread. Una sola clase implementa la aplicación cliente: QuoteClient.

Investiguemos estas clases empezando con la clase que contiene el método main() de la aplicación servidor.

Contiene una versión de applet de la clase QuoteClient.

La Clase QuoteServer

La clase QuoteServer contiene un sólo método: el método main() para la aplicación servidor de citas. El método main() sólo crea un nuevo objeto QuoteServerThread y lo arranca.

```
class QuoteServer {  
    public static void main(String[] args) {  
        new QuoteServerThread().start();  
    }  
}
```

El objeto QuoteServerThread implementa la lógica principal del servidor de citas.

La Clase QuoteServerThread

La clase [QuoteServerThread](#) es un Thread que se ejecuta continuamente esperando peticiones a través del un socket de datagramas.

QuoteServerThread tiene dos variables de ejemplar privadas. La primera, llamada socket, es una referencia a un objeto DatagramSocket object. Esta variable se inicializa a null. La segunda, qfs, es un objeto DataInputStream que se ha abierto sobre un fichero de texto ASCII que contiene una lista de citas. Cada vez que se llegue al servidor una petición de cita, el servidor recupera la siguiente línea desde el stream de entrada.

Cuando el programa principal crea el QuoteServerThread que utiliza el único constructo disponible:

```
QuoteServerThread() {  
    super("QuoteServer");  
    try {  
        socket = new DatagramSocket();
```

```

        System.out.println("QuoteServer listening on port: " +
socket.getLocalPort());
    } catch (java.net.SocketException e) {
        System.err.println("Could not create datagram socket.");
    }
    this.openInputFile();
}

```

La primera línea de este constructor llama al constructor de la superclase (Thread) para inicializar el thread con el nombre "QuoteServer". La siguiente sección de código es la parte crítica del constructor de QuoteServerThread -- crea un DatagramSocket. El QuoteServerThread utiliza este DatagramSocket para escuchar y responder las peticiones de citas de los clientes.

El socket es creado utilizando el constructor de DatagramSocket que no requiere argumentos:

```
socket = new DatagramSocket();
```

Una vez creado usando este constructor, el nuevo DatagramSocket se asigna a algún puerto local disponible. La clase DatagramSocket tiene otro constructor que permite especificar el puerto que quiere utilizar para asignarle el nuevo objeto DatagramSocket. Deberías observar que ciertos puertos están dedicados a servicios "bien-conocidos" y que no pueden ser utilizados. Si se especifica un puerto que está siendo utilizado, fallará la creación del DatagramSocket.

Después de crear con éxito el DatagramSocket, el QuoteServerThread muestra un mensaje indicando el puerto al que se ha asignado el DatagramSocket. El QuoteClient necesita este número de puerto para construir los paquetes de datagramas destinados a este puerto. Por eso, se debe utilizar este número de puerto cuando [ejecute el QuoteClient](#).

La última línea del constructor de QuoteServerThread llama a un método privado, openInputFile(), dentro de QuoteServerThread para abrir un fichero llamado [one-liners.txt](#) que contiene una lista de citas. Cada cita del fichero debe ser una línea en sí misma.

Ahora la parte interesante de [QuoteServerThread](#) -- es el método run(). (El método run() sobrescribe el método run() de la clase Thread y proporciona la implementación del thread. Para información sobre los Threads, puede ver [Threads de Control](#)).

El método run() QuoteServerThread primero comprueba que se ha creado un objeto DatagramSocket válido durante su construcción. Si socket es null, entonces el QuoteServerThread no podría desviar el DatagramSocket. Sin el socket, el servidor no puede operar, y el método run() retorna.

De otra forma, el método run() entra en un bucle infinito. Este bucle espera continuamente las peticiones de los clientes y responde estas peticiones. Este bucle contiene dos secciones críticas de código: la sección que escucha las peticiones y la que las responde, primero veremos la sección que recibe las peticiones:

```
packet = new DatagramPacket(buf, 256);
```

```
socket.receive(packet);  
address = packet.getAddress();  
port = packet.getPort();
```

La primera línea de código crea un nuevo objeto DatagramPacket encargado de recibir un datagrama a través del socket. Se puede decir que el nuevo DatagramPacket está encargado de recibir datos desde el socket debido al constructor utilizado para crearlo. Este constructor requiere sólo dos argumentos, un array de bytes que contiene los datos específicos del cliente, y la longitud de este array. Cuando se construye un DatagramPacket para enviarlo a través de un DatagramSocket, también debe suministrar la dirección de internet y el puerto de destino del paquete. Verás esto más adelante cuando expliquemos cómo responde un servidor a las peticiones del cliente.

La segunda línea de código recibe un datagrama desde el socket. La información contenida dentro del mensaje del datagrama se copia en el paquete creado en la línea anterior. El método receive() se bloquea hasta que se reciba un paquete. Si no se recibe ningún paquete, el servidor no hace ningún progreso y simplemente espera.

Las dos líneas siguientes obtienen la dirección de internet y el número de puerto desde el que se ha recibido el datagrama. La dirección Internet y el número de puerto indicado de donde vino el paquete. Este es donde el servidor debe responder. En este ejemplo, el array de bytes del datagrama no contiene información relevante. Sólo la llegada del paquete indica una petición por parte del cliente que puede ser encontrado en la dirección de Internet y el número de puertos indicados en el datagrama.

En este punto, el servidor ha recibido una petición de una cita desde un cliente. Ahora el servidor debe responder. Las seis líneas de código siguientes construyen la respuesta y la envían.

```
if (qfs == null)  
    dString = new Date().toString();  
else  
    dString = getNextQuote();  
dString.getBytes(0, dString.length(), buf, 0);  
packet = new DatagramPacket(buf, buf.length, address, port);  
socket.send(packet);
```

Si el fichero de citas no se puede abrir por alguna razón, qfs es null. En este caso, el servidor de citas sirve la hora del día en su lugar. De otra forma, el servidor de citas obtiene la siguiente cita del fichero abierto. La línea de código de la sentencia if convierte la cadena en un array de bytes.

La tercera línea de código crea un nuevo objeto DatagramPacket utilizado para enviar el mensaje a través del socket del datagrama.. Se puede decir que el nuevo DatagramPacket está destinado a enviar los datos a través del socket porque el constructor lo utiliza para eso. Este constructor requiere cuatro argumentos. El primer argumento es el mismo que el utilizado por el constructor utilizado para crear los datagramas receptores: un array de bytes que contiene el mensaje del emisor al receptor y la longitud de este array. Los dos siguientes argumentos son diferentes: una dirección de Internet y un número de puerto. Estos dos argumentos son la dirección completa del destino del datagrama y

debe ser suministrada por el emisor del datagrama.

La cuarta línea de código envía el DatagramPacket de esta forma.

El último método de interés de QuoteServerThread es el método `finalize()`. Este método hace la limpieza cuando el QuoteServerThread recoge la basura cerrando el DatagramSocket. Los puertos son recursos limitados y los sockets asignados a un puerto deben cerrarse cuando no se utilizan.

La Clase QuoteClient

La clase [QuoteClient](#) implementa una aplicación cliente para el QuoteServer. Esta aplicación sólo envía una petición al QuoteServer, espera una respuesta, y cuando ésta se recibe la muestra en la salida estandar. Echemos un vistazo al código.

La clase QuoteClient contiene un método -- el método `main()` para la aplicación cliente. La parte superior de `main()` declara varias variables locales para su utilización:

```
int port;
InetAddress address;
DatagramSocket socket = null;
DatagramPacket packet;
byte[] sendBuf = new byte[256];
```

La siguiente sección procesa los argumentos de la línea de comandos utilizados para invocar la aplicación QuoteClient.

```
if (args.length != 2) {
    System.out.println("Usage: java DatagramClient <hostname> <port#>");
    return;
}
```

Esta aplicación requiere dos argumentos: el nombre de la máquina en la que se está ejecutando QuoteServer, y el número de puerto por que el QuoteServer está escuchando. Cuando arranca el QuoteServer muestra un número de puerto. Este es el número de puerto que debe utilizar en la línea de comandos cuando arranque QuoteClient.

Luego, el método `main()` contiene un bloque `try` que contiene la lógica principal del programa cliente. Este bloque `try` contiene tres sección principales: una sección que crea un DatagramSocket, una sección que envía una petición al servidor, y una sección que obtiene la respuesta del servidor.

Primero veremos el código que crea un DatagramSocket:

```
socket = new DatagramSocket();
```

El cliente utiliza el mismo constructor para crear un DatagramSocket que el servidor. El DatagramSocket es asignado a cualquier puerto disponible.

Luego, el programa QuoteClient envía una petición al servidor:

```
address = InetAddress.getByName(args[0]);
port = Integer.parseInt(args[1]);
packet = new DatagramPacket(sendBuf, 256, address, port);
```

```
socket.send(packet);  
System.out.println("Client sent request packet.");
```

La primera línea de código obtiene la dirección Internet del host nombrado en la línea de comandos. La segunda línea de código obtiene el número de puerto de la línea de comandos. Estas dos piezas de información son utilizadas para crear un DatagramPacket destinado a esa dirección de Internet y ese número de puerto. La dirección Internet y el número de puertos deberían indicar la máquina en la que se arrancó el servidor y el puerto por el que el servidor está escuchando.

La tercera línea del código anterior crea un DatagramPacket utilizado para enviar datos. El paquete está construido con un array de bytes vacíos, su longitud, y la dirección Internet y el número de puerto de destino del paquete. El array de bytes está vacío porque este datagrama sólo pide la información del servidor. Todo lo que el servidor necesita saber para responder -- la dirección y el número de puerto donde responder -- es una parte automática del paquete.

Luego, el cliente obtiene una respuesta desde el servidor:

```
packet = new DatagramPacket(sendBuf, 256);  
socket.receive(packet);  
String received = new String(packet.getData(), 0);  
System.out.println("Client received packet: " + received);
```

Para obtener una respuesta del servidor, el cliente crea un paquete receptor y utiliza el método receive() del DatagramSocket para recibir la respuesta del servidor. El método receive() se bloquea hasta que un datagrama destinado al cliente entre a través del socket. Observa que si, por alguna razón, se pierde la respuesta del servidor, el cliente quedará bloqueado debido a la política de no garantías del modelo de datagrama. Normalmente, un cliente selecciona un tiempo para no estar esperando eternamente una respuesta -- si la respuesta no llega, el temporizador se cumple, y el servidor retransmite la petición.

Cuando el cliente recibe una respuesta del servidor, utiliza el método getData() para recuperar los datos del paquete. El cliente convierte los datos en una cadena y los muestra.

Ejecutar el Servidor

Después de haber compilado con éxito los programas cliente y servidor, puedes ejecutarlos. Primero debes ejecutar el servidor porque necesitas conocer el número de puerto que muestra antes de poder arrancar el cliente. Cuando el servidor asigna con éxito su DatagramSocket, muestra un mensaje similar a este:

```
QuoteServer listening on port: portNumber
```

portNumber es el número de puerto al que está asignado el DatagramSocket. Utiliza este número para arrancar el cliente.

Ejecutar el Cliente

Una vez que has arrancado el servidor y mostrado el mensaje que indica el puerto en el que está escuchando, puedes ejecutar el programa cliente. Recuerda ejecutar el programa cliente con dos argumentos en la línea de comandos: el nombre del host en el se está ejecutando el QuoteServer, y el número de puerto que mostró al arrancar.

Después de que el cliente envíe una petición y reciba una respuesta desde el servidor, deberías ver una salida similar a ésta:

```
Quote of the Moment: Life is wonderful. Without it we'd all be dead.
```

Proporcionar tu Propio Controlador de Seguridad

La seguridad se convierte en importante cuando se escriben programas que interactúan con Internet. ¿Te gustaría descargar algo que corrompiera tu sistema de ficheros? ¿Estarías abierto a un ataque de virus? Es imposible que los ordenadores en Internet estén completamente seguros del ataque de unos pocos villanos externos. Sin embargo, puedes seguir los pasos para proporcionar un nivel de protección significativa. Una de las formas que proporciona Java frente a los ataques es a través del uso de los controladores de seguridad. Un controlador de seguridad implementa e impone una política de seguridad para una aplicación.

Introducción al Controlador de Seguridad

El controlador de seguridad es un objeto del ámbito de la aplicación que determina qué operaciones potenciales deberían estar permitidas. Las clases de los paquetes Java cooperan con el controlador de seguridad pidiéndole permiso al controlador de seguridad de la aplicación para ejecutar ciertas operaciones.

Escribir un Controlador de Seguridad

Esta sección muestra una implementación sencilla de un controlador de seguridad que requiere que el usuario teclee una clave cada vez que la aplicación trata de abrir un fichero para leer o escribir.

Instalar tu Propio Controlador de Seguridad

Esta página muestra cómo poner a trabajar un controlador de seguridad en tus aplicaciones Java.

Nota: El controlador de seguridad de una aplicación sólo puede ser seleccionado una vez. Típicamente, un navegador selecciona su controlador de seguridad en el procedimiento de arrancada. Por eso, la mayoría de las veces, los applets no puede seleccionar un controlador de seguridad por que ya ha sido seleccionado. Ocurrirá una `SecurityException` si un applet intentas hacer esto. Puede ver [Entender las Capacidades y las Restricciones de los Applets](#) para más información.

Decidir los Métodos a Sobreescribir de la clase `SecurityManager`

Y finalmente, esta página mira la clase `SecurityManager` en mayor detalle, mostrándote qué métodos de esta clase que afectan a los distintos tipos de operaciones y ayuda a decidir qué métodos necesitará sobreescribir tu controlador de seguridad.

Introducción a los Controladores de Seguridad

Toda aplicación Java puede tener su propio objeto controlador de seguridad que actúa como un guardia de seguridad a tiempo completo. La clase `SecurityManager` del paquete `java.lang` es una clase abstracta que proporciona el interface de programación y una implementación parcial para todos los controladores de seguridad de Java.

Por defecto una aplicación no tiene controlador de seguridad. Esto es, el sistema de ejecución de Java no crea automáticamente un controlador de seguridad para cada aplicación. Entonces, por defecto, una aplicación permite todas las operaciones que están sujetas a las restricciones de seguridad.

Para cambiar este comportamiento indulgente, una aplicación debe crear e instalar su propio controlador de seguridad. Aprenderás como crear un controlador de seguridad en [Escribir un Controlador de Seguridad](#), y como instalarlo en [Instalar un Controlador de Seguridad](#).

Nota: Los navegadores existentes y los visualizadores de applets crean su propio controlador de seguridad cuando arrancan. Así, un applet está sujeto a las restricciones de acceso siempre que sean impuestas por el controlador de seguridad de una aplicación particular en la que el applet se está ejecutando.

Puedes obtener el controlador de seguridad actual de una aplicación utilizando el método `getSecurityManager()` de la clase `System`.

```
SecurityManager appsm = System.getSecurityManager();
```

Observa que `getSecurityManager()` devuelve `null` si no hay ningún controlador de seguridad actual en la aplicación por lo que debería asegurarse de que tiene un objeto válido antes de llamar a cualquiera de sus métodos.

Una vez que tienes el controlador de seguridad, puedes pedir permiso para permitir o prohibir ciertas operaciones. De hecho, muchas de las clases en los paquetes de Java hacen esto. Por ejemplo, el método `System.exit()`, que finaliza el interprete Java, utiliza el método `checkExit()` del controlador de seguridad para aprobar la operación de salida:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkExit(status);
}
...
// El código continúa aquí si checkedExit() retorna
```

Si el controlador de seguridad aprueba la operación de salida, el `checkExit()` retorna normalmente. Si el controlador de seguridad prohíbe la operación, el `checkExit()` lanza una `SecurityException`. De esta forma, el controlador de

seguridad permite o prohíbe una operación potencialmente dañina antes de que pueda ser completada.

La clase `SecurityManager` define muchos otros métodos utilizados para verificar otras clases de operaciones. Por ejemplo, el método `checkAccess()` verifica los accesos a los threads, y `checkPropertyAccess()` verifica el acceso a la propiedad especificada. Cada operación o grupo de operaciones tiene su propio método `checkXXX()`.

Además, el conjunto de métodos `checkXXX()` representa el conjunto de operaciones de las clases de los paquetes Java y el sistema de ejecución de Java que ya están sujetos a la protección del controlador de seguridad. Por eso, normalmente, sus código no tendrá que llamar a ningún método `checkXXX()` -- las clases de los paquetes de Java hacen esto por usted a un nivel lo suficientemente bajo que cualquier operación representada por un método `checkXXX()` ya está protegida. Sin embargo, cuando escribas tu propio controlador de seguridad, podrías tener que sobrescribir los métodos `checkXXX()` de `SecurityManager` para modificar la política de seguridad de las operaciones específicas, o podría tener que añadir algunos por ti mismo para poner otras clases de operaciones para el escurtinio del controlador de seguridad.

[Decidir los Métodos a Sobrescribir de SecurityManager](#) explica las operaciones o grupos de operaciones que cada método `checkXXX()` de la clase `SecurityManager` está diseñado para proteger.

Escribir un Controlador de Seguridad

Para escribir tu propio controlador de seguridad, debes crear una subclase de la clase `SecurityManager`. Esta subclase sobrescribe varios métodos de `SecurityManager` para personalizar las verificaciones y aprobaciones necesarias para una aplicación Java.

Esta página te lleva a través de un controlador de seguridad de ejemplo que restringe la lectura y escritura en el sistema de archivos. Para obtener la aprobación del controlador de seguridad, un método que abra un archivo para leer invoca uno de los métodos `checkRead()` de `SecurityManager`, un método que abra un archivo para escribir invoca a uno de los métodos `checkWrite()` de `SecurityManager`. Si el controlador de seguridad aprueba la operación el método `checkXXX()` retorna normalmente, de otra forma `checkXXX()` lanza una `SecurityException`.

Para imponer una política restrictiva en los accesos al sistema de archivos, nuestro ejemplo debe sobrescribir los métodos `checkRead()` y `checkWrite()` de `SecurityManager`. `SecurityManager` proporciona tres versiones de `checkRead()` y dos versiones de `checkWrite()`. Cada una de ellas debería verificar si la aplicación puede abrir un archivo para I/O. Una política implementada frecuentemente en los navegadores para que los applets cargados a través de la red no puedan leer o escribir en el sistema local de archivos a menos que el usuario lo apruebe,

La política implementada por nuestro ejemplo le pide al usuario una password cuando la aplicación intenta abrir un archivo para leer o escribir. Si la password es correcta se permite el acceso.

Todos los controladores de seguridad deben ser una subclase de `SecurityManager`. Así, nuestra [PasswordSecurityManager](#) descende de `SecurityManager`.

```
class PasswordSecurityManager extends SecurityManager {  
    . . .  
}
```

Luego, `PasswordSecurityManager` declara un ejemplar de la variable privada `password` para contener el password que el usuario debe introducir para poder permitir el acceso al sistema de archivos restringido. La password se selecciona durante la construcción:

```
PasswordSecurityManager(String password) {  
    super();  
    this.password = password;  
}
```

El siguiente método en la clase `PasswordSecurityManager` es un método de ayuda privado llamado `accessOK()`. Este método le pide al usuario una password y la verifica. Si el usuario introduce una password válida, el método devuelve `true`, de otra forma devuelve `false`.

```

private boolean accessOK() {
    int c;
    DataInputStream dis = new DataInputStream(System.in);
    String response;

    System.out.println("What's the secret password?");
    try {
        response = dis.readLine();
        if (response.equals(password))
            return true;
        else
            return false;
    } catch (IOException e) {
        return false;
    }
}

```

Finalmente al final de la clase PasswordSecurityManager hay tres métodos checkRead() y dos métodos checkWrite() sobreescritos:

```

public void checkRead(FileDescriptor filedescriptor) {
    if (!accessOK())
        throw new SecurityException("Not a Chance!");
}

public void checkRead(String filename) {
    if (!accessOK())
        throw new SecurityException("No Way!");
}

public void checkRead(String filename, Object executionContext) {
    if (!accessOK())
        throw new SecurityException("Forget It!");
}

public void checkWrite(FileDescriptor filedescriptor) {
    if (!accessOK())
        throw new SecurityException("Not!");
}

public void checkWrite(String filename) {
    if (!accessOK())
        throw new SecurityException("Not Even!");
}

```

Todos los métodos check**XXX**() llaman a accessOK() para pedirle al usuario la password. Si el acceso no es OK, entonces check**XXX**() lanza una SecurityException. De otra forma, check**XXX**() retorna normalmente. Observa que SecurityException es una excepción en tiempo de ejecución, y no necesita ser declarada en la clausula throws de estos métodos.

checkRead() y checkWrite() son sólo unos pocos de los muchos métodos

checkXXX() de SecurityManager que verifican varias clases de operaciones. Puedes sobrescribir o añadir cualquier número de método checkXXX() para implementar tu propia política de seguridad. No necesitas sobrescribir todos los métodos checkXXX() de SecurityManager, sólo aquellos que quieras personalizar. Sin embargo, la implementación por defecto proporcionada por la clase SecurityManager para todos los métodos checkXXX() lanza una SecurityException. En otras palabras, por defecto, la clase SecurityManager prohíbe todas las operaciones que están sujetas a las restricciones de seguridad. Por lo que podrías encontrar que tienes que sobrescribir muchos métodos checkXXX() para obtener el comportamiento deseado.

Todos los métodos checkXXX() de la clase SecurityManager operan de la misma forma:

- Si el acceso está permitido, el método retorna.
- Si el acceso no está permitido, el método lanza una SecurityException.

Asegurate de que implementas de esta forma tus métodos checkXXX() sobrescritos.

Bien, esta es nuestra subclase de SecurityManager. Como puedes ver implementar un controlador de seguridad es sencillo, sólo :

- Crea una subclase de SecurityManager.
- Sobreescribe unos cuantos métodos.

El truco está en determinar los métodos que se deben sobrescribir para implementar tu política de seguridad. [Decidir que Métodos Sobreescibir de SecurityManager](#) te ayudará a decidir los métodos que deberás sobrescribir dependiendo de los tipos de operaciones que quieras proteger. La [página siguiente](#) te enseña como instalar la clase PasswordSecurityManager como el controlador de seguridad de su aplicación Java.

También puede ver

[java.lang.SecurityManager](#)
[java.lang.SecurityException](#)

Instalar un Controlador de Seguridad

Una vez que has terminado de escribir tu subclase de `SecurityManager`, puedes instalarla como el controlador de seguridad actual de tu aplicación. Puedes hacer esto utilizando el método `setSecurityManager()` de la clase `System`.

Aquí tienes una pequeña aplicación de prueba, [SecurityManagerTest](#), que instala la clase `PasswordSecurityManager` de la [página anterior](#) como el controlador de seguridad actual. Luego, para verificar que el controlador de seguridad está en su lugar y es operacional, esta aplicación abre dos ficheros -- uno para leer y otro para escribir -- y copia el contenido del primero en el segundo.

El método `main()` comienza con la instalación de nuevo controlador de seguridad:

```
try {  
    System.setSecurityManager(new PasswordSecurityManager("Booga Booga"));  
} catch (SecurityException se) {  
    System.out.println("SecurityManager already set!");  
}
```

La línea en **negrita** del código anterior crea un nuevo ejemplar de la clase `PasswordSecurityManager` con la password "Booga Booga". Este ejemplar es pasado al método `setSecurityManager()` de la clase `System`, que instala el objeto como el controlador de seguridad por defecto para la aplicación que se está ejecutando. Este controlador de seguridad permanecerá efectivo durante el la ejecución de esta aplicación.

Sólo se puede seleccionar un vez el controlador de seguridad de una aplicación. En otras palabras, una aplicación Java sólo puede invocar una vez a `System.setSecurityManager()` durante su ciclo de vida. Cualquier intento posterior de instalar un controlador de seguridad dentro de una aplicación Java resultará en una `SecurityException`.

El resto del programa copia el contenido de este fichero [inputtext.txt](#) en un fichero de salida llamado `outputtext.txt`. Es sólo un texto que verifica que `PasswordSecurityManager` se ha instalado de forma apropiada.

```
try {  
    DataInputStream fis = new DataInputStream(new FileInputStream("inputtext.txt"));  
    DataOutputStream fos = new DataOutputStream(new  
FileOutputStream("outputtext.txt"));  
    String inputString;  
    while ((inputString = fis.readLine()) != null) {  
        fos.writeBytes(inputString);  
        fos.writeByte('\n');  
    }  
    fis.close();  
    fos.close();  
} catch (IOException ioe) {  
    System.err.println("I/O failed for SecurityManagerTest.");  
}
```

Las líneas en **negrita** del código anterior son los accesos al sistema de ficheros restringido. Estas llamadas a método resultan en una llamada al método `checkAccess()` del `PasswordSecurityManager`.

Ejecutar el Programa de Prueba

Cuando ejecutes la aplicación `SecurityManagerTest` te pedirá dos veces la password: una cuando la aplicación abre el fichero de entrada y otra cuando abre el fichero de salida. Si tecleas la password correcta, se permite el acceso -- el objeto fichero -- y la aplicación prosigue con la siguiente sentencia. Si tecleas una password incorrecta, `checkXXX()` lanza una `SecurityException`, que la aplicación no intenta capturar por lo

Decidir los Métodos a Sobreescibir del SecurityManager

Podrías tener que sobreescibir varios métodos `checkXXX()` del `SecurityManager` dependiendo de las operaciones a las que quieras que el controlador de seguridad les imponga restrcciones.

La primera columna de la siguiente tabla son objetos sobre los que se pueden realizar varias operaciones. La segunda columna lista los métodos de `SecurityManager` que aprueban las operaciones de los objetos de la primera columna.

Operaciones sobre	Aprobadas por
sockets	<code>checkAccept(String <i>host</i>, int <i>port</i>)</code> <code>checkConnect(String <i>host</i>, int <i>port</i>)</code> <code>checkConnect(String <i>host</i>, int <i>port</i>, Object <i>executionContext</i>)</code> <code>checkListen(int <i>port</i>)</code>
threads	<code>checkAccess(Thread <i>thread</i>)</code> <code>checkAccess(ThreadGroup <i>threadgroup</i>)</code>
class loader	<code>checkCreateClassLoader()</code>
sistema de ficheros	<code>checkDelete(String <i>filename</i>)</code> <code>checkLink(String <i>library</i>)</code> <code>checkRead(FileDescriptor <i>filedescriptor</i>)</code> <code>checkRead(String <i>filename</i>)</code> <code>checkRead(String <i>filename</i>, Object <i>executionContext</i>)</code> <code>checkWrite(FileDescriptor <i>filedescriptor</i>)</code> <code>checkWrite(String <i>filename</i>)</code>
comandos del sistema	<code>checkExec(String <i>command</i>)</code>
interprete	<code>checkExit(int <i>status</i>)</code>
paquete	<code>checkPackageAccess(String <i>packageName</i>)</code> <code>checkPackageDefinition(String <i>packageName</i>)</code>
propiedades	<code>checkPropertiesAccess()</code> <code>checkPropertyAccess(String <i>key</i>)</code> <code>checkPropertyAccess(String <i>key</i>, String <i>def</i>)</code>
networking	<code>checkSetFactory()</code>
windows	<code>checkTopLevelWindow(Object <i>window</i>)</code>

Dependiendo de tu política de seguridad, puedes sobreescibir algunos o todos estos métodos. Por ejemplo, supon que estás escribiendo un Navegador Web o un visualizador de applets y quieres evitar que los applets utilicen sockets. Puedes hacer esto sobreescibiendo los cuatro métodos que afectan al acceso a los sockets.

Muchos de los métodos `checkXXX()` son llamados en múltiples situaciones. Ya viste esto cuando escribimos el `PasswordSecurityManager` en [Escribir un Controlador de Seguridad](#) -- el método `checkAccess(ThreadGroup g)` es llamado cuando se crea un `ThreadGroup`, selecciona su estado de servicio, lo para, etc.

Cuando sobreescribas un método `checkXXX()` asegurate de que comprendes todas las situaciones en las que puede ser llamado.

La implementación por defecto suministrada por la clase `SecurityManager` para todos los métodos `checkXXX()` es:

```
public void checkXXX(. . .) {  
    throw new SecurityException();  
}
```

La mayoría de las veces querráa que haga algo más selectivo que prohibirlo todo! Por eso podrías encontrar que debes sobresscribir todos los métodos `checkXXX()` de `SecurityManager`.

Ozito

Cambios en el JDK 1.1: Trabajo en Red y Seguridad

En el JDK 1.1 se ha aumentado el rendimiento del paquete de red: se ha añadido soporte para opciones de estilo BSD en los sockets, las clases Socket and ServerSocket ya no son finales y se pueden extender, se han añadido nuevas subclases de SocketException para una mejor informe y manejo de errores de red, se ha añadido soporte para multitipado. El JDK 1.1 también incluye un aumento de rendimiento general y corrección de errores.

Para finalizar, se han añadido estas clases e interfaces al paquete java.net:

Nuevas clases:

DatagramSocketImpl
URLConnection
MulticastSocket

Nuevas Clases de Excepciones:

BindException
ConnectionException
NoRouteToHostException

Nuevo Interface:

FileNameMap

Para más información sobre los cambios en las clases existente puedes ver las notas en las siguientes lecciones:

[Trabajar con URLs](#) ([Notas 1.1](#))

Se ha hecho un cambio menor en la clase URLConnection y se han corregido algunos errores que había en esta clase.

[Todo sobre los Sockets](#) ([Notas 1.1](#))

Se ha aumentado el rendimiento del paquete de red.

[Todo sobre los Datagramas](#) ([Notas 1.1](#))

Se han añadido varios métodos a DatagramPacket y DatagramSocket.

[Proporcionar tu propio controlador de Seguridad](#)

La versión del JDK 1.1 contiene un nuevo conjunto de APIs de seguridad en el paquete java.security y sus sub-paquetes. Puedes ver una nueva ruta [Seguridad Java 1.1 \[PENDIENTE\]](#), para más información sobre los nuevos APIs de seguridad.

JavaBeans: Componentes de la Plataforma Java

Los JavaBeans traen la tecnología de componentes a la Plataforma Java. Se puede utilizar el API JavaBeans para escribir clases Java, conocidas como Beans, que se pueden manipular visualmente con herramientas visuales.

JavaBeans es una capacidad del JDK1.1. Cualquier navegador o herramienta que soporte JDK1.1 soporta implícitamente los JavaBeans.

Este documento es una guía de mano para conocer los JavaBeans y el Kit de Desarrollo de Beans (BDK). La [Especificación del API JavaBeans](#) proporciona una descripción completa de los JavaBeans. Imprime una copia de esta especificación, y mantenla cerca mientras lees este documento.

El software que se necesita para aprender sobre los JavaBeans está disponible en la web. Además del [Kit de Desarrollo de Beans](#) (BDK), necesitaremos el [Java Development Kit](#) (JDK).

[Conceptos sobre los JavaBeans y el BDK](#) describe lo que hace un Bean, y que es BDK,

[Utilizar el BeanBox](#) describe las operaciones básicas de BeanBox, y explica sus menús.

[Escribir un Bean Sencillo](#) enseña como crear un Bean rudimentario, como guardarlo, como añadirlo a ToolBox situando el Bean dentro de BeanBox, como inspeccionar las propiedades y los eventos del Bean, y como generar un informe de introspección de un Bean.

[Propiedades](#) explica cómo dar propiedades a los Beans: Las características de apariencia y comportamiento del Bean personalizables durante el diseño.

[Manipular Eventos en el BeanBox](#) describe las capacidades de manipulación de eventos del BeanBox. Si no estás familiarizado con el manejo de eventos deberías leer [Mecanismo de eventos del JDK 1.1](#) para preparar este material.

[El interface BeanInfo](#) describe como escribir clases de información de Beans: Separa las clases que se pueden utilizar explícitamente avisando de las propiedades, los métodos y los eventos del Bean a la herramienta visual.

[Personalizar Beans](#) presenta las propiedades de editor, y el interface Customizer .

[Persistencia de un Bean](#) explica como guardar y restaurar nuestros Beans, y sus estados personalizados.

[Nuevas Características](#) describe el Marco de Trabajo "Java Activation", BeanContext, y el "drag and drop" nativo.

Documentación Adicional

El directorio beans/docs del BDK contiene esta información:

- El API Beans
- El API BeanBox
- Beans de ejemplo
- El API java.util
- Ficheros (JAR) y sus manifiestos
- Makefiles para gnumake (Unix) y nmake (Windows)

Un buen punto de entrada es el fichero beans/README.html.

La página [Documentación JavaBeans](#) contiene las definiciones actuales del API JavaBeans, incluyendo las descripciones de las características de los JavaBeans, y documentación relacionada como el API Reflection, Serialización del Objetos, Invocación de Métodos Remotos (RMI), y una lista de libros de JavaBeans de terceras partes.

Ozito

Conceptos sobre los JavaBeans

El API JavaBeans permite escribir componentes software en Java. Los componentes son unidades software reutilizables y auto-contenidas que pueden ser unirse visualmente en componentes compuestos, applets, aplicaciones y servlets utilizando herramientas visuales de desarrollo de aplicaciones.

Los componentes JavaBean son conocidos como Beans. Una herramienta de desarrollo que soporte JavaBeans, mantiene los Beans en un paleta o caja de herramientas. Se puede seleccionar un Bean de la paleta, arrastarlo dentro de un formulario, modificar su apariencia y su comportamiento, definir su interacción con otros Beans, y componer un applet, una aplicación, o un nuevo Bean, junto con otros Beans. Todo esto se puede hacer sin escribir una línea de código.

La siguiente lista describe brevemente los conceptos clave de los Beans:

- Las herramientas de desarrollo descubren las características de un Bean (esto es, sus propiedades, sus métodos y sus eventos) mediante un proceso conocido como introspección. Los Beans soportan la introspección de dos formas:
 - Adheriéndose a las convenciones específicas de nombres conocidas como patrones de nombrado, cuando se nombran las características del Bean. La clase [java.beans.Introspector](#) examina el Bean buscando esos patrones de diseño para descubrir las características del Bean. La clase Introspector se encuentra en el API [core reflection](#).
 - Proporcionando explícitamente información sobre la propiedad, el método o el evento con una clase Bean Information relacionada. Esta clase implementa el interface BeanInfo. Una clase BeanInfo lista explícitamente aquellas características del Bean que están expuestas a la herramienta de desarrollo.

Puedes ver el capítulo 8 de [Especificaciones del API JavaBeans](#) para una explicación sobre la introspección, los patrones de diseño y BeanInfo.

- **Propiedades** Son las características de apariencia y comportamiento de un Bean que pueden ser modificadas durante el diseño. Las propiedades se exponen a las herramientas de desarrollo mediante los patrones de diseño o una clase BeanInfo. Puedes ver el capítulo 7 de la Especificación del API JavaBeans para una completa explicación.
- Los Beans exponen sus propiedades para poder ser personalizados durante el diseño. La personalización se soporta de dos formas: utilizando editores de propiedades, o utilizando personalizadores de Beans más sofisticados. Puedes ver el capítulo 9 de la Especificación del API JavaBeans para una explicación completa.
- Los Beans utilizan los eventos para comunicarse con otros Beans. Un Bean que quiere recibir eventos (un Bean oyente) registra su interés con un Bean

que lanza eventos (un Bean fuente). Las herramientas de desarrollo pueden examinar un Bean para determinar que eventos puede disparar (enviar) y cuales puede manejar (recibir). Puedes ver el Capítulo 6 de la Especificación del API JavaBeans para una explicación completa.

- La Persistencia permite a los Beans guardar su estado, y restaurarlo posteriormente. Una vez que se han cambiado las propiedades de Bean, se puede guardar su estado y restaurar el Bean posteriormente. Los JavaBeans utilizan la Serialización de Objetos Java para soportar la Persistencia. Puedes ver el capítulo 5 de la Especificación del API JavaBeans para una explicación completa.
- Los métodos de un Bean no son diferentes de los métodos Java, y pueden ser llamados desde otros Beans o desde un entorno de scripts. Por defecto, todos los métodos públicos son exportados.

Aunque los Beans han sido diseñados para ser entendidos por herramientas de desarrollo, todas las claves del API, incluyendo el soporte para eventos, las propiedades y la persistencia, han sido diseñadas para ser fácilmente entendibles por los programadores humanos.

Contenidos del BDK

Aquí tienes una descripción general de los ficheros y directorios del BDK:

- README.html Contiene un punto de entrada a la documentación del BDK.
- LICENSE.html Contiene las condiciones de licencia de BDK.
- GNUmakefile y Makefile son ficheros Make Unix y Windows (.gmk and .mk suffixes) para construir los demos y el BeanBox y para ejecutar el BeanBox
- beans/apis contiene
 - java un directorio que contiene ficheros fuentes de JavaBeans.
 - sun un directorio que contiene ficheros fuente del editor de propiedades.
- beans/beanbox contiene
 - makefiles para construir el BeanBox
 - scripts para ejecutar BeanBox
 - classes un directorio que contiene las clase de BeanBox
 - lib un directorio que contiene fichero Jar de soporte de BeanBox utilizado por MakeApplet para producir código.
 - sun y sunw directorios que contienen los ficheros fuente de BeanBox.
 - tmp un directorio que contiene ficheros de clases generados automáticamente, y ficheros de applets generados automáticamente por MakeApplet.
- beans/demos contiene
 - makefiles para construir los Beans de ejemplo
 - un directorio html que contiene demostraciones de los applets que deben ser ejecutadas en appletviewer, HotJava, o navegadores que soporten JDK1..
 - sunw un directorio que contiene
 - wrapper un directorio que contiene un Bean de un applet
 - demos un directorio que contiene los ficheros fuentes de las demos
- beans/doc contiene
 - documentación de las demos
 - un directorio javadoc que contiene JavaBeans y clases relacionadas y un interface de documentación.
 - documentación miscelanea.
- beans/jars contiene ficheros Jar para los Beans de ejemplo.

Utilizar BeanBox

El BeanBox es un contenedor de Beans. Se pueden escribir Beans, luego arrastrarlos dentro de BeanBox para ver si funcionan como se esperaba. El BeanBox también sirve como una demostración de cómo deben comportarse las herramientas de desarrollo compatibles con los JavaBeans. El BeanBox también es una buena herramienta para aprender sobre los Beans. En esta sección aprenderás como utilizar BeanBox.

[Arrancar y Utilizar BeanBox](#) explica como arrancar BeanBox, y describe la caja de herramientas y la hoja de propiedades.

[Los Menús de BeanBox](#) explica cada uno de los ítems de los menús de BeanBox.

[Utilizar BeanBox para Generar Applets](#) demuestra la facilidad de BeanBox para generar applets.

Ozito

Arrancar y Utilizar BeanBox

El directorio beans/beanbox contiene scripts para Windows (run.bat) y Unix (run.sh) que arrancan BeanBox. Se pueden utilizar estos comandos para arrancarlo o utilizar:

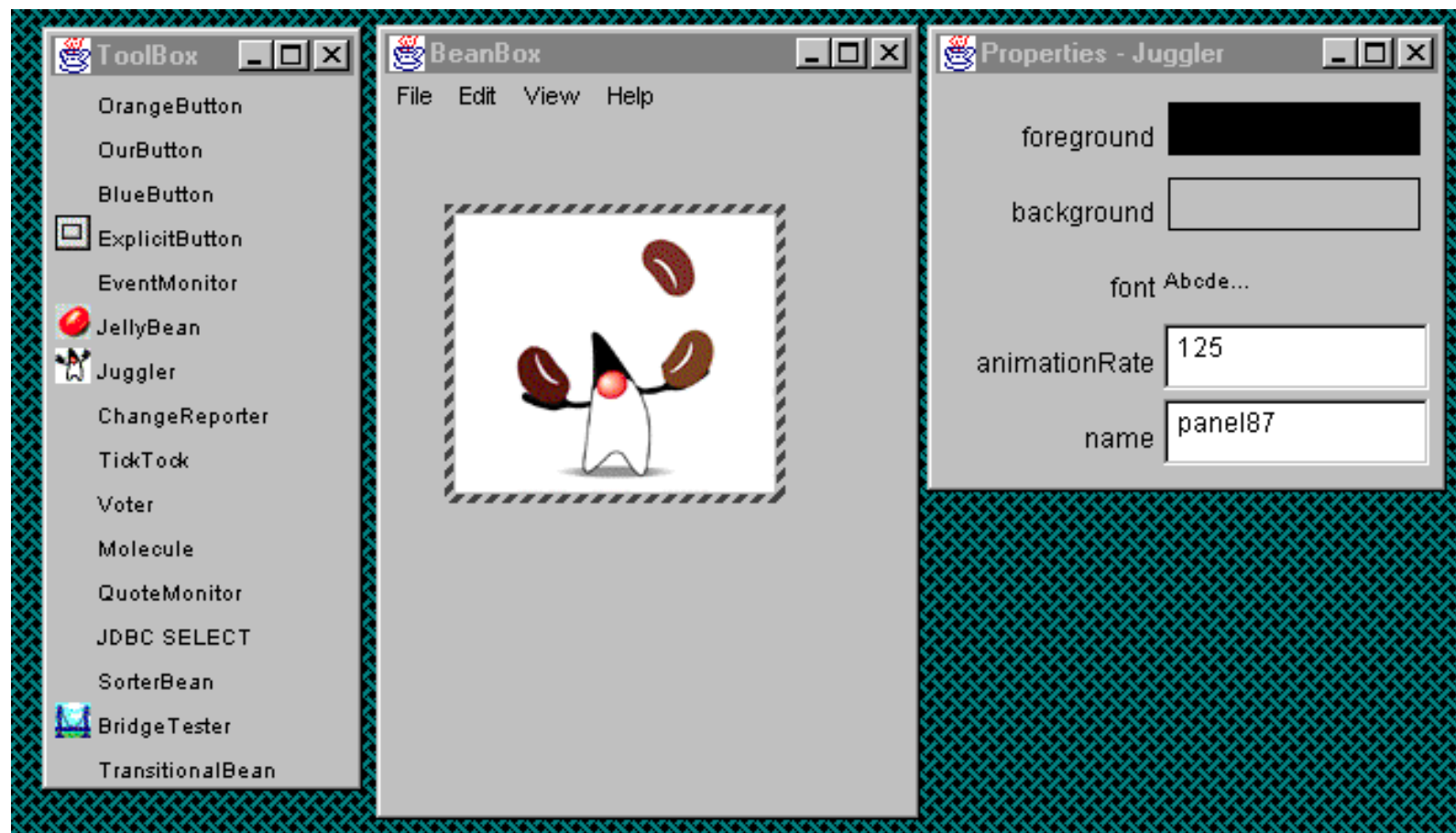
`gnumake run`

O:

`nmake run`

Puedes ver los ficheros del BDK beans/doc/makefiles.html y beans/doc/gnu.txt para información sobre cómo obtener copias de estas herramientas.

Cuando se arranca, el BeanBox muestra tres ventanas: BeanBox, ToolBox, y la Hoja de Propiedades. Aquí puedes ver su aspecto:



El ToolBox contiene los Beans disponibles para utilizar con el BeanBox. Se pueden añadir Beans al ToolBox. El BeanBox es el área de trabajo; se seleccionan los Beans de ToolBox y se arrastran hacia el BeanBox para poder trabajar sobre ellos. La hoja de propiedades muestra las propiedades del Bean seleccionado actualmente dentro del BeanBox.

La ilustración muestra el Bean Juggler dentro del BeanBox. El recuadro alrededor del Bean significa que está seleccionado. Se selecciona pulsando sobre el Bean dentro del BeanBox. La hoja de propiedades muestra las propiedades de Juggler.

Añadir un Bean a ToolBox

Cuando BeanBox arranca, carga automáticamente el ToolBox con los Beans que encuentre dentro de los ficheros JAR contenidos en el directorio beans/jars. Mover los ficheros JAR a este directorio hace que sean cargados automáticamente cuando arranca BeanBox. Se pueden cargar los Beans desde ficheros JAR localizados en cualquier directorio utilizando el menú File|LoadJar... de BeanBox.

Arrastrar un Bean al BeanBox

Pulsar sobre el nombre de un Bean dentro de ToolBox lo elige para situarlo dentro del BeanBox. Para arrastrar un ejemplar de JellyBean al BeanBox:

1. Pulsa sobre la palabra JellyBean en ToolBox. El cursor cambia a un cruce de ejes.
2. Pulsa dentro de BeanBox. El JellyBean aparecerá y será seleccionado.

Observa el cambio en la hoja de propiedades cuando pongas JellyBean en el BeanBox. Antes de situar JellyBean en el BeanBox, se mostraba la hoja de propiedades de BeanBox; después de situarlo, se mostrará la hoja de propiedades de JellyBean. Si no has visto el cambio, pulsa dentro del BeanBox, fuera de JellyBean. Esto seleccionará el BeanBox en vez de JellyBean. La hoja de propiedades mostrará entonces las propiedades del BeanBox.

Después de arrastrar un ejemplar de JellyBean al BeanBox, la hoja de propiedades mostrará las propiedades de JellyBean: color, foreground, priceInCents, background, y font.

Editar las Propiedades de un Bean

La hoja de propiedades muestra cada nombre de propiedad y su valor actual. Los valores se muestran en campos de texto editables (Strings y números), menús choice (Booleanos) o como valores de dibujo (Colores y Fuentes). Cada propiedad tiene un editor de propiedad asociado. Pulsar sobre una propiedad en la hoja de propiedades activa el editor de esa propiedad. Las propiedades mostradas en campos de texto o menús choice se editan dentro de la hoja de propiedades.

Como editar sus valores necesita unos interfaces de usuario más sofisticados, los colores y las fuentes utilizan un editor de propiedades personalizado. Cuando se pulsa sobre una propiedad de color o de fuente se lanza un panel separado para editarlo. Intenta pulsar en todas las propiedades de JellyBean.

Guardar y Restaurar Beans

El BeanBox utiliza Serialización de Objetos Java para grabar y restaurar Beans y sus estados. Los siguientes pasos demuestran como grabar y restaurar un Bean:

1. Arrastra un JellyBean al BeanBox.
2. Cambia la propiedad color al color que quieras.
3. Selecciona el menú File|Save de BeanBox. Aparecerá un navegador de ficheros, utilízala para grabar el Bean en un fichero.
4. Selecciona el menú File|Clear de BeanBox.
5. Selecciona el menú File|Load de BeanBox. Aparecerá de nuevo el navegador de ficheros; utilízalo para recuperar el Bean serializado.

Los menús de BeanBox

Esta página explica cada uno de los ítems de los menús de BeanBox File, Edit y View.

Menú File

Save

Guarda los Beans que hay en el BeanBox, incluyen la posición, el tamaño y el estado interno de cada Bean. El fichero guardado puede ser cargado mediante File|Load.

SerializeComponent...

Guarda los Beans que hay en el BeanBox en un fichero serializado (.ser). Este fichero debe ponerse en un fichero .jar para ser utilizable.

MakeApplet...

Genera un applet a partir del contenido del BeanBox.

Load...

Carga ficheros guardados en el BeanBox. No cargará ficheros .ser .

LoadJar...

Carga el contenido de ficheros JAR en el ToolBox.

Print

Imprime el contenido del BeanBox.

Clear

Elimina el contenido del BeanBox.

Exit

Sale de BeanBox sin ofrecer el guardado de los Beans.

Menú Edit

Cut

Elimina el Bean seleccionado en el BeanBox. El Bean cortado es serializado, y puede ser pegado.

Copy

Copia el Bean Seleccionado en el BeanBox. El Bean copiado es serializado y puede ser pegado.

Paste

Pega el último Bean cortado o copiado en el BeanBox.

Report...

Genera un informe de introspección sobre el Bean seleccionado.

Events

Lista los eventos disparados por el Bean seleccionado, agrupados por interfaces.

Bind property...

Lista todos los métodos de propiedades encontrados en el Bean seleccionado.

MenúView

Disable Design Mode

Elimina de la pantalla las ventanas del ToolBox y la hoja de propiedades. Elimina todo el comportamiento de diseño y prueba del BeanBox (los beans seleccionados) y los convierte en una aplicación.

Hide Invisible Beans

Ocultar los Beans invisibles.

Ozito

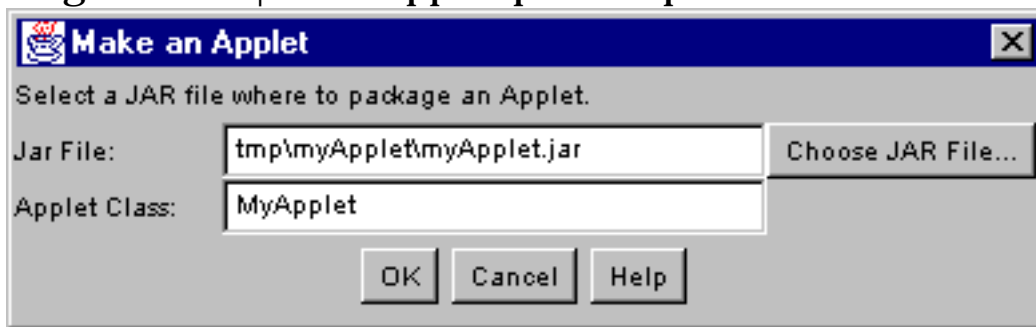
Utilizar BeanBox para Generar Applets

Se puede utilizar la opción File|MakeApplet... del menú del BeanBox para generar un applet a partir de los contenidos del BeanBox. Al hacer esto se crea:

- Un fichero JAR que contiene los ficheros de clases y los datos serializados.
- Un fichero de prueba HTML que utiliza el fichero JAR (y cualquier otro fichero JAR necesario):
- Un subdirectorio con los ficheros fuente Java y Makefile.
- Un fichero readme con información completa sobre el applet generado y los ficheros involucrados.

Seguiremos estos pasos para generar un applet desde BeanBox:

1. Utilizaremos el ejemplo "Juggler" que creamos en la página [eventos](#). Si hemos guardado el ejemplo en un fichero, lo cargaremos en el BeanBox utilizando el menú File|Load. Si ni lo grabamos, seguiremos los pasos de la página de eventos para construir el ejemplo. El applet generado tiene el mismo tamaño que el marco del BeanBox, por lo tanto deberíamos ajustar el tamaño del BeanBox al tamaño que queramos para el applet.
2. Elegimos File|Make Applet para disparar la ventana de diálogo MakeApplet:

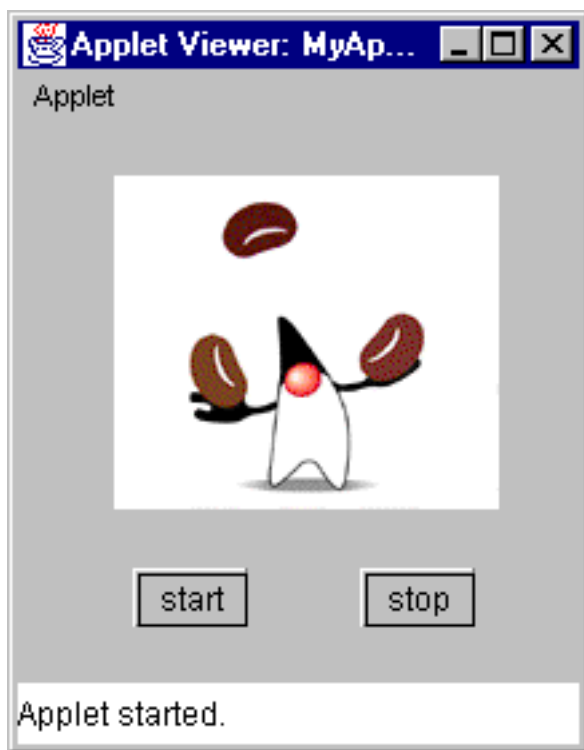


Utilizaremos el nombre para el applet y el fichero JAR por defecto para este ejemplo. Pulsamos el botón OK.

Esto es todo. Los ficheros generados se han situado en el directorio beanbox/tmp/myApplet. Se puede inspeccionar nuestro trabajo llamando al appletviewer de esta forma:

```
appletviewer <BDKInstallation>/beanbox/tmp/myApplet.html.
```

Esto es lo que verás:



No olvides echar un vistazo al fichero myApplet_readme, y a los otros ficheros generados.

Los applets generados pueden utilizarse en cualquier navegador compatible con Java 1.1. El AppletViewer es una buena plataforma de prueba. Otro navegador compatible es el [HotJava](#). La versión beta 3 del IE 4.0 no soporta ficheros JAR, y deberás expandir los ficheros JAR y HTML generados. También, un bug en la deserialización de componentes hace que estos no escuchen los eventos del ratón. Puedes ver el fichero readme necesario para más información. El applet generado no funcionará en Netscape Communicator versiones 4.0 y 4.1.

Ozito

Escribir un Bean Sencillo

En ésta página aprenderemos algo más sobre los Beans y BeanBox:

- Creando un Bean sencillo.
- Compilando y guardando el Bean en un archivo JAR.
- Cargaando el Bean en el ToolBox
- Arrastrando un ejemplar del Bean dentro del BeanBox.
- Inspeccionando las propiedades del Beans, los métodos y los eventos.
- Generando un informe de introspección.

Nuestro Bean se llamará SimpleBean. Aquí tienes los pasos para crearlo y verlo en el BeanBox:

1. Escribir el código de SimpleBean. Ponerlo en un fichero llamado SimpleBean.java, en cualquier directorio. Aquí tienes el código:

```
import java.awt.*;
import java.io.Serializable;

public class SimpleBean extends Canvas
    implements Serializable{

    //Constructor sets inherited properties
    public SimpleBean(){
        setSize(60,40);
        setBackground(Color.red);
    }
}
```

SimpleBean descende del componente [java.awt.Canvas](#). También implementa el interface [java.io.Serializable](#), que es requerido por todos los beans. Seleccionar el color del fondo y el tamaño del componente es todo lo que hace SimpleBean.

2. Asegurate de que la variable de entorno CLASSPATH apunta a todos ficheros .class (o .jar) necesarios.
3. Compila el Bean:

```
javac SimpleBean.java
```

Esto produce el fichero de clase SimpleBean.class

4. Crea un fichero de manifiesto. Utiliza tu editor de texto favorito para crear un fichero, que llamaremos manifest.tmp, que contenga el siguiente texto:

```
Name: SimpleBean.class
```



```
Java-Bean: True
```

5. Crea el fichero JAR. Este fichero contendrá el manifiesto y el fichero de clase SimpleBean:

```
jar cfm SimpleBean.jar manifest.tmp SimpleBean.class
```

6. Carga el fichero JAR en el ToolBox. Despliega el menú File|LoadJar... Esto traerá un navegador de ficheros. Busca la posición del fichero SimpleBean.jar y selecciónalo. SimpleBean aparecerá en la parte inferior del ToolBox. (Observa que cuando arranca el BeanBox, todos los beans que haya en el directorio beans/jars se cargan automáticamente en el ToolBox).
7. Arrastra un ejemplar de SimpleBean dentro del BeanBox. Pulsa sobre la palabra SimpleBean en el ToolBox. El cursor cambia a un punto de mira. Mueve el cursor al punto del BeanBox donde quieres situar el Bean y pulsa de nuevo el ratón. SimpleBean aparecerá como un rectángulo pintado con un borde marcado. Este borde significa que está seleccionado. Las propiedades de SimpleBean aparecerán en la hoja de propiedades.

Se puede redimensionar el SimpleBean, ya que descende de Canvas, arrastrando una esquina de la ventana. Verás como el cursor cambia a un ángulo recto cuando pasa sobre una esquina. También se puede reposicionar dentro del BeanBox, arrastrando desde cualquier parte del borde que no sea una esquina. Verás que el cursor cambia a una flechas cruzadas cuando pasa por estas posiciones.

Makefiles de SimpleBean

Abajo tienes dos MakeFiles (Unix y Windows) configurados para crear SimpleBean.

```
# gnumake file

CLASSFILES= SimpleBean.class

JARFILE= SimpleBean.jar

all: $(JARFILE)

# Create a JAR file with a suitable manifest.
$(JARFILE): $(CLASSFILES) $(DATAFILES)
    echo "Name: SimpleBean.class" >> manifest.tmp
    echo "Java-Bean: True" >> manifest.tmp
    jar cfm $(JARFILE) manifest.tmp *.class
    @/bin/rm manifest.tmp
```

```
# Compile the sources
%.class: %.java
    export CLASSPATH; CLASSPATH=. ; \
    javac $<

# make clean
clean:
    /bin/rm -f *.class
    /bin/rm -f $(JARFILE)
```

Aquí tienes la versión Windows de nmake:

```
# nmake file
CLASSFILES= simplebean.class

JARFILE= simplebean.jar

all: $(JARFILE)

# Create a JAR file with a suitable manifest.

$(JARFILE): $(CLASSFILES) $(DATAFILES)
    jar cfm $(JARFILE) << manifest.tmp *.class
Name: SimpleBean.class
Java-Bean: True
<<

.SUFFIXES: .java .class

{sunw\demo\simple}.java{sunw\demo\simple}.class :
    set CLASSPATH=.
    javac $<

clean:
    -del sunw\demo\simple\*.class
    -del $(JARFILE)
```

Se pueden utilizar estos makefiles como plantillas para crear los tuyos propios. Los makefiles de ejemplo, situados en el directorio beans/demo también te enseñan como utilizar los makefiles para construir y mantener tus Beans.

Inspeccionar las Propiedades y Eventos de SimpleBean

La hoja de propiedades muestra las propiedades del Bean seleccionado. Si seleccionamos SimpleBean, la hoja de propiedades mostrará cuatro propiedades: foreground, background, font, y name. Nosotros no declaramos propiedades en SimpleBean (lo verás más adelante), por eso estas propiedades son heredadas de Canvas. Pulsando sobre cada propiedad se lanza un editor de propiedad. El BeanBox proporcionar editores por defecto para los tipos primitivos, además de los tipos Font y Color. Puedes encontrar los fuentes para estos editores de propiedades en `beans/apis/sun/beans/editors`.

Los Beans se comunican unos con otros enviando y recibiendo notificaciones de eventos. Para ver los eventos que SimpleBean puede enviar, elige el menú Edit|Events. Se mostrará una lista de eventos agrupados por interfaces. Bajo cada grupo de interface hay una lista de métodos de evento. Todos estos son heredados de Canvas.

Podrás aprender más sobre [propiedades](#) y [eventos](#) en próximas secciones.

Generar un Informe de Introspección del Bean

La introspección es el proceso de descubrir las características de un Bean en tiempo de diseño por uno de estos dos métodos:

- Reflexión de bajo nivel, que utiliza patrones de diseño para descubrir las características del Bean.
- Examinando una clase asociada de información del Bean que describe explícitamente las características del Bean.

Se puede generar un informe de introspección eligiendo el menú Edit|Report. El informe lista los eventos, las propiedades y los métodos del Bean además de sus características.

Por defecto los informes son enviados a la salida estándar del intérprete Java, que es la ventana donde has arrancado BeanBox. Se puede redireccionar el informe a un fichero cambiando el comando del intérprete Java en `beanbox/run.sh` o `run.bat`:

```
java sun.beanbox.BeanBoxFrame > beanreport.txt
```

Propiedades

En las siguientes páginas aprenderemos cómo implementar las propiedades de los Beans.

Ahora es un buen momento para leer y revisar la [Especificación del API de los JavaBeans](#). El Capítulo 7 describe las propiedades.

- [Propiedades Sencillas](#) explica como dar propiedades a un Bean: las características de apariencia y comportamiento de un Bean en tiempo de diseño.
 - [Propiedades Límite](#) describe como un cambio en una propiedad puede proporcionar una notificación de cambio a otros objetos.
 - [Propiedades Restringidas](#) describe como los cambios en algunas propiedades pueden ser ocultados para otros objetos.
 - [Propiedades Indexadas](#) describe las propiedades multi-valor.
-

Propiedades Sencillas

Para obtener el mejor resultado de esta sección, deberías leer primero los capítulos 7 y 8, de la [Especificación del API JavaBean](#).

Si creamos una clase Bean, le damos una variable de ejemplar llamada `color`, y accedemos a `color` a través de un método obtenedor llamado `getColor` y un método seleccionador llamado `setColor`, entonces hemos creado una propiedad.

Las propiedades son aspectos de la apariencia o comportamiento de un Bean que pueden ser modificados en el momento del diseño.

Los nombres de los métodos para obtener o seleccionar propiedades deben seguir unas reglas específicas, llamadas patrones de diseño. Utilizando estos nombres de métodos basados en patrones de diseño, las herramientas compatibles con JavaBeans (y el BeanBox) pueden:

- Descubrir las propiedades de un Bean.
- Determinar los atributos de lectura/escritura de las propiedades.
- Determinar los tipos de propiedades.
- Localizar un editor apropiado para cada tipo de propiedad.
- Mostrar las propiedades (normalmente en una hoja de propiedades).
- Modificar esas propiedades (en tiempo de diseño).

Por ejemplo, una herramienta de construcción cuando introspeccione nuestro Bean, descubrirá dos métodos:

```
public Color getColor() { ... }  
public void setColor(Color c) { ... }
```

A partir de esto la herramienta puede asegurar que existe una propiedad llamada `color`, que se puede leer y reescribir, y que su tipo es `Color`. Además, la herramienta puede intentar localizar un editor de propiedad para ese tipo, y mostrar la propiedad para que pueda ser editada.

Añadir la propiedad `Color` a `SimpleBean`

Haremos los siguientes cambios en `SimpleBean.java` para añadir la propiedad `color`:

1. Crear e inicializar una variable de ejemplar privada:

```
private Color color = Color.green;
```

2. Escribir un método para obtener la propiedad:

```
public Color getColor(){  
    return color;  
}
```

3. Escribir un método para seleccionar la propiedad:

```
public void setColor(Color newColor){  
    color = newColor;  
    repaint();  
}
```

4. Sobrecribir el método paint(). Esto es necesario para todas las subclases de Canvas.

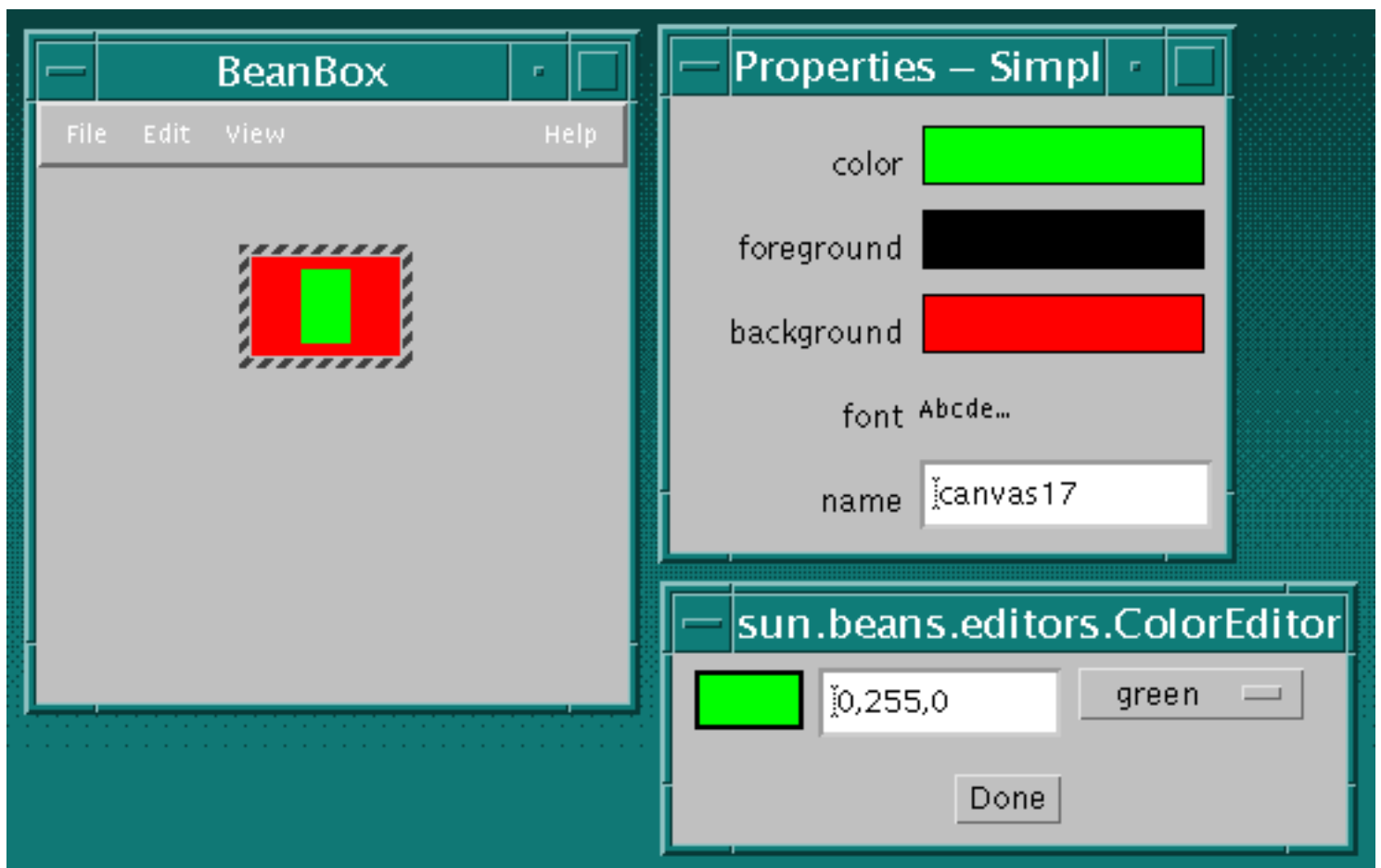
```
public void paint(Graphics g) {  
    g.setColor(color);  
    g.fillRect(20, 5, 20, 30);  
}
```

5. Compilar el Bean, cargarlo en el ToolBox, y crear un ejemplar en el BeanBox.

Los resultados son:

- SimpleBean se mostrará con un rectángulo verde centrado.
- La hoja de propiedades contendrá una nueva propiedad Color. El mecanismo de introspección también buscará un editor para la propiedad color. El editor de la propiedad Color es uno de los editores por defecto suministrados con el BeanBox. Este editor se asigna como editor de la propiedad Color de SimpleBean. Pulsa sobre la propiedad color en la hoja de propiedades para lanzar este editor.

Aquí tienes una ilustración del BeanBox que muestra el ejemplar revisado de SimpleBean dentro del BeanBox, la nueva propiedad color en la hoja de propiedades de SimpleBean y un editor de la propiedad color. Recuerda, pulsando sobre la propiedad color en la hoja de propiedades se lanzará este editor.



Aquí tenemos el código fuente completo de SimpleBean revisado para la propiedad color.

```
package sunw.demo.simple;

import java.awt.*;
import java.io.Serializable;

public class SimpleBean extends Canvas
    implements Serializable{

    private Color color = Color.green;

    //property getter method
    public Color getColor(){
        return color;
    }

    //property setter method. Sets new SimpleBean
    //color and repaints.
    public void setColor(Color newColor){
        color = newColor;
        repaint();
    }
}
```

```
}

public void paint(Graphics g) {
    g.setColor(color);
    g.fillRect(20, 5, 20, 30);
}

//Constructor sets inherited properties
public SimpleBean(){
    setSize(60,40);
    setBackground(Color.red);
}
}
```

Propiedades Compartidas

Algunas veces cuando cambia una propiedad de un Bean, otros objetos podrían querer ser notificados del cambio y tomar alguna acción basándose en ese cambio. Cuando una propiedad compartida cambia, la notificación del cambio se envía a los oyentes interesados.

Un Bean que contiene una propiedad compartidas debe mantener una lista de los oyentes de la propiedad, y alertar a dichos oyentes cuando cambie la propiedad. La clase `PropertyChangeSupport` implementa métodos para añadir y eliminar objetos `PropertyChangeListener` de una lista, y para lanzar objetos `PropertyChangeEvent` a dichos oyentes cuando cambia la propiedad compartida. Nuestros Beans pueden descender de esta clase, o utilizar una clase interna.

Un objeto que quiera escuchar los cambios de una propiedad debe poder añadirse o eliminarse de la lista de oyentes del Bean que contiene la propiedad, y responder al método de notificación del evento que señala que la propiedad ha cambiado. Implementando el interface `PropertyChangeListener` el oyente puede ser añadido a la lista mantenida por el Bean de la propiedad compartida, y como implementa el método `PropertyChangeListener.propertyChange()`, el oyente puede responder a las notificaciones de cambio de la propiedad.

La clase `PropertyChangeEvent` encapsula la información del cambio de la propiedad, y es enviada desde la fuente del evento de cambio de propiedad a cada objeto de la lista de oyentes mediante el método `propertyChange()`.

Las siguientes secciones proporcionan los detalles de la implementación de propiedades compartidas.

Implementar Propiedades Compartidas dentro de un Bean

Como ya habrás leído, un Bean que contenga propiedades compartidas deberá:

- Permitir a sus oyentes registrar o eliminar su interés en recibir eventos de cambio de propiedad.
- Disparar eventos de cambio de propiedad a los oyentes interesados.

La clase `PropertyChangeSupport` implementa dos métodos para añadir y eliminar objetos `PropertyChangeListener` de una lista de oyentes, e implementa un método que dispara eventos de cambios de propiedad a cada uno de los oyentes de la lista. Nuestro Bean puede descender de `PropertyChangeSupport`, o utilizarla como clase interna.

Para implementar una propiedad compartida, seguiremos estos pasos:

1. Importaremos el paquete `java.beans`, esto nos da acceso a la clase `PropertyChangeSupport`.
2. Ejemplarizar un objeto `PropertyChangeSupport`:

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

Este objeto mantiene una lista de oyentes del cambio de propiedad y lanza eventos de cambio de propiedad.

3. Implementar métodos para mantener la lista de oyentes. Como `PropertyChangeSupport` implementa esto métodos sólo tenemos que envolver las llamadas a los métodos del objeto soportado:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
```

```

        changes.addPropertyChangeListener(1);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }
}

```

4. Modificar un método de selección de la propiedad para que lance un evento de cambio de propiedad:

```

public void setLabel(String newLabel) {
    String oldLabel = label;
    label = newLabel;
    sizeToFit();
    changes.firePropertyChange("label", oldLabel, newLabel);
}

```

Observa que `setLabel()` almacena el valor antiguo de `label`, porque los dos valores, el nuevo y el antiguo deben ser pasados a `firePropertyChange()`.

```

public void firePropertyChange(String propertyName,
                               Object oldValue, Object newValue)

```

`firePropertyChange()` convierte sus parámetros en un objeto `PropertyChangeEvent`, y llama a `propertyChange(PropertyChangeEvent pce)` de cada oyente registrado. Observa que los valores nuevo y antiguo son tratados como valores `Object`, por eso si los valores de la propiedad son tipos primitivos como `int`, se debe utilizar la versión del objeto `java.lang.Integer`. Observa también que el evento de cambio de propiedad se dispara después de que la propiedad haya cambiado.

Cuando el `BeanBox` reconoce el patrón de diseño de una propiedad compartida dentro de un `Bean`, se verá un interface `propertyChange` cuando se despliega el menú `Edit|Events`.

Ahora que hemos dada a nuestro `Bean` la habilidad de lanzar eventos cuando cambia una propiedad, el siguiente paso es crear un oyente.

Implementar Oyentes de Propiedades Compartida

Para oír los eventos de cambio de propiedad, nuestro `Bean` oyente debe implementar el interface `PropertyChangeListener`. Este interface contiene un método:

```

public abstract void propertyChange(PropertyChangeEvent evt)

```

El `Bean` fuente llama a este método de notificación de todos los oyentes de su lista de oyentes.

Por eso para hacer que nuestra clase pueda oír y responder a los eventos de cambio de propiedad, debe:

1. Implementar el interface `PropertyChangeListener`.

```

public class MyClass implements java.beans.PropertyChangeListener,
                               java.io.Serializable {

```

2. Implementar el método `propertyChange()` en el oyente. Este método necesita contener el código que maneja lo que se necesita hacer cuando el oyente recibe el evento de cambio de propiedad. Por ejemplo, una llamada a un método de selección de la clase oyente: un cambio en una propiedad del `Bean` fuente se propaga a una propiedad del `Bean` oyente.

Para registrar el interés en recibir notificaciones sobre los cambios en una propiedad de un `Bean`, el `Bean` oyente llama al método de registro de oyentes del `Bean` fuente, por ejemplo:

```
button.addPropertyChangeListener(aButtonListener);
```

O se puede utilizar una clase adaptador para capturar el evento de cambio de propiedad, y subsecuentemente llamar al método correcto dentro del objeto oyente. Aquí tienes un ejemplo tomado de los ejemplos comentados del fichero
beans/demo/sunw/demo/misc/ChangeReporter.java.

```
OurButton button = new OurButton();
...
PropertyChangeAdapter adapter = new PropertyChangeAdapter();
...
button.addPropertyChangeListener(adapter);
...
class PropertyChangeAdapter implements PropertyChangeListener
{
    public void propertyChange(PropertyChangeEvent e)
    {
        reporter.reportChange(e);
    }
}
```

Propiedades Compartida en el BeanBox

El BeanBox maneja las propiedades compartida utilizando una clase adaptador. Los Beans OurButton y ChangeReporter pueden ser utilizados para ilustrar esta técnica. Para ver como funciona, seguiremos estos pasos:

1. Arrastrar ejemplares de OurButton y de ChangeReporter al BeanBox.
2. Seleccionar el ejemplar de OurButton y elegir el menú Edit|Events|propertyChange|propertyChange.
3. Conectar la línea que aparece al ejemplar de ChangeReporter. Se mostrará el cuadro de diálogo EventTargetDialog.
4. Elegir reportChange desde EventTargetDialog. Se generará y compilará la clase adaptador de eventos.
5. Seleccionar OurButton y cambiar alguna de sus propiedades. Veras informes de cambios en ChangeReporter.

Detrás de la escena, el BeanBox genera el adaptador de eventos. Este adaptador implementa el interface PropertyChangeListener, y también genera una implementación del método propertyChange() que llama el método ChangeReporter.reportChange(). Aquí tienes el código fuente del adaptador generado:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.misc.ChangeReporter;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

public class ____Hookup_14636f1560 implements
    java.beans.PropertyChangeListener, java.io.Serializable {

    public void setTarget(sunw.demo.misc.ChangeReporter t) {
```

```
        target = t;
    }

    public void propertyChange(java.beans.PropertyChangeEvent arg0) {
        target.reportChange(arg0);
    }

    private sunw.demo.misc.ChangeReporter target;
}
```

El Bean ChangeReporter no necesita implementar el interface PropertyChangeListener; en su lugar, la clase adaptador generada por el BeanBox implementa PropertyChangeListener, y el método propertyChange() del adaptador llama al método apropiado del objeto fuente (ChangeReporter).

El BeanBox pone las clases de los adaptadores de eventos en el directorio
beans/beanbox/tmp/sunw/beanbox.

Ozito

Propiedades Restringidas

Una propiedad de un Bean está Restringida cuando cualquier cambio en esa propiedad puede ser vetado, Normalmente es un objeto exterior el que ejerce su derecho a veto, pero el propio Bean puede vetar un cambio en una propiedad.

El API de JavaBeans proporciona un mecanismo de eventos similar al mecanismo de las propiedades compartidas, que permite a los objetos vetar los cambios de una propiedad de un Bean.

Existen tres partes en la implementación de una propiedad Restringida:

- Un Bean fuente que contiene una o más propiedades restringidas.
- Objetos oyentes que implementan el interface `VetoableChangeListener`. Estos objetos aceptan o rechazan la proposición de un cambio en la propiedad restringida del Bean fuente.
- Un objeto `PropertyChangeEvent` que contiene el nombre de la propiedad, y sus valores nuevo y antiguo. Esta es la misma clase utilizada por las propiedades compartidas.

Implementar Propiedades Restringidas dentro de un Bean.

Un Bean que contenga propiedades restringidas debe:

- Permitir que objetos `VetoableChangeListener` registren su interés en recibir notificaciones de proposiciones de cambio de una propiedad.
- Disparar eventos de cambio de propiedad hacia aquellos oyentes interesados, cuando se proponga un cambio de propiedad. El evento debería ser disparado antes de que el cambio real de la propiedad tenga lugar. El `PropertyChangeEvent` es disparado por una llamada al método `vetoableChange()` de todos los oyentes.
- Si un oyente veta el cambio, debe asegurarse que todos los demás oyentes pueden volver al valor antiguo. Esto significa volver a llamar al método `vetoableChange()` de todos los oyentes, con un `PropertyChangeEvent` que contenga el valor antiguo.

La clase `VetoableChangeSupport` se proporciona para implementar estas capacidades. Esta clase implementa métodos para añadir y eliminar objetos `VetoableChangeListener` a una lista de oyentes, y un método que dispara eventos de cambio de propiedad a todos los oyentes de la lista cuando se propone un cambio de propiedad. Este método también capturará cualquier veto, y re-enviará el evento de cambio de propiedad con el valor original de la propiedad. Nuestro Bean puede descender de la clase `VetoableChangeSupport`, o utilizar un ejemplar de ella.

Observa que, en general, las propiedades restringidas también deberían ser propiedades compartidas. Cuando ocurre un cambio en una propiedad restringida, puede ser enviado un `PropertyChangeEvent` mediante `PropertyChangeListener.propertyChange()` para indicar a todos los Beans `VetoableChangeListener` que el cambio a tenido efecto.

El Bean `JellyBean` tiene una propiedad restringida. Veremos su código para ilustrar los pasos e implementar propiedades restringidas:

1. Importar el paquete `java.beans`, esto nos da acceso a la clase `VetoableChangeSupport`.
2. Ejemplarizar un objeto `VetoableChangeSupport` dentro de nuestro Bean:

```
private VetoableChangeSupport vetos =
    new VetoableChangeSupport(this);
```

VetoableChangeSupport maneja una lista de objetos VetoableChangeListener, y dispara eventos de cambio de propiedad a cada objeto de la lista cuando ocurre un cambio en una propiedad restringida.

3. Implementar métodos para mantener la lista de oyentes de cambio de propiedad. Esto sólo envuelve la llamada a los métodos del objeto VetoableChangeSupport:

```
public void addVetoableChangeListener(VetoableChangeListener l) {
    vetos.addVetoableChangeListener(l);
}
public void removeVetoableChangeListener(VetoableChangeListener l) {
    vetos.removeVetoableChangeListener(l);
}
```

4. Escribir un método seleccionador de propiedades que dispare un evento de cambio de propiedad cuando la propiedad ha cambiado. Esto incluye añadir una clausula throws a la firma del método. El método setPriceInCents() de JellyBean se parece a esto:

```
public void setPriceInCents(int newPriceInCents)
    throws PropertyVetoException {
    int oldPriceInCents = ourPriceInCents;

    // First tell the vetoers about the change. If anyone objects, we
    // don't catch the exception but just let it pass on to our caller.
    vetos.fireVetoableChange("priceInCents",
        new Integer(oldPriceInCents),
        new Integer(newPriceInCents));

    // No-one vetoed, so go ahead and make the change.
    ourPriceInCents = newPriceInCents;
    changes.firePropertyChange("priceInCents",
        new Integer(oldPriceInCents),
        new Integer(newPriceInCents));
}
```

Observa que setPriceInCents() almacena el valor antiguo de price, porque los dos valores, el nuevo y el antiguo, deben ser pasados a fireVetoableChange(). También observa que los precios primitivos int se han convertido a objetos Integer.

```
public void fireVetoableChange(String propertyName,
    Object oldValue,
    Object newValue)
    throws PropertyVetoException
```

Estos valores se han empaquetado en un objeto PropertyChangeEvent enviado a cada oyente. Los valores nuevo y antiguo son tratados como valores Object, por eso si son tipos primitivos como int, deben utilizarse sus versiones objetos como java.lang.Integer.

Ahora necesitamos implementar un Bean que escuche los cambios en las propiedades restringidas.

Implementar Oyentes de Propiedades Restringidas

Para escuchar los eventos de cambio de propiedad, nuestro Bean oyente debe implementar el interface `VetoableChangeListener`. El interface contiene un método:

```
void vetoableChange(PropertyChangeEvent evt)
    throws PropertyVetoException;
```

Por eso para hacer que nuestra clase pueda escuchar y responder a los eventos de cambio de propiedad debe:

1. Implementar el interface `VetoableChangeListener`.
2. Implementar el método `vetoableChange()`. Este es el método al que llamará el Bean fuente en cada objeto de la lista de oyentes (mantenida por el objeto `VetoableChangeSupport`). Este también es el método que ejerce el poder del veto. Un cambio de propiedad es vetado lanzando una `PropertyVetoException`.

Observa que el objeto `VetoableChangeListener` frecuentemente es una clase adaptador. La clase adaptador implementa el interface `VetoableChangeListener` y el método `vetoableChange()`. Este adaptador es añadido a la lista de oyentes del Bean restringido, intercepta la llamada a `vetoableChange()`, y llama al método del Bean fuente que ejerce el poder del veto.

Propiedades Restringidas en el BeanBox

Cuando el BeanBox reconoce el patrón de diseño de una propiedad restringida dentro de un Bean, se verá un ítem de un interface `vetoableChange` al desplegar el menú `Edit|Events`.

El BeanBox genera una clase adaptador cuando se conecta un Bean que tiene una propiedad restringida con otro Bean. Para ver como funciona esto, sigue estos pasos:

1. Arrastra ejemplares de `Voter` y de `JellyBean` al BeanBox.
2. Selecciona el ejemplar de `JellyBean` y elige el menú `Edit|Events|vetoableChange|vetoableChange`.
3. Conecta la línea que aparece con el Bean `Voter`. Esto mostrará el panel `EventTargetDialog`.
4. Elige el método `vetoableChange` del Bean `Voter`, y pulsa sobre el botón `OK`. Esto genera un adaptador de eventos que puedes ver en el directorio `beans/beanbox/tmp/sunw/beanbox`.
5. Prueba la propiedad restringida. Selecciona el `JellyBean` y edita sus propiedades `priceInCents` en la hoja de propiedades. Se lanzará una `PropertyVetoException`, y se mostrará un dialogo de error.

Detrás de la escena el BeanBox genera el adaptador de evento. Este adaptador implementa el interface, `VetoableChangeListener`, y también genera un método `vetoableChange()` que llama al método `Voter.vetoableChange()`. Aquí tienes el código generado para el adaptador:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.misc.Voter;
import java.beans.VetoableChangeListener;
```



```
import java.beans.PropertyChangeEvent;

public class ____Hookup_1475dd3cb5 implements
    java.beans.VetoableChangeListener, java.io.Serializable {

    public void setTarget(sunw.demo.misc.Voter t) {
        target = t;
    }

    public void vetoableChange(java.beans.PropertyChangeEvent arg0)
        throws java.beans.PropertyVetoException {
        target.vetoableChange(arg0);
    }

    private sunw.demo.misc.Voter target;
}
```

El Bean Voter no necesita implementar el interface VetoableChangeListener; en su lugar, la clase adaptador generada implementa VetoableChangeListener. El método vetoableChange() del adaptador llama al método apropiado en el objeto fuente (Voter).

Para Propiedades Restringidas

Al igual que las propiedades compartidas, existe un patrón de diseño soportado para añadir y eliminar objetos VetoableChangeListener que se han unido a un nombre de propiedad específico:

```
void addVetoableChangeListener(String propertyName,
    VetoableChangeListener listener);
void removeVetoableChangeListener(String propertyName,
    VetoableChangeListener listener);
```

Como alternativa, por cada propiedad restringida de un Bean se pueden proporcionar métodos con la siguiente firma para registrar y eliminar oyentes de una propiedad básica:

```
void add<PropertyName>Listener(VetoableChangeListener p);
void remove<PropertyName>Listener(VetoableChangeListener p);
```


Propiedades Indexadas

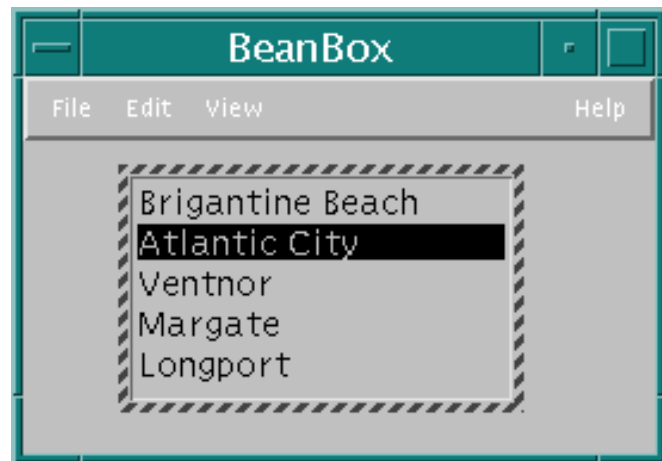
Las propiedades indexadas representan colecciones de valores a los que se accede por índices como en los arrays. Los patrones de diseño de las propiedades indexadas son

```
//Métodos para acceder al array de propiedades completo
public <PropertyType>[] get();
public void set<PropertyName>([] value);
```

```
//Métodos para acceder a valores individuales
public <PropertyType> get(int index);
public void set<PropertyName>(int index, value);
```

De acuerdo con estos patrones las herramientas de programación saben que nuestro Bean contiene una propiedad indexada.

El Bean OurListBox ilustra como utilizar las propiedades indexadas. OurListBox descende de la clase [List](#) para proporcionar un Bean que presenta al usuario una lista de elecciones: Choices que se puede proporcionar y diseñar durante el diseño. Aquí tienes una ilustración de un ejemplar de OurListBox :



OurListBox expone el item indexado con los siguientes métodos accesoros:

```
public void setItems(String[] indexprop) {
    String[] oldValue=fieldIndexprop;
    fieldIndexprop=indexprop;
    populateListBox();
    support.firePropertyChange("items",oldValue, indexprop);
}

public void setItems(int index, String indexprop) {
    String[] oldValue=fieldIndexprop;
    fieldIndexprop[index]=indexprop;
    populateListBox();
    support.firePropertyChange("Items",oldValue, fieldIndexprop);
}

public String[] getItems() {
    return fieldIndexprop;
}
```

```

}

public String getItem(int index) {
    return items[index];
}

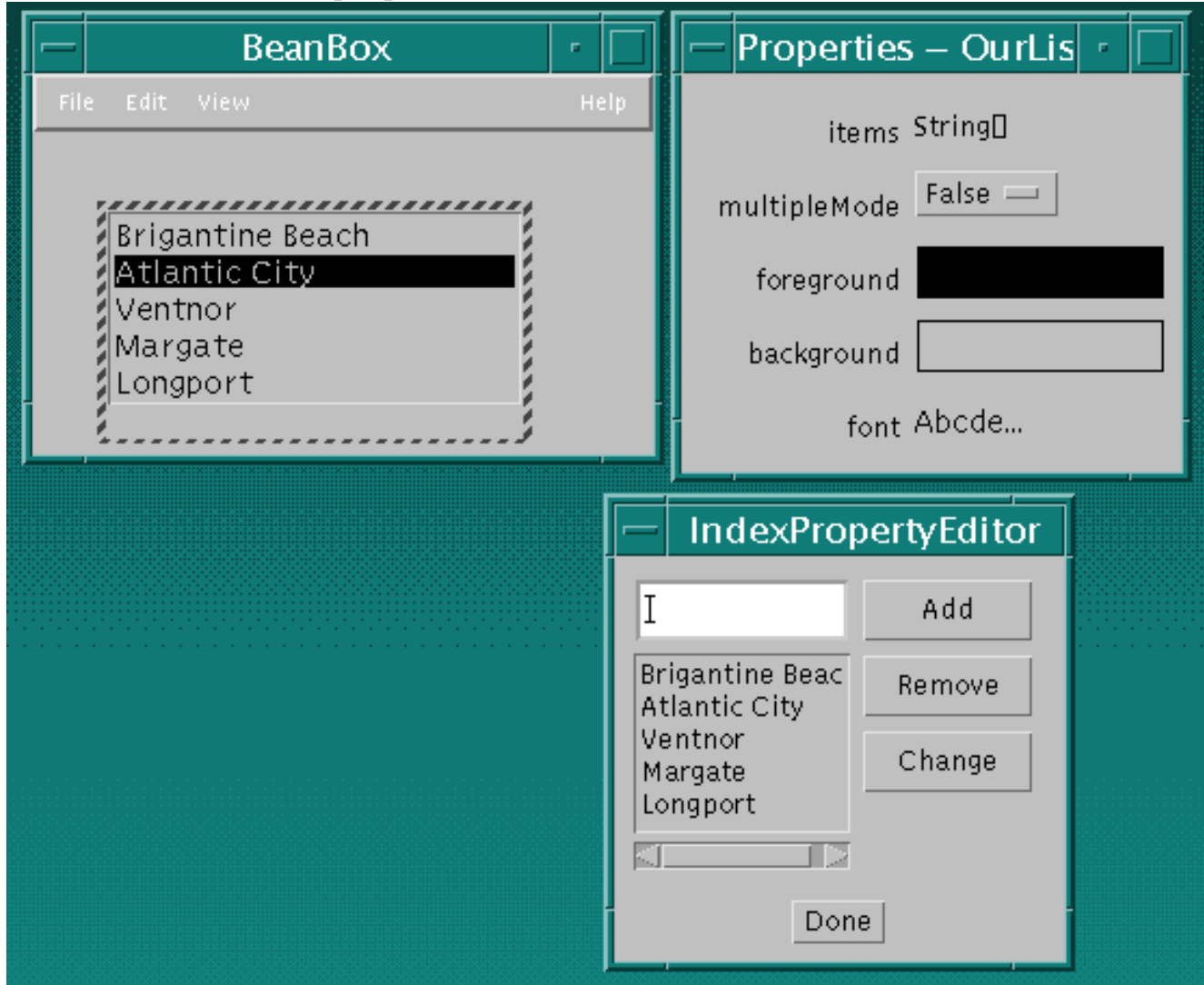
```

Cuando un ítem es seleccionado por uno de los métodos `setItem()`, `OurListBox` se puebla con el contenido de un array `String`.

La exposición de las propiedades indexadas es casi tan fácil como la de las propiedades sencillas. Sin embargo, escribir un editor de propiedades indexadas requiere escribir un editor de propiedades personalizado.

Editores de Propiedades Indexadas

El Bean `OurListBox` proporciona un `IndexPropertyDescriptor` asociado que es un buen ejemplo de cómo implementar un editor de propiedades indexadas. La siguiente ilustración muestra un ejemplar de `OurListBox` en el `BeanBox`, la hoja de propiedades que contiene una entrada para los ítems de la propiedad indexada, y el `IndexPropertyDescriptor` que sale cuando se pulsa sobre los ítems de la propiedad:



Implementar `IndexPropertyDescriptor` es lo mismo que implementar cualquier editor de propiedades personalizado:

1. Implementar el interface `PropertyDescriptor`:

```
public class IndexPropertyEditor extends Panel
    implements PropertyEditor, ActionListener {
```

Se puede utilizar la clase `PropertyEditorSupport`, utilizando una subclase o como clase interna.

2. Denotar el editor personalizado en una clase `BeanInfo` relacionada. `OurListBox` tiene una clase `OurListBoxBeanInfo` relacionada que contiene el siguiente código:

```
itemsprop.setPropertyEditorClass(IndexPropertyEditor.class);
```

3. Hacer que el editor de propiedades sea una fuente de eventos compartidos. El editor de propiedades registrará los oyentes de la propiedad y disparará los eventos de cambio de propiedad a esos oyentes. Así es como los cambios en la propiedad se propagan hacia el Bean (mediante la hoja de propiedades). Por eso `IndexPropertyEditor` ejemplariza una clase interna `PropertyChangeSupport`:

```
private PropertyChangeSupport support =
    new PropertyChangeSupport(this);
```

Proporciona la habilidad de que los objetos registren su interés en ser notificados cuando una propiedad es editada:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
    support.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l) {
    support.removePropertyChangeListener(l);
}
```

Y dispara un evento de cambio de propiedad a dichos oyentes:

```
public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == addButton) {
        listBox.addItem(textBox.getText());
        textBox.setText("");
        support.firePropertyChange("", null, null);
    }
    else if (evt.getSource() == textBox) {
        listBox.addItem(textBox.getText());
        textBox.setText("");
        support.firePropertyChange("", null, null);
    }
    ...
}
```

`IndexPropertyEditor` mantiene `listbox` como una copia de `OurListBox`. Cuando se hace un cambio en `listbox`, se dispara un evento de cambio de propiedad a todos los oyentes.

Cuando una hoja de propiedades, que está registrada como un oyente de IndexPropertyEditor, recibe un evento de cambio de propiedad desde IndexPropertyEditor, llama a IndexPropertyEditor.getValue() para recuperar los ítems nuevos o cambiados para actualizar el Bean.

Manipular Eventos en el BeanBox

Los Beans utilizan el mecanismo de eventos implementado en el JDK 1.1, por eso implementar eventos en los Beans es igual que implementar eventos en cualquier componente del JDK 1.1. Esta sección describe como se utiliza este mecanismo de eventos en los Beans y en el BeanBox.

Cómo descubre el BeanBox las capacidades de Eventos de un Beans

El BeanBox utiliza la introspección de patrones de diseño o una clase [BeanInfo](#) para descubrir los eventos que puede disparar un Bean.

Utilizar la Introspección para Descubrir los Eventos Lanzados por un Bean

Los JavaBeans proporcionan patrones de diseño orientados a eventos para dar a las herramientas de introspección la posibilidad de descubrir los eventos que puede lanzar un Bean.

Para que un Bean sea la fuente de un evento, debe implementar métodos que añadan y eliminen oyentes para el tipo del objeto. Los patrones de diseño para esos métodos son

```
public void add<EventListenerType>(<EventListenerType> a)
public void remove<EventListenerType>(<EventListenerType> a)
```

Estos métodos le permiten a un Bean fuente saber donde lanzar los eventos. Entonces el Bean fuente lanza los eventos a aquellos Beans oyentes que utilicen los métodos para aquellos interfaces particulares. Por ejemplo, si un Bean fuente registra objetos ActionListener, lanzará eventos a dichos objetos llamando al método actionPerformed() de dichos oyentes.

Para ver los eventos descubiertos utilizando patrones de diseño, arrastra un ejemplar de OurButton dentro del BeanBox y despliega el menú Edit|Events. Esto muestra una lista de interface de eventos que OurButton puede disparar. Observa que el propio OurButton sólo añade dos de esos interfaces; el resto son heredados de la clase base.

Utilizar BeanInfo para Definir los Eventos Lanzados por el Bean

Se pueden "publicar" explícitamente los eventos lanzados por un Bean, utilizando una clase que implementa el interface BeanInfo. El Bean ExplicitButton subclasifica OurButton, y proporciona una clase ExplicitButtonBeanInfo asociada. ExplicitButtonBeanInfo implementa el siguiente método para definir explícitamente los interfaces que ExplicitButton utiliza para lanzar eventos.

```

public EventSetDescriptor[] getEventSetDescriptors() {
    try {
        EventSetDescriptor push = new EventSetDescriptor(beanClass,
            "actionPerformed",
            java.awt.event.ActionListener.class,
            "actionPerformed");

        EventSetDescriptor changed = new EventSetDescriptor(beanClass,
            "propertyChange",
            java.beans.PropertyChangeListener.class,
            "propertyChange");

        push.setDisplayName("button push");
        changed.setDisplayName("bound property change");

        EventSetDescriptor[] rv = { push, changed };
        return rv;
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }
}

```

Arrastra un ejemplar de `ExplicitButton` al `BeanBox`, y despliega el menú `Edit|Events`. Observa que sólo se listan aquellos interfaces expuestos explícitamente en la clase `ExplicitButtonBeanInfo`. No se exponen las capacidades heredadas. Puedes ver la página [El Interface BeanInfo](#) para más información sobre este interface.

Ver los Eventos de un Bean en el BeanBox

Si has seleccionado un Bean `OurButton` en el `BeanBox`, despliega el menú `Edit|Events`, verás una lista de interfaces a los que `OurButton` puede lanzar eventos. Cuando se selecciona algún interface, se muestran los métodos que lanzan diferentes eventos a dichos interfaces. Estos corresponden a todos los eventos que `OurButton` puede lanzar

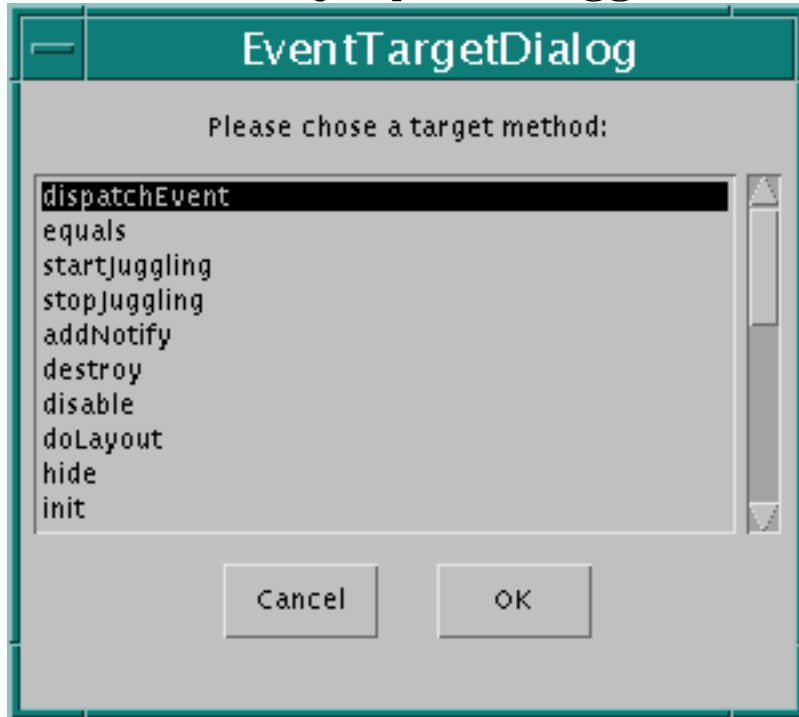
Capturando Eventos en el BeanBox

En este ejemplo utilizaremos dos ejemplares del Bean `OurButton` para arrancar y parar un ejemplar del Bean `Juggler` animado. Etiquetaremos estos botones como "start" y "stop"; haremos que cuando se pulse el botón "start" se llame al método `startJuggling` del Bean `Juggler`; y que cuando se pulse el botón "stop", se llame al método `stopJuggling` del Bean `Juggler`.

1. Arrancar el `BeanBox`.
2. Arrastra un Bean `Juggler` y dos ejemplares de `OurButton` en el `BeanBox`.
3. Selecciona un ejemplar de `OurButton`. En la hoja de propiedades, cambia la

etiqueta a "start".

4. Selecciona el segundo ejemplar de OurButton y cambia su etiqueta a "stop".
5. Selecciona el botón start. elige el menú Edit|Events|action|actionPerformed. Esto hace que aparezca una línea marcada entre el botón start y el cursor. Pulsar sobre el ejemplar de Juggler. Esto trae el EventTargetDialog:



Esta lista contiene los métodos de Juggler que no toman argumentos, o argumentos del tipo actionPerformed.

6. Selecciona el método startJuggling y pulsa OK. Verás un mensaje indicando que el BeanBox está generando una clase adaptador.
7. repite los dos últimos pasos sobre el botón stop, excepto en elegir el método stopJuggling en el EventTargetDialog.

Si pulsas ahora sobre los botones start y stop se arrancará y parará la ejecución de Juggler. Aquí tienes una descripción general de lo que ha sucedido:

- Los botones start y stop son fuentes de eventos. Las fuentes de eventos lanzan eventos a los destinos de eventos. En este ejemplo el Bean Juggler es el Destino del Evento.
- Cuando se selecciona el botón start y se elige un método de evento (mediante el menú Edit|Event), se está eligiendo el tipo de evento que lanzará la fuente del evento.
- cuando se conecta la línea con otro Bean, estás seleccionando el Bean Destino del evento.
- El EventTargetDialog lista los métodos que pueden aceptar ese tipo de eventos o que no tomen parámetros. Cuando se elige un método en el EventTargetDialog, se está eligiendo el método que recibirá el evento lanzado, y actuará sobre él.

Utiliza el menú File|Save para grabar este ejemplo en un fichero de tu elección.

Clases Adpatadoras de Eventos

El BeanBox genera una clase adaptador que se interpone entre la fuente y el destino del evento. La clase adaptador implementa el interface del oyente de evento apropiado (como el oyente real), captura el evento lanzado por el botón, y llama al método seleccionado del bean destino. Aquí tienes la clase adaptador generada por el BeanBox que se interpone entre el botón start y el JugglerBean:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.juggler.Juggler;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ____Hookup_1474c0159e implements
    java.awt.event.ActionListener, java.io.Serializable {

    public void setTarget(sunw.demo.juggler.Juggler t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.startJuggling(arg0);
    }

    private sunw.demo.juggler.Juggler target;
}
```

El adaptador implementa el interface ActionListener que se ha seleccionado en el menú Edit|Events del BeanBox. ActionListener declare un método, actionPerformed(), que es implementado por el adaptador para llamar al método del Bean destino (startJuggling()) que se ha seleccionado. El método setTarget() del adaptador es llamado por el BeanBox para seleccionar el Bean destino real, en este caso Juggler.

El Bean EventMonitor

El Bean EventMonitor (beans/demo/sunw/demo/encapsulatedEvents) imprime informes de los eventos del Bean fuente, en un listbox. Para ver como funciona esto, sigue estos pasos:

1. Arrastra ejemplares de OurButton y EventMonitor hasta el BeanBox. Podrías redimensionar el EventMonitor (y el BeanBox) para acomodar la visión de los informes de eventos.

2. Selecciona el ejemplar de OurButton, y elige cualquier método en el menú Edit|Events.
3. Conecta la línea que aparece con el EventMonitor, y elige su initiateEventSourceMonitoring en el EventTargetDialog.
4. Selecciona el Bean OurButton. Estarás viendo un informe de eventos en el EventMonitor

Cuando el primer evento es enviado, EventMonitor analiza el Bean fuente para descubrir todos los eventos que lanza, crea y registra oyentes de eventos para cada tipo de evento, y luego informa siempre que se lance un evento. Esto es útil para depuración, Intenta conectar otros Beans a EventMonitor para observar sus eventos.

Eventos para Propiedades Compartidas y Restringidas

Las páginas sobre propiedades [Compartidas](#) y [Restringidas](#) describen dos interfaces de oyentes de eventos específicos.

El Interface BeanInfo

¿Cómo examinan las herramientas de desarrollo a un Bean para exponer sus características (propiedades, eventos y métodos) en un hoja de propiedades? Utilizando la clase `java.beans.Introspector`. Esta clase utiliza el corazón de reflexión del API del JDK para descubrir los métodos del Bean, y luego aplica los patrones de diseño de los JavaBeans para descubrir sus características. Este proceso de descubrimiento se llama introspección.

De forma alternativa, se pueden exponer explícitamente las características del Bean en una clase asociada separada que implemente el interface `BeanInfo`. Asociando una clase `BeanInfo` con un Bean se puede:

- Exponer solo aquellas características que queremos exponer.
- Relegar en `BeanInfo` la exposición de algunas características del Bean, mientras se deja el resto para la reflexión de bajo nivel.
- Asociar un icono con el Bean fuente.
- Especificar una clase personalizada.
- Segregar las características entre normales y expertas.
- Proporcionar un nombre más descriptivo, información adicional sobre la característica del Bean.

`BeanInfo` define métodos que devuelven descriptores para cada propiedad, método o evento que se quiere exponer. Aquí tienes los prototipos de estos métodos:

```
PropertyDescriptor[] getPropertyDescriptors();
MethodDescriptor[]  getMethodDescriptors();
EventSetDescriptor[] getEventSetDescriptors();
```

Cada uno de estos métodos devuelve un array de descriptores para cada característica.

Descriptores de Características

Las clases `BeanInfo` contienen descriptores que precisamente describen las características del Bean fuente. El JDK implementa las siguientes clases:

- `FeatureDescriptor` es la clase base para las otras clases de descriptores. Declara los aspectos comunes a todos los tipos de descriptores.
- `BeanDescriptor` describe el tipo de la clase y el nombre del Bean fuente y describe la clase personalizada del Bean fuente si existe.
- `PropertyDescriptor` describe las propiedades del Bean fuente.
- `IndexedPropertyDescriptor` es una subclase de `PropertyDescriptor`, y describe las propiedades indexadas del Bean fuente.
- `EventSetDescriptor` describe los eventos lanzados por el Bean fuente.
- `MethodDescriptor` describe los métodos del Bean fuente.
- `ParameterDescriptor` describe los parámetros de métodos.

El interface `BeanInfo` declara métodos que devuelven arrays de los descriptores anteriores.

Crear una Clase BeanInfo

Utilizaremos la clase `ExplicitButtonBeanInfo` para ilustrar la creación de una clase `BeanInfo`. Aquí están los pasos generales para crear una clase `BeanInfo`:

1. Nombrar la clase **BeanInfo**. Se debe añadir el estring "BeanInfo" al nombre de la clase fuente. Si el nombre de la clase fuente es `ExplicitButton`, la clase `BeanInfo` asociada se debe llamar `ExplicitButtonBeanInfo`
2. Subclasificar **SimpleBeanInfo**. Esta es una clase de conveniencia que implementa los métodos de `BeanInfo` para que devuelvan null o un valor nulo equivalente.

```
public class ExplicitButtonBeanInfo extends SimpleBeanInfo {
```

Utilizando `SimpleBeanInfo` nos ahorramos tener que implementar todos los métodos de `BeanInfo`; solo tenemos que sobrescribir aquellos métodos que necesitamos.

3. Sobrecribir los métodos apropiados para devolver las propiedades, los métodos o los eventos que queremos exponer. `ExplicitButtonBeanInfo` sobrescribe el método `getPropertyDescriptors()` para devolver cuatro propiedades:

```
public PropertyDescriptor[] getPropertyDescriptors() {
    try {
        PropertyDescriptor background =
            new PropertyDescriptor("background", beanClass);
        PropertyDescriptor foreground =
            new PropertyDescriptor("foreground", beanClass);
        PropertyDescriptor font =
            new PropertyDescriptor("font", beanClass);
        PropertyDescriptor label =
            new PropertyDescriptor("label", beanClass);

        background.setBound(true);
        foreground.setBound(true);
        font.setBound(true);
        label.setBound(true);

        PropertyDescriptor rv[] =
            {background, foreground, font, label};
        return rv;
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }
}
```

Existen dos cosas importantes que observar aquí:

- Si se deja fuera algún descriptor, la propiedad, evento o método no descrito no se expondrá. En otras palabras, se puede exponer selectivamente las propiedades, eventos o métodos, dejando fuera las que no queramos exponer.
 - Si un método obtenedor de características (por ejemplo `getMethodDescriptor()`) devuelve Null, se utilizará la reflexión de bajo nivel para esa característica. Esto significa, por ejemplo, que se pueden especificar explícitamente propiedades, y dejar que la reflexión de bajo nivel descubra los métodos. Si no se sobrescribe el método por defecto de `SimpleBeanInfo` que devuelve null, la reflexión de bajo nivel se utilizará para esta característica.
4. Opcionalmente, asociar un icono con el Bean fuente.

```

public java.awt.Image getIcon(int iconKind) {
    if (iconKind == BeanInfo.ICON_MONO_16x16 ||
        iconKind == BeanInfo.ICON_COLOR_16x16 ) {
        java.awt.Image img = loadImage("ExplicitButtonIcon16.gif");
        return img;
    }
    if (iconKind == BeanInfo.ICON_MONO_32x32 ||
        iconKind == BeanInfo.ICON_COLOR_32x32 ) {
        java.awt.Image img = loadImage("ExplicitButtonIcon32.gif");
        return img;
    }
    return null;
}

```

El BeanBox muestra el icono junto al nombre del Bean en el ToolBox. Se puede esperar que las herramientas de desarrollo hagan algo similar.

5. Especificar la **clase** del Bean fuente, y , si el Bean está personalizado, especificarlo también.

```

public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(beanClass, customizerClass);
}
...
private final static Class beanClass = ExplicitButton.class;
private final static Class customizerClass = OurButtonCustomizer.class;

```

Guarda la clase BeanInfo en el mismo directorio que la clase fuente. El BeanBox busca primero la clase BeanInfo de un Bean en el path del paquete del Bean. Si no se encuentra el BeanInfo, entonces la información del Bean busca en el path (mantenido por el Introspector). La información del Bean se busca por defecto en el path sun.beans.infos. Si no se encuentra la clase BeanInfo, se utiliza la reflexión de bajo nivel para descubrir las características del Bean.

Utilizar BeanInfo para Controlar las Características a Exponer

Si relegamos en la reflexión del bajo nivel para descubrir las características del Bean, todas aquellas propiedades, métodos y eventos que conformen el patrón de diseño apropiado serán expuestas en una herramienta de desarrollo. Esto incluye cualquier característica de la clase base. Si el BeanBox encuentra una clase BeanInfo asociada, entonces la información es utilizada en su lugar, y no se examinan más clases base utilizando la reflexión. En otras palabras, la información del BeanInfo sobrescribe la información de la reflexión de bajo nivel, y evita el examen de la clase base.

Mediante la utilización de una clase BeanInfo, se pueden exponer subconjuntos de una característica particular del Bean. Por ejemplo, mediante la no devolución de un método descriptor para un método particular, ese método no será expuesto en una herramienta de desarrollo.

Cuando se utiliza la clase BeanInfo

- Las características de la clase base no serán expuestas. Se pueden recuperar las características de la clase base utilizando el método BeanInfo.getAdditionalBeanInfo().

- Las propiedades, eventos o métodos que no tengan descriptor no serán expuestos. Para una característica particular, sólo aquellos ítems devueltos en el array de descriptors serán expuestos. Por ejemplo, si devolvemos descriptors para todos los métodos de un Bean excepto `foo()`, entonces `foo()` no será expuesto.
- La reflexión de bajo nivel será utilizada para las características cuyos metodos obtenedores devuelvan null. Por ejemplo su nuestra clase `BeanInfo` contiene esta implementación de métodos:

```
public MethodDescriptor[] getMethodDescriptors() {  
    return null;  
}
```

Entonces la reflexión de bajo nivel se utilizará para descubrir los métodos públicos del Bean.

Localizar las clases `BeanInfo`

Antes de examinar un Bean, el Introspector intentará encontrar una clase `BeanInfo` asociada con el bean. Por defecto, el Introspector toma el nombre del paquete del Bean totalmente cualificado, y le añade "`BeanInfo`" para formar un nuevo nombre de clase. Por ejemplo, si el Bean fuente es `sunw.demo.buttons.ExplicitButton`, el Introspector intentará localizar `sunw.demo.buttons.ExplicitButtonBeanInfo`.

Si esto falla, se buscará en todos los paquetes en el path de `BeanInfo`. El path de búsqueda de `BeanInfo` es mantenido por `Introspector.setBeanInfoSearchPath()` y `Introspector.getBeanInfoSearchPath()`.

Personalización de Beans

La apariencia y el comportamiento de un Bean pueden ser personalizados en el momento del diseño dentro de las herramientas de desarrollo compatibles con los Beans. Aquí podemos ver las dos formas más típicas de personalizar un Bean:

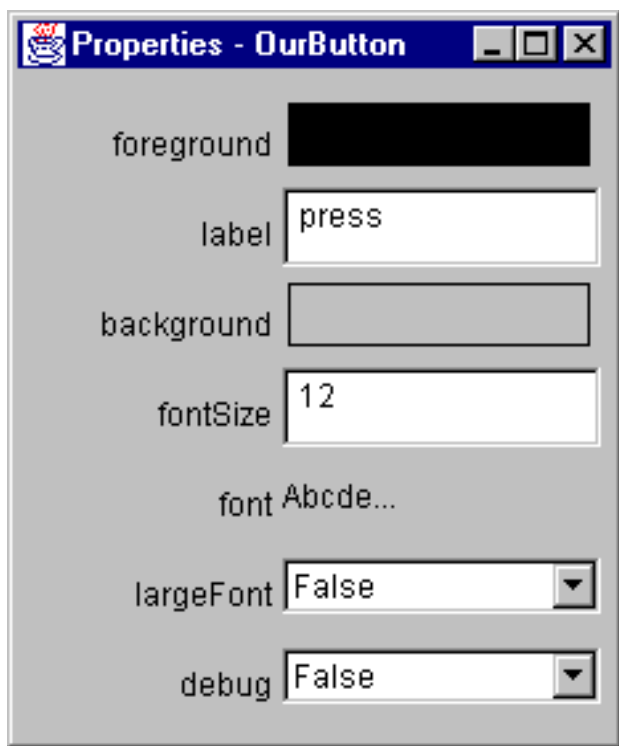
- Utilizando un editor de propiedades. Cada propiedad del Bean tiene su propio editor. Normalmente las herramientas de desarrollo muestran los editores de propiedades de un Bean en una hoja de propiedades. Un editor de propiedades está asociado y edita un tipo de propiedad particular.
- Utilizando personalizadores. Los personalizadores ofrecen un completo control GUI sobre la personalización del Bean. Estos personalizadores se utilizan allí donde los editores de propiedades no son prácticos o no se pueden aplicar. Al contrario que un editor de propiedad, que está asociado con una propiedad, un personalizador está asociado con un Bean.

Editores de Propiedades

Un editor de propiedad es una herramienta para personalizar un tipo de propiedad particular. Los editores de propiedades se muestran, o se activan desde la hoja de propiedades. Una hoja de propiedades determinará el tipo de la propiedad, buscará un editor de propiedad adecuado, y mostrará el valor actual de la propiedad de una forma adecuada.

Los editores de propiedades deben implementar el interface `PropertyEditor`. Este interface proporciona métodos que especifican cuando una propiedad debe ser mostrada en una hoja de propiedades.

Aquí tienes la hoja de propiedades del `BeanBox` contiendo las propiedades de `OurButton`:



Se empieza el proceso de edición de estas propiedades pulsando sobre la entrada de la propiedad en la hoja.

- Las propiedades label y fontSize se muestran en un campo de texto editable. Los cambios se realizan allí mismo.
- Las propiedades largeFont y debug son cajas de selección con elecciones discretas.
- Pulsando sobre las entradas foreground, background, y font se desplegarán paneles separados.

Cada uno de ellos se muestra dependiendo de los métodos de PropertyEditor que hayamos implementado para que devuelvan valores distintos de null.

Por ejemplo, el editor de propiedad int implementa el método `setAsText()`. Esto indica a la hoja de propiedades que la propiedad puede ser mostrada como un String, y lo hace en un campo de texto editable.

Los editores de propiedades Color y Font utilizan un panel separado, y sólo utilizan la hoja de propiedades para mostrar el valor actual de la propiedad. Este editor se muestra pulsando sobre ese valor.

Para mostrar el valor actual de la propiedad dentro de la hoja de propiedades, necesitamos sobreescibir `isPaintable()` para que devuelva true, y `paintValue()` para que dibuje el nuevo valor de la propiedad en un rectángulo en la hoja de propiedades. Aquí tienes como `ColorEditor` implementa `paintValue()`:

```
public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box) {  
    Color oldColor = gfx.getColor();  
    gfx.setColor(Color.black);
```

```

gfx.drawRect(box.x, box.y, box.width-3, box.height-3);
gfx.setColor(color);
gfx.fillRect(box.x+1, box.y+1, box.width-4, box.height-4);
gfx.setColor(oldColor);
}

```

Para soportar el editor de propiedades personalizado, necesitamos sobrescribir dos métodos más: `supportsCustomEditor()` para que devuelva `true`, y `getCustomEditor()` para que devuelve un ejemplar del editor personalizado. `ColorEditor.getCustomEditor()` devuelve `this`.

Además, la clase `PropertyEditorSupport` mantiene una lista de `PropertyChangeListener` y lanza notificaciones de eventos de cambio de propiedad a dichos oyentes cuando una propiedad compartida ha cambiado.

Como se Asocian los Editores de Propiedades con las Propiedades

Los editores de propiedades son descubiertos y asociados con una propiedad dada mediante:

- La asociación explícita mediante un objeto `BeanInfo`. El Bean Molecule utiliza esta técnica. Dentro de la clase `MoleculeBeanInfo`, el editor de propiedades del Bean Molecule se selecciona con la siguiente línea de código:

```
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

- Registro explícito mediante `java.Beans.PropertyEditorManager.registerEditor()`. Este método toma un par de argumentos: el tipo de la clase, y el editor asociado con ese tipo.
- Búsqueda por nombre. Si una clase no tiene asociado explícitamente un editor de propiedades, entonces el `PropertyEditorManager` busca la clase del editor de propiedad mediante:
 - Añadiendo "Editor" al nombre totalmente cualificado de la clase. Por ejemplo, para la clase `java.beans.ComplexNumber`, el manejador del editor de propiedades buscará la clase `java.beans.ComplexNumberEditor` class.
 - Añadiendo "Editor" al nombre de la clase y buscando la clase en el path de búsqueda. El path de búsqueda por defecto del `BeanBox` está en `sun.beans.editors`.

Los Editores de Propiedades del BDK

El DBK proporciona editores para tipos de datos primitivos como `int`, `bool`, `float`, y para los tipos de clases como `Color` y `Font`. El código fuente para estos editores está en `beans/apis/sun/beans/editors`. Estos fuentes son un buen punto de entrada para escribir tus propios editores. Algunas cosas a observar sobre los editores de propiedades del BDK:

- Todas las propiedades "numéricas" están representadas como objetos String. El IntEditor sobrescribe PropertyEditorSupport.setAsText().
- El editor de propiedades bool es un menú de elecciones discretas. Sobreescribiendo el método PropertyEditorSupport.getTags() para que devuelva un String[] que contenga "True" y "False":

```
public String[] getTags() {
    String result[] = { "True", "False" };
    return result;
}
```

- Los editores de propiedades Color y Font implementan un editor personalizado. Porque estos objetos requieren un interface más sofisticado para ser fácilmente editados en un componente separado que se despliega para hacer la edición de la propiedad. Sobreescribir supportsCustomEditor() para que devuelva true indica a la hoja de propiedades que este editor de propiedad es un componente personalizado. Los métodos isPaintable() y paintValue() también son sobrescritos para proporcionar el color y la fuente en las áreas de ejemplo de la hoja de propiedades.

El código fuente de estos editores de propiedades está en beans/apis/sun/beans/editors.

Observa que si no se encuentra un editor de propiedades para una propiedad, el BeanBox no mostrará esa propiedad en la hoja de propiedades.

Personalizadores

Cuando se utiliza un Personalizador de Bean, se obtiene el control completo sobre la configuración o edición del Bean. Un personalizador es como una aplicación que específicamente contiene la personalización de un Bean. Algunas veces las propiedades son insuficientes para representar los atributos configurables de un Bean. Los Personalizadores se usan cuando se necesita utilizar instrucciones sofisticadas para modificar un Bean, y cuando los editores de propiedades son demasiados primitivos para conseguir la personalización del Bean.

Todos los personalizadores deben:

- Descender de java.awt.Component o de una de sus subclases.
- Implementar el interface java.beans.Customizer. Esto significa la implementación de métodos para registrar objetos PropertyChangeListener, y lanzar eventos de cambio de propiedad a dichos oyentes cuando ocurre un cambio en el Bean fuente.
- Implementar un constructor por defecto.
- Asociar el personalizador con la clase fuente mediante BeanInfo.getBeanDescriptor().

Si un Bean asociado con un Personalizador es arrastrado dentro del BeanBox, podrás ver un ítem "Customize..." en el menú Edit.

Personalizadores del BDK

El OurButtonCustomizer sirve como ejemplo para demostrar el mecanismo de construcción de un personalizador. OurButtonCustomizer:

- Desciende de java.awt.Panel (una subclase de Component).
- Implementa el interface Customizer, y utiliza un objeto PropertyChangeSupport para manejar el registro y notificación de PropertyChangeListener. Puedes ver la página [Propiedades Compartidas](#) para una descripción de PropertyChangeSupport.
- Implementa un constructor por defecto:

```
public OurButtonCustomizer() {  
    setLayout(null);  
}
```

- Esta asociado con su clase fuente, ExplicitButton, mediante ExplicitButtonBeanInfo. Aquí tienes las sentencias de ExplicitButtonBeanInfo que hacen la asociación:

```
public BeanDescriptor getBeanDescriptor() {  
    return new BeanDescriptor(beanClass, customizerClass);  
}  
...  
private final static Class customizerClass =  
    OurButtonCustomizer.class;
```

Los Beans BridgeTester y JDBC Select también tienen personalizadores.

Persistencia de un Bean

Un Bean persiste cuando tiene sus propiedades, campos e información de estado almacenadas y restauradas desde un fichero. El mecanismo que hace posible la persistencia se llama serialización. Cuando un ejemplar de bean es serializado se convierte en una canal de datos para ser escritos. Cualquier Applet, aplicación o herramienta que utilice el Bean puede "reconstituirlo" mediante la deserialización. Los JavaBeans utilizan el API [Object Serialization](#) del JDK para sus necesidades de serialización.

Siempre que una clase en el árbol de herencia implemente los interfaces Serializable o Externalizable, esa clase será serializable.

Todos los Bean deben persistir. Para persistir, nuestros Beans deben soportar la serialización implementando los interfaces [java.io.Serializable](#) o [java.io.Externalizable](#). Estos interfaces te ofrecen la elección entre serialización automática y "hazlo tu mismo".

Controlar la Serialización

Se puede controlar el nivel de serialización de nuestro Bean:

- Automático: implementando Serializable. Todo es serializado.
- Excluyendo los campos que no se quieran serializar marcándolos con el modificador transient (o static).
- Escribir los Beans a un fichero de formato específico: implementando Externalizable, y sus dos métodos.

Serialización por Defecto: El Interface Serializable

El interface Serializable proporciona serialización automática mediante la utilización de las herramientas de Java Object Serialization. Serializable no declara métodos; actúa como un marcador, diciéndole a las herramientas de Serialización de Objetos que nuestra clase Bean es serializable. Marcar las clases con Serializable significa que le estamos diciendo a la Máquina Virtual Java (JVM) que estamos seguros de que nuestra clase funcionará con la serialización por defecto. Aquí tenemos algunos puntos importantes para el trabajo con el interface Serializable:

- Las clases que implementan Serializable deben tener un constructor sin argumentos. Este constructor será llamado cuando un objeto sea "reconstituido" desde un fichero .ser.
- No es necesario implementar Serializable en nuestra subclase si ya está implementado en una superclase.
- Todos los campos excepto static y transient son serializados. Utilizaremos el modificador transient para especificar los campos que no queremos serializar, y para especificar las clases que no son serializables.

El BeanBox escribe los Beans serializables a un fichero con la extensión .ser.

El Bean OurButton utiliza la serialización por defecto para conseguir la persistencia de sus propiedades. OurButton sólo añade Serializable a su definición de clase para hacer

uso de la serialización por defecto:

```
public class OurButton extends Component implements Serializable,...
```

Si arrastramos un ejemplar de OurButton al BeanBox, la hoja de propiedades muestra las propiedades de OurButton. Lo que nos asegura que la serialización está funcionando.

1. Cambia algunas de las propiedades de OurButton. Por ejemplo cambia el tamaño de la fuente y los colores.
2. Serializa el ejemplar de OurButton modificado seleccionando el menú File|SerializeComponent... del BeanBox. Aparece un dialogo para guardar el fichero.
3. Pon el fichero .ser en un fichero JAR con un manifiesto adecuado.
4. Limpia el BeanBox seleccionando el menú File|Clear.
5. Recarga el ejemplar serializado el menú File|LoadJar.

El ejemplar OurButton aparecerá en el BeanBox con las propiedades modificadas intactas. Implementando Serializable en nuestra clase, las propiedades primitivas y los campos pueden ser serializados. Para miembros de la clase más complejos se necesita utilizar técnicas diferentes.

Serialización Selectiva Utilizando el Modificador transient

Para excluir campos de la serialización de un objeto Serializable, marcaremos los campos con el modificador transient:

```
transient int Status;
```

La serialización por defecto no serializa los campos transient y static.

Serialización Selectiva: writeObject y readObject()

Si nuestra clase serializable contiene alguno de los siguientes métodos (las firmas deben ser exactas), el serialización por defecto no tendrá lugar:

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException;
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

Se puede controlar cómo se serializarán los objetos más complejos, escribiendo nuestras propias implementación de los métodos writeObject() y readObject(). Implementaremos writeObject cuando necesites ejercer mayor control sobre la serialización, cuando necesitemos serializar objetos que la serialización por defecto no puede manejar, o cuando necesitamos añadir datos a un canal de serialización que no son miembros del objeto. Implementaremos readObject() para reconstruir el canal de datos para escribir con writeObject().

Ejemplo: el Bean Molecule

El Bean Molecule mantiene un número de versión en un campo estático. Como los campos estáticos no son serializables por defecto, `writeObject()` y `readObject()` son implementados para serializar este campo. Aquí están las implementaciones de `writeObject()` y `readObject()` de `Molecule.java`:

```
private void writeObject(java.io.ObjectOutputStream s)
                        throws java.io.IOException {
    s.writeInt(ourVersion);
    s.writeObject(moleculeName);
}

private void readObject(java.io.ObjectInputStream s)
                        throws java.lang.ClassNotFoundException,
                                java.io.IOException {
    // Compensate for missing constructor.
    reset();
    if (s.readInt() != ourVersion) {
        throw new IOException("Molecule.readObject: version mismatch");
    }
    moleculeName = (String) s.readObject();
}
```

Estas implementaciones limitan los campos serializados a `ourVersion` y `moleculeName`. Ningún otro dato de la clase será serializado.

Es mejor utilizar los métodos `defaultWriteObject()` y `defaultReadObject` de `ObjectInputStream` antes de hacer nuestras propias especificaciones en el canal de escritura. Por ejemplo:

```
private void writeObject(java.io.ObjectOutputStream s)
                        throws java.io.IOException {
    //First write out defaults
    s.defaultWriteObject();
    //...
}

private void readObject(java.io.ObjectInputStream s)
                        throws java.lang.ClassNotFoundException,
                                java.io.IOException {
    //First read in defaults
    s.defaultReadObject();
    //...
}
```

El Interface Externalizable

Este interface se utiliza cuando necesitamos un completo control sobre la serialización de nuestro Bean (por ejemplo, cuando lo escribimos y leemos en un formato de fichero específico). Necesitamos implementar dos métodos: `readExternal()` y

`writeExternal()`. Las clases `Externalizable` también deben tener un constructor sin argumentos.

Ejemplo: Los Beans `BlueButton` y `OrangeButton`

Cuando ejecutamos `BeanBox`, podemos ver dos Beans llamados `BlueButton` y `OrangeButton` en el `ToolBox`. Estos dos Beans son realmente dos ejemplares serializados de la clase `ExternalizableButton`.

`ExternalizableButton` implementa el interface `Externalizable`. Esto significa que hace toda su propia serialización, mediante la implementación de `Externalizable.readExternal()` y `Externalizable.writeExternal()`.

El programa `BlueButtonWriter` es utilizado por los `makefile` de los botones para crear un ejemplar de `ExternalizableButton`, cambiar su propiedad `background` a azul, escribir el Bean en un fichero `BlueButton.ser`. `OrangeButton` se crea de la misma manera, utilizando `OrangeButtonWriter`. El `makefile` pone estos ficheros `.ser` en `buttons.jar`, donde el `ToolBox` puede encontrarlos y reconstituirlos.

Nuevas Características de JavaBeans

Glasgow es una versión del BDK, cuyas características son:

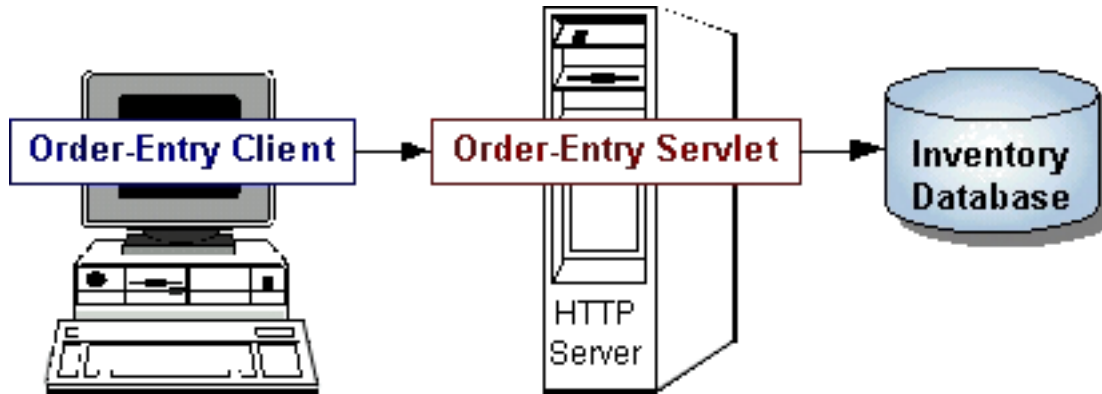
- El Java Activation Framework (JAF). El JAF es un tipo de datos y un API de registro de comandos. Con el JAF se puede descubrir el tipo de datos de un objeto arbitrario, y buscar comandos, aplicaciones o Beans que puedan procesar ese tipo de datos, y activar el comando apropiado a la gestión del usuario. Por ejemplo, un navegador puede identificar el tipo de los datos de un fichero, y lanzar el plug-in adecuado para ver o editar el fichero. El JAF es una extensión del estándar de Java. Se puede obtener una copia desde el site de [Glasgow web](#).
- El Protocolo de Contenidos y Servicios, también conocido como beancontext. Antes de este protocolo un Bean solo conocia y tenia acceso a la Máquina Virtual Java (JVM) en la que el Bean se ejecutaba, y al corazón del API Java. Este protocolo presenta una forma estándar para que los Beans puedan descubrir y acceder a los atributos o servicios proporcionados por un entorno del Bean, para descubrir y acceder a los atributos y servicios del Bean.

El API `java.beans.beancontext` presenta la posibilidad de anidar Beans y contextos de Beans en una estructura de árbol. En el momento de la ejecución, un Bean puede obtener servicios desde su entorno contenedor; el Bean puede asegurarse estos servicios, o propagar estos servicios a cualquier Bean que él contenga.

- Soporte de "Drag and Drop" . El API `java.awt.dnd` proporciona soporte para "Arrastrar y Soltar" entre aplicaciones Java y aplicaciones nativas de la plataforma.

Introducción a los Servlets

Los Servlets son módulos que extienden los servidores orientados a petición-respuesta, como los servidores web compatibles con Java. Por ejemplo, un servlet podría ser responsable de tomar los datos de un formulario de entrada de pedidos en HTML y aplicarle la lógica de negocios utilizada para actualizar la base de datos de pedidos de la compañía.



Los Servlets son para los servidores lo que los applets son para los navegadores. Sin embargo, al contrario que los applets, los servlets no tienen interface gráfico de usuario.

Los servlets pueden ser incluidos en muchos servidores diferentes porque el API Servlet, el que se utiliza para escribir Servlets, no asume nada sobre el entorno o protocolo del servidor. Los servlets se están utilizando ampliamente dentro de servidores HTTP; muchos [servidores Web soportan el API Servlet](#).

Utilizar Servlets en lugar de Scripts CGI!

Los Servlets son un reemplazo efectivo para los scripts CGI. Proporcionan una forma de generar documentos dinámicos que son fáciles de escribir y rápidos en ejecutarse. Los Servlets también solucionan el problema de hacer la programación del lado del servidor con APIs específicos de la plataforma: están desarrollados con el API Java Servlet, una extensión estándar de Java.

Por eso se utilizan los servlets para manejar peticiones de cliente HTTP. Por ejemplo, tener un servlet procesando datos POSTeados sobre HTTP utilizando un formulario HTML, incluyendo datos del pedido o de la tarjeta de crédito. Un servlet como este podría ser parte de un sistema de procesamiento de pedidos, trabajando con bases de datos de productos e inventarios, y quizás un sistema de pago on-line.

Otros usos de los Servlets

- Permitir la colaboración entre la gente. Un servlet puede manejar múltiples peticiones concurrentes, y puede sincronizarlas. Esto permite a los servlets

soportar sistemas como conferencias on-line

- Reenviar peticiones. Los Servlets pueden reenviar peticiones a otros servidores y servlets. Con esto los servlets pueden ser utilizados para cargar balances desde varios servidores que reflejan el mismo contenido, y para particionar un único servicio lógico en varios servidores, de acuerdo con los tipos de tareas o la organización compartida.

Listo para Escribir

Para prepararte a escribir Servlets, esta sección explica:

[Arquitectura del Paquete Servlet](#)

Explica los propósitos de los principales objetos e interfaces del paquete Servlet.

[Un Servlet Sencillo](#)

Muestra la apariencia del código de un servlet sencillo.

[Ejemplos de Servlets](#)

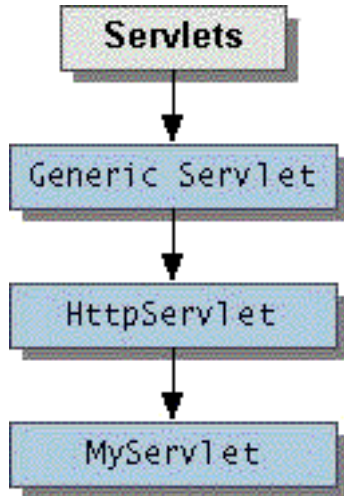
Muestra los ejemplos de Servlets utilizados en el resto de la lección.

Arquitectura del Paquete Servlet

El paquete `javax.servlet` proporciona clases e interfaces para escribir servlets. La arquitectura de este paquete se describe a continuación:

El Interface Servlet

La abstracción central en el API Servlet es el interface [Servlet](#). Todos los servlets implementan este interface, bien directamente o, más comunmente, extendiendo una clase que lo implemente como [HttpServlet](#)



El interface Servlet declara, pero no implementa, métodos que manejan el Servlet y su comunicación con los clientes. Los escritores de Servlets proporcionan algunos de esos métodos cuando desarrollan un servlet.

Interacción con el Cliente

Cuando un servlet acepta una llamada de un cliente, recibe dos objetos:

- Un [ServletRequest](#), que encapsula la comunicación desde el cliente al servidor.
- Un [ServletResponse](#), que encapsula la comunicación de vuelta desde el servlet hacia el cliente.

`ServletRequest` y `ServletResponse` son interfaces definidos en el paquete `javax.servlet`.

El Interface ServletRequest

El Interface `ServletRequest` permite al servlet acceder a :

- Información como los nombres de los parámetros pasados por el cliente, el protocolo (esquema) que está siendo utilizado por el cliente, y los nombres del host remote que ha realizado la petición y la del server que la ha recibido.

- El stream de entrada, [ServletInputStream](#). Los Servlets utilizan este stream para obtener los datos desde los clientes que utilizan protocolos como los métodos POST y PUT del HTTP.

Los interfaces que extienden el interface ServletRequest permiten al servlet recibir más datos específicos del protocolo. Por ejemplo, el interface [HttpServletRequest](#) contiene métodos para acceder a información de cabecera específica HTTP.

El Interface ServletResponse

El Interface ServletResponse le da al servlet los métodos para responder al cliente.

- Permite al servlet seleccionar la longitud del contenido y el tipo MIME de la respuesta.
- Proporciona un stream de salida, [ServletOutputStream](#), y un Writer a través del cual el servlet puede responder datos.

Los interfaces que extienden el interface ServletResponse le dan a los servlets más capacidades específicas del protocolo. Por ejemplo, el interface [HttpServletResponse](#) contiene métodos que permiten al servlet manipular información de cabecera específica HTTP.

Capacidades Adicionales de los Servlets HTTP

Las clases e interfaces descritos anteriormente construyen un servlet básico. Los servlets HTTP tienen algunos objetos adicionales que proporcionan capacidades de seguimiento de sesión. El escritor de servlets pueden utilizar esos APIs para mantener el estado entre el servlet y el cliente persiste a través de múltiples conexiones durante un periodo de tiempo. Los servlets HTTP también tienen objetos que proporcionan cookies. El API cookie se utiliza para guardar datos dentro del cliente y recuperar esos datos.

Un Servlet Sencillo

La siguiente clase define completamente un servlet:

```
public class SimpleServlet extends HttpServlet
{
    /**
     * Maneja el método GET de HTTP para construir una sencilla página Web.
     */
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        PrintWriter          out;
        String               title = "Simple Servlet Output";

        // primero selecciona el tipo de contenidos y otros campos de cabecera de
la respuesta
        response.setContentType("text/html");

        // Luego escribe los datos de la respuesta
        out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>This is output from SimpleServlet.");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Esto es todo!

Las clases mencionadas en la página [Arquitectura del Paquete Servlet](#) se han mostrado en negrita:

- SimpleServlet extiende la clase **HttpServlet**, que implementa el interface **Servlet**.
- SimpleServlet sobreescribe el método **doGet** de la clase **HttpServlet**. Este método es llamado cuando un cliente hace un petición GET (el método de petición por defecto de HTTP), y resulta en una sencilla página HTML devuelta al cliente.
- Dentro del método **doGet**
 - La petición del usuario está representada por un objeto **HttpServletRequest**.
 - La respuesta al usuario esta representada por un objeto **HttpServletResponse**.
 - Como el texto es devuelto al cliente, el respuesta se envía utilizando el objeto **Writer** obtenido desde el objeto **HttpServletResponse**.

Ejemplos de Servlets



Las páginas restantes de esta sección muestran como escribir servlets HTTP. Se asume algún conocimiento del protocolo HTTP; para aprender más sobre este protocolo podrías echar un vistazo al [RFC del HTTP/1.1](#).

Las páginas utilizan un ejemplo llamado Librería de Duke, un sencilla librería on-line que permite a los clientes realizar varias funciones. Cada función está proporcionada por un Servlet:

Función	Servlet
Navegar por los libros de oferta	CatalogServlet
Comprar un libro situándolo en un "tajeta de venta"	CatalogServlet
Obtener más información sobre un libro específico	BookDetailServlet
Manejar la base de datos de la librería	BookDBServlet
Ver los libros que han sido seleccionados para comprar	ShowCartServlet
Eliminar uno o más libros de la tarjeta de compra.	ShowCartServlet
Comprar los libros de la tarjeta de compra	CashierServlet
Recibir un Agradecimiento por la compra	ReceiptServlet

Las páginas utilizan servlets para ilustrar varias tareas. Por ejemplo, el [BookDetailServlet](#) se utiliza para mostrar [cómo manejar peticiones GET de HTTP](#), el [BookDBServlet](#) se utiliza para mostrar [cómo inicializar un servlet](#), y el [CatalogServlet](#) se utiliza para mostrar [el seguimiento de sesión](#).

El ejemplo Duke's Bookstore está compuesto por varios ficheros fuente. Para tu conveniencia puedes bajartelos en un fichero zip para ejecutar el ejemplo, desde la site de SUN.

[Bajarse el fichero ZIP](#)

Para ejecutar el ejemplo, necesitas arrancar [servletrunner](#) o un servidor web, y [llamar al servlet desde un navegador](#)

Interactuar con los Clientes

Un Servlet HTTP maneja peticiones del cliente a través de su método `service`. Este método soporta peticiones estándar de cliente HTTP despachando cada petición a un método designado para manejar esa petición. Por ejemplo, el método `service` llama al método `doGet` mostrado anteriormente en el ejemplo del [servlet sencillo](#).

Peticiones y Respuestas

Esta página explica la utilización de los objetos que representan peticiones de clientes (un objeto `HttpServletRequest`) y las respuestas del servlet (un objeto `HttpServletResponse`). Estos objetos se proporcionan al método `service` y a los métodos que `service` llama para manejar peticiones HTTP.

Manejar Peticiones GET y POST

Los métodos en los que delega el método `service` las peticiones HTTP, incluyen

- `doGet`, para manejar GET, GET condicional, y peticiones de HEAD
- `doPost`, para manejar peticiones POST
- `doPut`, para manejar peticiones PUT
- `doDelete`, para manejar peticiones DELETE

Por defecto, estos métodos devuelven un error `BAD_REQUEST` (400). Nuestro servlet debería sobrescribir el método o métodos diseñados para manejar las interacciones HTTP que soporta. Esta sección muestra cómo implementar método para manejar las peticiones HTTP más comunes: GET y POST.

El método `service` de `HttpServletRequest` también llama al método `doOptions` cuando el servlet recibe una petición `OPTIONS`, y a `doTrace` cuando recibe una petición `TRACE`. La implementación por defecto de `doOptions` determina automáticamente que opciones HTTP son soportadas y devuelve esa información. La implementación por defecto de `doTrace` realiza una respuesta con un mensaje que contiene todas las cabeceras enviadas en la petición `trace`. Estos métodos no se sobrescriben normalmente.

Problemas con los Threads

Los Servlets HTTP normalmente pueden servir a múltiples clientes concurrentes. Si los métodos de nuestro Servlet no funcionan con clientes que acceden a recursos compartidos, deberemos:

- Sincronizar el acceso a estos recursos, o

- Crear un servlet que maneje sólo una petición de cliente a la vez.

Esta lección te muestra cómo implementar la segunda opción. (la primera está cubierta en la página [Threads de Control](#).)

Descripciones de Servlets

Además de manejar peticiones de cliente HTTP, los servlets también son llamados para suministrar descripción de ellos mismos. Esta página muestra como proporcionar una descripción sobreescribiendo el método `getServletInfo`, que suministra una descripción del servlet.

Peticiones y Respuestas

Los métodos de la clase `HttpServletRequest` que manejan peticiones de cliente toman dos argumentos:

1. Un objeto `HttpServletRequest`, que encapsula los datos desde el cliente.
2. Un objeto `HttpServletRequest`, que encapsula la respuesta hacia el cliente.

Objetos `HttpServletRequest`

Un objeto `HttpServletRequest` proporciona acceso a los datos de cabecera HTTP, como cualquier cookie encontrada en la petición, y el método HTTP con el que se ha realizado la petición. El objeto `HttpServletRequest` también permite obtener los argumentos que el cliente envía como parte de la petición.

Para acceder a los datos del cliente

- El método `getParameter` devuelve el valor de un parámetro nombrado. Si nuestro parámetro pudiera tener más de un valor, deberíamos utilizar `getParameterValues` en su lugar. El método `getParameterValues` devuelve un array de valores del parámetro nombrado. (El método `getParameterNames` proporciona los nombres de los parámetros.
- Para peticiones GET de HTTP, el método `getQueryString` devuelve en un `String` una línea de datos desde el cliente. Debemos analizar estos datos nosotros mismos para obtener los parámetros y los valores.
- Para peticiones POST, PUT, y DELETE de HTTP:
 - Si esperamos los datos en formato texto, el método `getReader` devuelve un `BufferedReader` utilizado para leer la línea de datos.
 - Si esperamos datos binarios, el método `getInputStream` devuelve un `ServletInputStream` utilizado para leer la línea de datos.

Nota: Se debe utilizar el método `getParameter[Values]` o uno de los métodos que permitan analizar los datos. No pueden utilizarse juntos en una única petición.

Objetos `HttpServletResponse`

Un objeto `HttpServletResponse` proporciona dos formas de devolver datos al usuario:

- El método `getWriter` devuelve un `Writer`
- El método `getOutputStream` devuelve un `ServletOutputStream`

Se utiliza el método `getWriter` para devolver datos en formato texto al usuario y el método `getOutputStream` para devolver datos binarios.

Si cerramos el Writer o el ServletOutputStream después de haber enviado la respuesta, permitimos al servidor saber cuando la respuesta se ha completado.

Cabecera de Datos HTTP

Debemos seleccionar la cabecera de datos HTTP antes de acceder a Writer o a OutputStream. La clase HttpServletResponse proporciona métodos para acceder a los datos de la cabecera. Por ejemplo, el método `setContentType` selecciona el tipo del contenido. (Normalmente esta es la única cabecera que se selecciona manualmente).

Ozito

Manejar Peticiones GET y POST

Para manejar peticiones HTTP en un servlet, extendemos la clase `HttpServlet` y sobrescribimos los métodos del servlet que manejan las peticiones HTTP que queremos soportar. Esta página ilustra el manejo de peticiones GET y POST. Los métodos que manejan estas peticiones son `doGet` y `doPost`.

Manejar Peticiones GET

Manejar peticiones GET implica sobrescribir el método `doGet`. El siguiente ejemplo muestra a `BookDetailServlet` haciendo esto. Los métodos explicados en [Peticiones y Respuestas](#) se muestran en negrita:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        // selecciona el tipo de contenido en la cabecera antes de acceder a Writer
response.setContentType("text/html");
PrintWriter out = response.getWriter();

        // Luego escribe la respuesta
        out.println("<html>" +
                    "<head><title>Book Description</title></head>" +
                    ...);

        //Obtiene el identificador del libro a mostrar
String bookId = request.getParameter("bookId");
        if (bookId != null) {
            // Y la información sobre el libro y la imprime
            ...
        }
        out.println("</body></html>");
out.close();
    }
    ...
}
```

El servlet extiende la clase `HttpServlet` y sobrescribe el método `doGet`. Dentro del método `doGet`, el método `getParameter` obtiene los argumentos esperados por el servlet.

Para responder al cliente, el método `doGet` utiliza un `Writer` del objeto `HttpServletResponse` para devolver datos en formato texto al cliente. Antes de acceder al `writer`, el ejemplo selecciona la cabecera del tipo del contenido. Al final del método `doGet`, después de haber enviado la respuesta, el `Writer` se cierra.

Manejar Peticiones POST

Manejar peticiones POST implica sobrescribir el método `doPost`. El siguiente ejemplo muestra a `ReceiptServlet` haciendo esto. De nuevo, los métodos explicados en [Peticiones y Respuestas](#) se muestran en negrita:

```

public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        // selecciona la cabecera de tipo de contenido antes de acceder a Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Luego escribe la respuesta
        out.println("<html>" +
                    "<head><title> Receipt </title>" +
                    ...);

        out.println("<h3>Thank you for purchasing your books from us " +
                    request.getParameter("cardname") +
                    ...);
        out.close();
    }
    ...
}

```

El servlet extiende la clase `HttpServlet` y sobrescribe el método `doPost`. Dentro del método `doPost`, el método `getParameter` obtiene los argumentos esperados por el servlet.

Para responder al cliente, el método `doPost` utiliza un `Writer` del objeto `HttpServletResponse` para devolver datos en formato texto al cliente. Antes de acceder al `writer`, el ejemplo selecciona la cabecera del tipo de contenido. Al final del método `doPost`, después de haber enviado la respuesta, el `Writer` se cierra.

Problemas con los Threads

Los servlets HTTP normalmente pueden servir a múltiples clientes concurrentemente. Si los métodos de nuestro servlet trabajan con clientes que acceden a recursos compartidos, podemos manejar la concurrencia creando un servlet que maneje sólo una petición de cliente a la vez. (También se puede sincronizar el acceso a los recursos, un punto que se cubre en la sección [Threads de Control](#) de este tutorial).

Para hacer que el servlet maneje sólo un cliente a la vez, tiene que implementar el interface [SingleThreadModel](#) además de extender la clase `HttpServlet`.

Implementar el interface `SingleThreadModel` no implica escribir ningún método extra. Sólo se declara que el servlet implementa el interface, y el servidor se asegura de que nuestro servlet sólo ejecute un método `service` cada vez.

Por ejemplo, el `ReceiptServlet` acepta un nombre de usuario y un número de tarjeta de crédito, y le agradece al usuario su pedido. Si este servlet actualizara realmente una base de datos, por ejemplo, una que siga la pista del inventario, entonces la conexión con la base de datos podría ser un recurso compartido. El servlet podría sincronizar el acceso a ese recurso, o implementar el interface `SingleThreadModel`. Si el servlet implementa este interface, el único cambio en el código de [la página anterior](#) es la línea mostrada en negrita:

```
public class ReceiptServlet extends HttpServlet
    implements SingleThreadModel {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        ...
    }
    ...
}
```

Proporcionar una Descripción de Servlet

Algunas aplicaciones, como [La Herramienta de Administración del Java Web Server](#), obtienen información sobre el servlet y la muestran. La descripción del servlet es un string que puede describir el proposito del servlet, su autor, su número de versión, o aquello que el autor del servlet considere importante.

El método que devuelve esta información es `getServletInfo`, que por defecto devuelve null. No es necesario sobrescribir este método, pero las aplicaciones no pueden suministrar descripción de nuestro servlet a menos que nosotros lo hagamos.

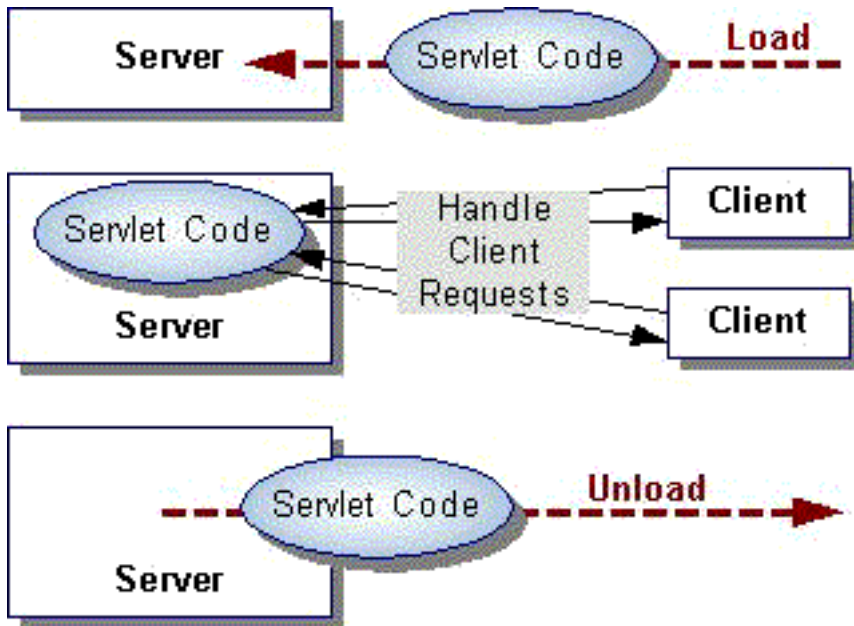
El siguiente ejemplo muestra la descripción de `BookStoreServlet`:

```
public class BookStoreServlet extends HttpServlet {  
    ...  
    public String getServletInfo() {  
        return "The BookStore servlet returns the " +  
            "main web page for Duke's Bookstore.";  
    }  
}
```

El Ciclo de Vida de un Servlet

Cada servlet tiene el mismo ciclo de vida:

- Un servidor carga e inicializa el servlet.
- El servlet maneja cero o más peticiones de cliente.
- El servidor elimina el servlet. (Algunos servidores sólo cumplen este paso cuando se desconectan).



Inicializar un Servlet

Cuando un servidor carga un servlet, ejecuta el método `init` del servlet. La inicialización se completa antes de manejar peticiones de clientes y antes de que el servlet sea destruido.

Aunque muchos servlets se ejecutan en servidores multi-thread, los servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método `init`, cuando carga el servlet, y no lo llamará de nuevo a menos que vuelva a recargar el servlet. El servidor no puede recargar un servlet sin primero haber destruido el servlet llamando al método `destroy`.

Interactuar con Clientes

Después de la inicialización, el servlet puede manejar peticiones de clientes. Esta parte del ciclo de vida de un servlet se pudo ver en la [sección anterior](#).

Destruir un Servlet

Los servlets se ejecutan hasta que el servidor los destruye, por ejemplo, a petición del administrador del sistema. Cuando un servidor destruye un servlet, ejecuta el método `destroy` del propio servlet. Este método sólo se ejecuta una vez. El

servidor no ejecutará de nuevo el servlet, hasta haberlo cargado e inicializado de nuevo.

Mientras se ejecuta el método destroy, otro thread podría estar ejecutando una petición de servicio. La página [Manejar Threads de Servicio a la Terminación de un Thread](#) muestra como proporcionar limpieza cuando threads de larga ejecución podrían estar ejecutando peticiones de servicio.

Ozito

Inicializar un Servlet

El método `init` proporcionado por la clase `HttpServlet` inicializa el servlet y graba la inicialización. Para hacer una inicialización específica de nuestro servlet, debemos sobrescribir el método `init` siguiendo estas reglas:

- Si ocurre un error que haga que el servlet no pueda manejar peticiones de cliente, lanzar una `UnavailableException`.

Un ejemplo de este tipo de error es la imposibilidad de establecer una conexión requerida.

- No llamar al método `System.exit`.
- Guardar el parámetro `ServletConfig` para que el método `getServletConfig` pueda devolver su valor.

La forma más sencilla de hacer esto es hacer que el nuevo método `init` llame a `super.init`. Si grabamos el objeto nosotros mismos, debemos sobrescribir el método `getServletConfig` para devolver el objeto desde su nueva posición.

Aquí hay un ejemplo del método `init`:

```
public class BookDBServlet ... {  
  
    private BookstoreDB books;  
  
    public void init(ServletConfig config) throws ServletException {  
  
        // Store the ServletConfig object and log the initialization  
        super.init(config);  
  
        // Load the database to prepare for requests  
        books = new BookstoreDB();  
    }  
    ...  
}
```

El método `init` es bastante sencillo: llama al método `super.init` para manejar el objeto `ServletConfig` y grabar la inicialización, y seleccionar un campo privado.

Si el `BookDBServlet` utilizará una base de datos real, en vez de simularla con un objeto, el método `init` sería más complejo. Aquí puedes ver el pseudo-código de como podría ser ese método `init`:

```
public class BookDBServlet ... {  
  
    public void init(ServletConfig config) throws ServletException {  
  
        // Store the ServletConfig object and log the initialization  
        super.init(config);  

```



```

        // Open a database connection to prepare for requests
        try {
            databaseUrl = getInitParameter("databaseUrl");
            ... // get user and password parameters the same way
            connection = DriverManager.getConnection(databaseUrl,
                                                    user, password);
        } catch (Exception e) {
            throw new UnavailableException (this,
            "Could not open a connection to the database");
        }
    }
    ...
}

```

Parámetros de Inicialización

Le segunda versión del método `init` llama al método `getInitParameter`. Este método toma el nombre del parámetro como argumento y devuelve un `String` que representa su valor.

(La especificación de parámetros de inicialización es específica del servidor. Por ejemplo, los parámetros son especificados como una propiedad cuando un servlet se ejecuta con el `ServletRunner`. La página [La Utilidad `servletrunner`](#) contiene una explicación general de las propiedades y cómo crearlas).

Si por alguna razón, necesitamos obtener los nombres de los parámetros, podemos utilizar el método `getParameterNames`.

Destruir un Servlet

El método `destroy` proporcionado por la clase `HttpServlet` destruye el servlet y graba su destrucción. Para destruir cualquier recurso específico de nuestro servlet, debemos sobrescribir el método `destroy`. Este método debería deshacer cualquier trabajo de inicialización y cualquier estado de persistencia sincronizado con el estado de memoria actual.

El siguiente ejemplo muestra el método `destroy` que acompaña el método `init` de la página anterior:

```
public class BookDBServlet extends GenericServlet {  
  
    private BookstoreDB books;  
  
    ... // the init method  
  
    public void destroy() {  
        // Allow the database to be garbage collected  
        books = null;  
    }  
}
```

Un servidor llama al método `destroy` después de que se hayan completado todas las llamadas de servidor, o en un servidor específico hayan pasado un número de segundos, lo que ocurra primero. Si nuestro servlet manejar operaciones de larga ejecución, los métodos `service` se podrían estar ejecutando cuando el servidor llame al método `destroy`. Somos responsables de asegurarnos de que todos los threads han terminado. [La página siguiente](#) muestra cómo.

El método `destroy` mostrado arriba espera a que todas las interacciones de cliente se hayan completado cuando se llama al método `destroy`, porque el servlet no tiene operaciones de larga ejecución.

Grabar el Estado del Cliente

El API Servlet proporciona dos formas de seguir la pista al estado de un cliente:

Seguimiento de Sesión

El seguimiento de sesión es un mecanismo que los servlets utilizan para mantener el estado sobre la serie de peticiones desde un mismo usuario (esto es, peticiones originadas desde el mismo navegador) durante algún periodo de tiempo,.

Cookies

Las Cookies son un mecanismo que el servlet utiliza para mantener en el cliente una pequeña cantidad de información asociada con el usuario. Los servlets pueden utilizar la información del cookie como las entradas del usuario en la site (como una firma de seguridad de bajo nivel, por ejemplo), mientras el usuario navega a través de la site (o como expositor de las preferencias del usuario, por ejemplo) o ambas.

Seguimiento de Sesión

El seguimiento de sesión es un mecanismo que los servlets utilizan para mantener el estado sobre la serie de peticiones desde un mismo usuario (esto es, peticiones originadas desde el mismo navegador) durante un periodo de tiempo.

Las sesiones son compartidas por los servlets a los que accede el cliente. Esto es conveniente para aplicaciones compuestas por varios servlets. Por ejemplo, Duke's Bookstore utiliza seguimiento de sesión para seguir la pista de los libros pedidos por el usuario. Todos los servlets del ejemplo tienen acceso a la sesión del usuario.

Para utilizar el seguimiento de sesión debemos:

- Obtener una sesión (un objeto [HttpSession](#)) para un usuario.
- Almacenar u obtener datos desde el objeto HttpSession.
- Invalidar la sesión (opcional).

Obtener una Sesión

El método getSession del objeto HttpServletRequest devuelve una sesión de usuario. Cuando llamamos al método con su argumento create como true, la implementación creará una sesión si es necesario.

Para mantener la sesión apropiadamente, debemos llamar a getSession antes de escribir cualquier respuesta. (Si respondemos utilizando un Writer, entonces debemos llamar a getSession antes de acceder al Writer, no sólo antes de enviar cualquier respuesta).

El ejemplo Duke's Bookstore utiliza seguimiento de sesión para seguir la pista de los libros que hay en la hoja de pedido del usuario. Aquí tenemos un ejemplo de CatalogServlet obteniendo una sesión de usuario:

```
public class CatalogServlet extends HttpServlet {  
  
    public void doGet (HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException  
    {  
        // Get the user's session and shopping cart  
        HttpSession session = request.getSession(true);  
        ...  
        out = response.getWriter();  
        ...  
    }  
}
```

Almacenar y Obtener Datos desde la Sesión

El Interface HttpSession proporciona métodos que almacenan y recuperan:

- Propiedades de Sesión Estándar, como un identificador de sesión.

- Datos de la aplicación, que son almacenados como parejas nombre-valor, donde el nombre es un string y los valores son objetos del lenguaje de programación Java. Como varios servlets pueden acceder a la sesión de usuario, deberemos adoptar una convención de nombrado para organizar los nombres con los datos de la aplicación. Esto evitará que los servlets sobrescriban accidentalmente otros valores de la sesión. Una de esas convenciones es **servletname.name** donde **servletname** es el nombre completo del servlet, incluyendo sus paquetes. Por ejemplo, com.acme.WidgetServlet.state es un cookie con el servletname com.acme.WidgetServlet y el name state.

El ejemplo Duke's Bookstore utiliza seguimiento de sesión para seguir la pista de los libros de la hoja de pedido del usuario. Aquí hay un ejemplo de CatalogServlet obteniendo un identificador de sesión de usuario, y obteniendo y seleccionando datos de la aplicación asociada con la sesión de usuario:

```
public class CatalogServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the user's session and shopping cart
        HttpSession session = request.getSession(true);
        ShoppingCart cart =
            (ShoppingCart)session.getValue(session.getId());

        // If the user has no cart, create a new one
        if (cart == null) {
            cart = new ShoppingCart();
            session.putValue(session.getId(), cart);
        }

        ...
    }
}
```

Como un objeto puede ser asociado con una sesión, el ejemplo Duke's Bookstore sigue la pista de los libros que el usuario ha pedido dentro de un objeto. El tipo del objeto es ShoppingCart y cada libro que el usuario a seleccionado es almacenado en la hoja de pedidos como un objeto ShoppingCartItem. Por ejemplo, el siguiente código procede del método doGet de CatalogServlet:

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    HttpSession session = request.getSession(true);
    ShoppingCart cart = (ShoppingCart)session.getValue(session.getId());
    ...
    // Check for pending adds to the shopping cart
    String bookId = request.getParameter("Buy");
}
```

```

//If the user wants to add a book, add it and print the result
String bookToAdd = request.getParameter("Buy");
if (bookToAdd != null) {
    BookDetails book = database.getBookDetails(bookToAdd);

    cart.add(bookToAdd, book);
    out.println("<p><h3>" + ...);
}
}

```

Finalmente, observa que una sesión puede ser designada como nueva. Una sesión nueva hace que el método `isNew` de la clase `HttpSession` devuelva `true`, indicando que, por ejemplo, el cliente, todavía no sabe nada de la sesión. Una nueva sesión no tiene datos asociados.

Podemos tratar con situaciones que involucren nuevas sesiones. En el ejemplo Duke's Bookstore, si el usuario no tiene hoja de pedido (el único dato asociado con una sesión), el servlet crea una nueva. Alternativamente, si necesitamos información sobre el usuario al iniciar una sesión (como el nombre de usuario), podríamos querer redireccionar al usuario a un "página de entrada" donde recolectamos la información necesaria.

Invalidar la Sesión

Una sesión de usuario puede ser invalidada manual o automáticamente, dependiendo de donde se esté ejecutando el servlet. (Por ejemplo, el Java Web Server, invalida una sesión cuando no hay peticiones de página por un periodo de tiempo, unos 30 minutos por defecto). Invalidar una sesión significa eliminar el objeto `HttpSession` y todos sus valores del sistema.

Para invalidar manualmente una sesión, se utiliza el método `invalidate` de "session". Algunas aplicaciones tienen un punto natural en el que invalidar la sesión. El ejemplo Duke's Bookstore invalida una sesión de usuario después de que el usuario haya comprado los libros. Esto sucede en el `ReceiptServlet`:

```

public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {

        ...
        scart = (ShoppingCart)session.getValue(session.getId());
        ...
        // Clear out shopping cart by invalidating the session
        session.invalidate();

        // set content type header before accessing the Writer
        response.setContentType("text/html");
        out = response.getWriter();
    }
}

```

```
    ...  
}  
}
```

Manejar todos los Navegadores

Por defecto, el seguimiento de sesión utiliza cookies para asociar un identificador de sesión con un usuario. Para soportar también a los usuarios que acceden al servlet con un navegador que no soporta cookies, o si este está programado para no aceptarlas, debemos utilizar reescritura de URL en su lugar.

Cuando se utiliza la reescritura de URL se llama a los métodos que, cuando es necesario, incluyen el ID de sesión en un enlace. Debemos llamar a esos métodos por cada enlace en la respuesta del servlet.

El método que asocia un ID de sesión con una URL es `HttpServletResponse.encodeUrl`. Si redirecionamos al usuario a otra página, el método para asociar el ID de sesión con la URL redireccionada se llama `HttpServletResponse.encodeRedirectUrl`.

Los métodos `encodeUrl` y `encodeRedirectUrl` deciden si las URL necesitan ser reescritas, y devolver la URL cambiada o sin cambiar. (Las reglas para las URLs y las URLs redireccionadas son diferentes, pero en general si el servidor detecta que el navegador soporta cookies, entonces la URL no se reescribirá).

El ejemplo Duke's Bookstore utiliza reescritura de URL para todos los enlaces que devuelve a sus usuarios. Por ejemplo, el `CatalogServlet` devuelve un catalogo con dos enlaces para cada libro. Un enlace ofrece detalles sobre el libro y el otro ofrece al usuario añadir el libro a su hoja de pedidos. Ambas URLs son reescritas:

```
public class CatalogServlet extends HttpServlet {  
  
    public void doGet (HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException  
    {  
        // Get the user's session and shopping cart, the Writer, etc.  
        ...  
        // then write the data of the response  
        out.println("<html>" + ...);  
        ...  
        // Get the catalog and send it, nicely formatted  
        BookDetails[] books = database.getBooksSortedByTitle();  
        ...  
        for(int i=0; i < numBooks; i++) {  
            ...  
            //Print out info on each book in its own two rows  
            out.println("<tr>" + ...  
  
                "<a href=\"\" +
```

```

        response.encodeUrl("/servlet/bookdetails?bookId=" +
                               bookId) +
        "\"> <strong>" + books[i].getTitle() +
        " </strong></a></td>" + ...

        "<a href=\"\" +
        response.encodeUrl("/servlet/catalog?Buy=" + bookId)
        + "\">    Add to Cart    </a></td></tr>" +
    }
}
}

```

Si el usuario pulsa sobre un enlace con una URL re-escrita, el servlet reconoce y extrae el ID de sesión. Luego el método getSession utiliza el ID de sesión para obtener el objeto HttpSession del usuario.

Por otro lado, si el navegador del usuario no soporta cookies y el usuario pulsa sobre una URL no re-escrita. Se pierde la sesión de usuario. El servlet contactado a través de ese enlace crea una nueva sesión, pero la nueva sesión no tiene datos asociados con la sesión anterior. Una vez que un servlet pierde los datos de una sesión, los datos se pierden para todos los servlets que comparten la sesión. Debemos utilizar la re-escritura de URLs consistentemente para que nuestro servlet soporte clientes que no soportan o aceptan cookies.

Utilizar Cookies

Las Cookies son una forma para que un servidor (o un servlet, como parte de un servidor) envíe información al cliente para almacenarla, y para que el servidor pueda posteriormente recuperar esos datos desde el cliente. Los servlet envían cookies al cliente añadiendo campos a las cabeceras de respuesta HTTP. Los clientes devuelven las cookies automáticamente añadiendo campos a las cabeceras de peticiones HTTP.

Cada cabecera de petición o respuesta HTTP es nombrada como un sólo valor. Por ejemplo, una cookie podría tener un nombre de cabecera BookToBuy con un valor 304qty1, indicando a la aplicación llamante que el usuario quiere comprar una copia del libro con el número 304 en el inventario. (Las cookies y sus valores son específicos de la aplicación).

Varias cookies pueden tener el mismo nombre. Por ejemplo, un servlet podría enviar dos cabeceras llamadas BookToBuy; una podría tener el valor anterior, 304qty1, mientras que la otra podría tener el valor 301qty3. Estas cookies podrían indicar que el usuario quiere comprar una copia del libro con el número 304 en el inventario y tres copias del libro con el número 301 del inventario.

Además de un nombre y un valor, también se pueden proporcionar atributos opcionales como comentarios. Los navegadores actuales no siempre tratan correctamente a los atributos opcionales, por eso ten cuidado con ellos.

Un servidor puede proporcionar una o más cookies a un cliente. El software del cliente, como un navegador, se espera que pueda soportar veinte cookies por host de al menos 4 kb cada una.

Cuando se envía una cookie al cliente, el estándar HTTP/1.0 captura la página que no está en la caché. Actualmente, el `javax.servlet.http.Cookie` no soporta los controles de caché del HTTP/1.1.

Las cookies que un cliente almacena para un servidor sólo pueden ser devueltas a ese mismo servidor. Un servidor puede contener múltiples servlets; el ejemplo Duke's Bookstore está compuesto por varios servlets ejecutándose en un sólo servidor. Como las cookies son devueltas al servidor, los servlets que se ejecutan dentro de un servidor comparten las cookies. Los ejemplos de esta página ilustran esto mostrando como los `servletsCatalogServlet` y `ShowCart` trabajan con los mismos cookies.

Nota: Esta página tiene código que no forma parte del ejemplo Duke's Bookstore. Duke's Bookstore utilizaría código como el de esta página si utilizará cookies en vez de seguimiento de sesión para los pedidos de los clientes. Como las cookies no forman parte de Duke's Bookstore, piensa en los ejemplos de esta página como pseudo-código.

Para enviar una cookie:

1. [Ejemplariza un objeto Cookie.](#)
2. [Selecciona cualquier atributo.](#)
3. [Envía el cookie](#)

Para obtener información de un cookie:

1. [Recupera todos los cookies](#) de la petición del usuario.
2. Busca el cookie o cookies con el nombre que te interesa, utiliza las técnicas de programación estándar.
3. [Obtén los valores de las cookies](#) que hayas encontrado.

Crear un Cookie

El constructor de la clase `javax.servlet.http.Cookie` crea un cookie con un nombre inicial y un valor. Se puede cambiar el valor posteriormente utilizando el método `setValue`.

El nombre del cookie debe ser un token HTTP/1.1. Los tokens son strings que contienen uno de los caracteres especiales listados en [RFC 2068](#). (Strings alfanuméricos cualificados como tokens.) Además, los nombres que empiezan con el carácter dollar ("\$_") están reservados por [RFC 2109](#).

El valor del cookie puede ser cualquier string, aunque no está garantizado que los valores null funcionen en todos los navegadores. Además, si enviamos una cookie que cumpla con las especificaciones originales de las cookies de Netscape, no se deben utilizar caracteres blancos ni ninguno de estos caracteres:

[] () = , " " / ? @ : ;

Si nuestro servlet devuelve una respuesta al usuario con un `Writer`, debemos crear la cookie antes de acceder a `Writer`. (Porque las cookies se envían al cliente como una cabecera, y las cabeceras deben escribirse antes de acceder al `Writer`.)

Si el `CatalogServlet` utilizará cookies para seguir la pista de una hoja de pedido, el servlet podría crear las cookies de esta forma:

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    BookDBServlet database = (BookDBServlet)
        getServletConfig().getServletContext().getServlet("bookdb");

    // Check for pending adds to the shopping cart
    String bookId = request.getParameter("Buy");

    //If the user wants to add a book, remember it by adding a cookie
    if (bookId != null) {
        Cookie getBook = new Cookie("Buy", bookId);
        ...
    }

    // set content-type header before accessing the Writer
    response.setContentType("text/html");
```

```

        // now get the writer and write the data of the response
        PrintWriter out = response.getWriter();
        out.println("<html>" +
                    "<head><title> Book Catalog </title></head>" + ...);
        ...
    }

```

Seleccionar los Atributos de un Cookie

La clase `Cookie` proporciona varios métodos para seleccionar los valores del cookie y sus atributos. La utilización de estos métodos es correcta, están explicados en el javadoc para la clase `Cookie`.

El siguiente ejemplo selecciona el campo `comment` del cookie `CatalogServlet`. Este campo describe el propósito del cookie.

```

public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    //If the user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User wants to buy this book " +
                           "from the bookstore.");
    }
    ...
}

```

También se puede seleccionar la caducidad del cookie. Este atributo es útil, por ejemplo, para borrar un cookie. De nuevo, si Duke's Bookstore utilizará cookies para su hoja de pedidos, el ejemplo podría utilizar este atributo para borrar un libro de la hoja de pedido. El usuario borra un libro de la hoja de pedidos en el `ShowCartServlet`; su código se podría parecer a esto:

```

public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
    if (bookId != null) {
        // Find the cookie that pertains to the book to remove
        ...
    }
}

```

```

        // Delete the cookie by setting its maximum age to zero
        thisCookie.setMaxAge(0);

        ...
    }

    // also set content type header before accessing the Writer
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    //Print out the response
    out.println("<html> <head>" +
        "<title>Your Shopping Cart</title>" + ...);

```

Enviar Cookies

Las cookies se envían como cabeceras en la respuesta al cliente, se añaden con el método `addCookie` de la clase `HttpServletResponse`. Si estamos utilizando un `Writer` para devolver texto, debemos llamar a `addCookie` antes de llamar al método `getWriter` de `HttpServletResponse`.

Continuando con el ejemplo de `CatalogServlet`, aquí está el código para enviar la cookie:

```

public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    //If the user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User has indicated a desire " +
            "to buy this book from the bookstore.");
        response.addCookie(getBook);
    }
    ...
}

```

Recuperar Cookies

Los clientes devuelven las cookies como campos añadidos a las cabeceras de petición HTTP. Para recuperar una cookie, debemos recuperar todas las cookies utilizando el método `getCookies` de la clase `HttpServletRequest`.

El método `getCookies` devuelve un array de objetos `Cookie`, en el que podemos buscar la cookie o cookies que querramos. (Recuerda que distintas cookies pueden tener el mismo nombre, para obtener el nombre de una cookie, utiliza su método `getName`.)

Para continuar con el ejemplo ShowCartServlet:

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    ...

    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
    if (bookId != null) {

        // Find the cookie that pertains to the book to remove
        Cookie[] cookies = request.getCookies();
        ...

        // Delete the book's cookie by setting its maximum age to zero
        thisCookie.setMaxAge(0);
    }

    // also set content type header before accessing the Writer
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    //Print out the response
    out.println("<html> <head>" +
               "<title>Your Shopping Cart</title>" + ...);
}
```

Obtener el valor de una Cookie

Para obtener el valor de una cookie, se utiliza el método `getValue`.

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
    if (bookId != null) {
        // Find the cookie that pertains to that book
        Cookie[] cookies = request.getCookies();
        for(i=0; i < cookies.length; i++) {
            Cookie thisCookie = cookie[i];
            if (thisCookie.getName().equals("Buy") &&
                thisCookie.getValue().equals(bookId)) {

```

```
        // Delete the cookie by setting its maximum age to zero
        thisCookie.setMaxAge(0);
    }
}

// also set content type header before accessing the Writer
response.setContentType("text/html");
PrintWriter out = response.getWriter();

//Print out the response
out.println("<html> <head>" +
            "<title>Your Shopping Cart</title>" + ...);
```

La utilidad servletrunner

Una vez que hemos escrito el servlet, podemos probarlo en la utilidad servletrunner

El servletrunner es un pequeño, proceso multi-thread que maneja peticiones de servlets. Como servletrunner es multi-thread, puede utilizarse para ejecutar varios servlets simultáneamente, o para probar un servlet que llama a otros servlets para satisfacer las peticiones de clientes.

Al contrario que algunos navegadores, servletrunner no recarga automáticamente los servlets actualizados. Sin embargo, podemos parar y reiniciar servletrunner con una pequeña sobrecarga de a pila para ejecutar una nueva versión de un servlet.

[Selecciónar las Propiedades de un Servlet](#)

Podríamos tener que especificar algunos datos para ejecutar un servlet. Por ejemplo, si un servlet necesita parámetros de inicialización, debemos configurar estos datos antes de arrancar servletrunner.

[Arrancar servletrunner](#)

Después de configurar el fichero de propiedades, podemos ejecutar la utilidad servletrunner. Esta página explica cómo.

Seleccionar Propiedades de un Servlet

Las propiedades son parejas de clave-valor utilizadas para la configuración, creación e inicialización de un servlet. Por ejemplo, `servlet.catalog.code=CatalogServlet` es una propiedad cuya clave es `servlet.catalog.code` y cuyo valor es `CatalogServlet`.

La utilidad `servletrunner` tiene dos propiedades para los servlets:

- `servlet.nombre.code`
- `servlet.nombre.initargs`

La propiedad code

El valor de la propiedad `servlet.nombre.code` es el nombre completo de la clase del servlet, incluido su paquete. Por ejemplo:

```
servlet.bookdb.code=database.BookDBServlet
```

La propiedad `servlet.nombre.code` llama a nuestro servlet asociando un nombre (en el ejemplo, `bookdb`) con una clase (en el ejemplo, `database.BookDBServlet`).

La propiedad initargs

El valor de la propiedad `servlet.nombre.initArgs` contiene los parámetros de inicialización del servlet. La sintaxis de este parámetro es **parameterName=parameterValue**. La propiedad completa (la pareja completa clave-valor) debe ser una sola línea lógica. Para mejorar la lectura, se puede utilizar la barra invertida para dividir la línea lógica en varias líneas de texto. Por ejemplo, si el servlet `database` leyera datos desde un fichero, el argumento inicial del servlet podría parecerse a esto:

```
servlet.bookdb.initArgs=\ndbfile=servlets/DatabaseData
```

Los parámetros de inicialización múltiples se especifican separados por comas. Por ejemplo, si el servlet `database` se conectará a una base de datos real, sus argumentos iniciales podrían parecerse a esto:

```
servlet.bookdb.initArgs=\nuser=duke,\npassword=dukes_password,\nurl=fill_in_the_database_url
```

El fichero de Propiedades

Las propiedades se almacenan en un fichero de texto con un nombre por defecto de `servlet.properties`. (Se puede especificar otro nombre cuando se arranca `servletrunner`.) El fichero guarda las propiedades para todos los servlets que se

ejecuten en el servletrunner. Aquí puedes ver el fichero de propiedades para el ejemplo Duke's Bookstore:

```
# This file contains the properties for the Duke's Bookstore servlets.

# Duke's Book Store -- main page
servlet.bookstore.code=BookStoreServlet

# The servlet that manages the database of books
servlet.bookdb.code=database.BookDBServlet

# View all the books in the bookstore
servlet.catalog.code=CatalogServlet

# Show information about a specific book
servlet.bookdetails.code=BookDetailServlet

# See the books that you've chosen to buy
servlet.showcart.code=ShowCartServlet

# Collects information for buying the chosen books
servlet.cashier.code=CashierServlet

# Provide a receipt to the user who's bought books
servlet.receipt.code=ReceiptServlet
```

Arrancar servletrunner

El servletrunner está en el directorio <jsdk> /bin. Se podrá ejecutar más fácilmente si lo ponemos en el path. Por ejemplo:

```
% setenv PATH /usr/local/jsdk/bin: (para UNIX)
```

```
C> set PATH=C:\jsdk\bin;%PATH% (para Win32)
```

Llamar a servletrunner con la opción -help muestra una ayuda sin ejecutarlo:

```
% servletrunner -help
Usage: servletrunner [options]
Options:
  -p port      the port number to listen on
  -b backlog   the listen backlog
  -m max       maximum number of connection handlers
  -t timeout   connection timeout in milliseconds
  -d dir       servlet directory
  -r root      document root directory
  -s filename  servlet property file name
  -v          verbose output
%
```

Para ver los valores por defecto de estas opciones, podemos llamar a servletrunner con la opción -v. Esto arranca la utilidad, se debe parar inmediatamente si una vez obtenida la información no estamos listos para ejecutarlo. o si queremos ejecutar algo distinto de los valores por defecto. Por ejemplo, en Unix, utilizando el comando kill para parar servletrunner.

```
% servletrunner -v
Server settings:
  port = 8080
  backlog = 50
  max handlers = 100
  timeout = 5000
  servlet dir = ./examples
  document dir = ./examples
  servlet propfile = ./examples/servlet.properties
```

Nota: En los valores por defecto mostrados arriba. servlet dir, document dir y el directorio servlet propfile contienen un punto ("."). El punto designa el directorio de trabajo actual. Normalmente este directorio es desde donde se arranca el ejecutable. Sin embargo, en este caso, el punto se refiere al directorio donde está instalado el "servlet

development kit".

Si arrancamos servletrunner desde un directorio distinto al de instalación, servletrunner primero cambia su directorio de trabajo (y, por lo tanto, lo que podrías pensar como el valor de ".").

Una vez que servletrunner está en ejecución, podemos utilizarlo para [probar nuestros servlets](#).

Ozito

Ejecutar Servlets

Esta lección muestra unas cuantas formas de llamar a los servlets:

[Tecleando la URL del servlet en un Navegador Web](#)

Los servlets pueden ser llamados directamente tecleando su URL en un navegador Web. Así es como se accede a la página principal del ejemplo Duke's Bookstore. Esta página muestra la forma general de la URL de un servlet.

[Llamar a un Servlet desde dentro de una página HTML](#)

Las URLs de los servlets pueden utilizarse en etiquetas HTML, donde se podría encontrar una URL de un script CGI-bin o una URL de fichero. Esta página muestra como utilizar la URL de un servlet como destino de un enlace, como la acción de un formulario, y como la localización a utilizar cuando META tag dice que la página sea refrescada. Esta sección asume conocimientos de HTML.

[Desde otro servlet](#)

Los Servlets pueden llamar a otros servlets. Si los dos servlets están en distinto servidor, uno puede hacer peticiones HTTP al otro. Si los dos se ejecutan en el mismo servidor, entonces un servlet puede llamar a los métodos públicos del otro directamente.

Estas páginas asumen que:

- Nuestra máquina, localhost, está ejecutando [servletrunner](#) o un servidor con soporte para servlets, como [Java Web Server](#) en el puerto 8080.
- El ejemplo, Duke's Bookstore, está localizado en el nivel superior del directorio de procesos para los servlets. Para servletrunner, esto significa que los ficheros class están en el directorio servlet especificado por [la opción -d](#).

Si estas dos condiciones se cumplen, podremos ejecutar el servlet de ejemplo tecleando las URLs dadas en el ejemplo.

Llamar a Servlets desde un Navegador

La URL de un servlet tiene la siguiente forma general, donde nombre-servlet corresponde al [nombre que le hemos dado a nuestro servlet](#):

```
http://nombre-de-máquina:puerto/servlet/nombre-servlet
```

Por ejemplo, el servlet que lanza la página principal de Duke's Bookstore tiene la propiedad `servlet.bookstore.code=BookStoreServlet`. Para ver la página principal, teclearemos esta URL en nuestro navegador:

```
http://localhost:8080/servlet/bookstore
```

Las URLs de servlets pueden contener preguntas, como las peticiones GET de HTTP. Por ejemplo, el servlet que sirve los detalles sobre un libro particular toma el número de inventario del libro como pregunta. El nombre del servlet es `bookdetails`; la URL del servlet para obtener (GET) y mostrar toda la información sobre las características de un libro:

```
http://localhost:8080/servlet/bookdetails?bookId=203
```

Llamar a Servlets desde una Página HTML

Para invocar un servlet desde dentro de una página HTML se utiliza la URL del servlet en la etiqueta HTML apropiada.

Esta página utiliza los servlets ShowCart, Cashier, y Receipt de Duke's Bookstore. Afortunadamente este es el orden en que se verán los servlets cuando miremos nuestra hoja y compremos nuestros libros.

Para un acceso más directo al servlet ShowCart servlet, pulsa el enlace Show Cart que hay en la página principal del Duke's Bookstore. Si tenemos servletrunner o un servidor web configurados para ejecutar el ejemplo, vayamos a la página principal de la librería mostrada en la [página anterior](#). Sólo por diversión, podríamos añadir un libro a nuestra hoja de pedido antes de acceder al servlet ShowCart.

Ejemplos de URLs de Servlets en etiquetas HTML

La página devuelta por ShowCartServlet tiene varios enlaces, cada uno de los cuales tiene un servlet como destino. Aquí podemos ver el código de esos enlaces:

```
public class ShowCartServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        out.println(... +
                    "<a href=\"\" +
                    response.encodeUrl("/servlet/cashier") +
                    "\">Check Out</a>      " +
                    ...);
        ...
    }
    ...
}
```

Este código resulta en una página HTML que tiene el siguiente enlace:

```
<a href="http://localhost:8080/servlet/cashier>Check Out</a>
```

Si llamamos a la página del showcart, podremos ver el enlace como si vieramos el fuente de la página. Luego pulsamos sobre el enlace. El servlet cashier devolverá la página que contiene el siguiente ejemplo.

La página mostrada por el server cashier presenta un formulario que pide el nombre del usuario y el número de la tarjeta de crédito. El código que imprime el formulario se parece a esto:

```

public class CashierServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        out.println(... +
                     "<form action=\"" +
                     response.encodeUrl("/servlet/receipt") +
                     "\" method=\"" + "post\">" +
                     ...
                     "<td><input type=\"" + "text\" name=\"" + "cardname\"" +
                     "value=\"" + "Gwen Canigetit\" size=\"" + "19\"></td>" +
                     ...
                     "<td><input type=\"" + "submit\"" +
                     "value=\"" + "Submit Information\"></td>" +
                     ...
                     "</form>" +
                     ...);
        out.close();
    }
    ...
}

```

Este código resulta en una página HTML que tiene la siguiente etiqueta para iniciar el formulario:

```
<form action="http://localhost:8080/servlet/receipt" method="post">
```

Si cargamos la página del servlet cashier en nuestro navegador podremos ver la etiqueta que inicia el formulario como si vieramos el fuente de la página. Luego enviamos el formulario. El servlet receipt devolverá una página que contiene el siguiente ejemplo. La página del servlet receipt se resetea a sí misma, por eso si queremos verla, tenemos que hacerlo **rápido!**.

La página devuelta por el servlet receipt tiene una "meta tag" que utiliza una URL de servlet como parte del valor del atributo http-equiv. Específicamente, la etiqueta redirecciona la página hacia a la página principal del Duke's Bookstore despues de dar las gracias al usuario por su pedido. Aquí podemos ver el código de esta etiqueta:

```

public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        out.println("<html>" +

```

```
        "<head><title> Receipt </title>" +  
        "<meta http-equiv=\"refresh\" content=\"4; url=" +  
        "http://" + request.getHeader("Host") +  
        "/servlet/bookstore;\">" +  
        "</head>" +  
        ...  
    }  
    ...  
}
```

Este código resulta en una página HTML que tiene la siguiente etiqueta:

```
<meta http-equiv="refresh"  
      content="4; url=http://localhost:8080/servlet/bookstore;">
```

Ozito

Llamar a un Servlet desde otro Servlet

Para hacer que nuestro servlet llame a otro servlet, podemos:

- Un servlet puede hacer peticiones HTTP a otro servlet. La apertura de una conexión URL se explica en las páginas de la sección [Trabajar con URLs](#).
- Un servlet puede llamar directamente a los métodos públicos de otros servlet, si los dos se están ejecutando dentro del mismo servidor.

Esta página explica la segunda opción. Para llamar directamente a los métodos públicos de otro servlet, debemos:

- Conocer el nombre [del servlet](#) al que queremos llamar.
- Obtener el acceso al objeto Servlet del servlet.
- Llamar al método público del servlet.

Para obtener el acceso al objeto Servlet, utilizamos el método `getServlet` del objeto `ServletContext`. Obtener el objeto `ServletContext` desde el objeto `ServletConfig` almacenado en el objeto Servlet. Un ejemplo aclarará esto. Cuando el servlet `BookDetail` llama al servlet `BookDB`, el servlet `BookDetail` obtiene el objeto Servlet del `BookDB Servlet` de esta forma:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        BookDBServlet database = (BookDBServlet)
            getServletConfig().getServletContext().getServlet("bookdb");
        ...
    }
}
```

Una vez que tenemos el objeto Servlet, podemos llamar a cualquiera de los métodos públicos del servlet. Por ejemplo, el servlet `BookDetail` llama al método `getBookDetails` del servlet `BookDB`:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        BookDBServlet database = (BookDBServlet)
            getServletConfig().getServletContext().getServlet("bookdb");
        BookDetails bd = database.getBookDetails(bookId);
        ...
    }
}
```

```
}  
}
```

Debemos tener precaución cuando llamemos a métodos de otro servlet. Si el servlet al que queremos llamar implementa el interface [SingleThreadModel](#), nuestra llamada podría violar la naturaleza mono-thread del servlet. (El servidor no tiene forma de intervenir y asegurarse de que nuestra llamada suceda cuando el servlet no está interactuando con otro cliente). En este caso, nuestro servlet debería hacer una petición HTTP al otro servlet en vez de llamar directamente a sus métodos.

Mercados Globales

Los mercados globales son lucrativos. Vender tus aplicaciones en todo el mundo genera muchas rentas e incrementa la cuota de mercado. Para tener éxito en el mercado internacional tus aplicaciones deben soportar idiomas locales y convenciones para el formateo de datos. Tus clientes no esperarán menos de tí. Por ejemplo, los Norteamericanos esperan sus mensajes en Inglés, las fechas en el formato mm/dd/yy, y la moneda expresada en dólares (\$). Los Alemanes querrán sus mensajes en Alemán, las fechas en el formato dd.mm.yyyy, y la moneda expresada en Marcos alemanes (DM).

Las compañías de software utilizaban la forma más dura para personalizar sus programas. En la primera versión de un producto, codificaban dentro del programa los datos dependientes de la cultura como los textos o las fechas. Cuando querían soportar un nuevo idioma, tenían que hacer una copia del código fuente, cambiar todos los elementos necesarios y recompilar. Para liberar una nueva versión, tenían que actualizar los diferentes ficheros de código fuente y luego lanzar un nuevo conjunto de ejecutables. Este proceso no sólo era tedioso, también era propenso a errores y costoso. Afortunadamente, han emergido nuevas técnicas para ayudar a desarrollar software global: la Internacionalización y la Localización.

Internationalización

Esta sección describe las ventajas y propiedades de la Internacionalización. Cuando se internacionaliza un programa se hace más fácil el poder portarlo entre idiomas y regiones.

Localización

Después de que un programa ha sido internacionalizado, puede ser adaptado, o localizado, para un idioma o región específica. Esta sección describe brevemente el proceso de localización.

Datos Sensibles a la Cultura

Los datos sensibles a la cultura son la razón que tenemos para internacionalizar y localizar nuestros programas. Esta sección describe los diferentes tipos de datos sensibles a la cultura que nos podríamos encontrar.

Internationalización

Internationalización es el proceso de diseñar una aplicación para que pueda ser adaptada a diferentes idiomas y regiones, sin necesidad de cambios de ingeniería. Algunas veces el término internacionalización se abrevia i18n, porque en el idioma inglés hay 18 letras entre la "i" y la "n" de Internacionalización.

Un programa internacionalizado tiene las siguientes características:

- Con la adición de datos de localización, el mismo ejecutable puede ser ejecutado en cualquier lugar del mundo.
- El texto mostrado por el programa está en el idioma nativo del usuario final.
- Los elementos textuales como mensajes de estado y etiquetas de elementos GUI no están codificadas dentro del programa. Son almacenados fuera del código fuente y recuperados de forma dinámica.
- El soporte de nuevos idiomas no requiere re-compilación.
- Otros datos dependientes de la cultura, como fechas y monedas, aparecen en el formato e idioma de la región del usuario final.
- Puede ser localizado rápidamente.

Si has internacionalizado tu producto, ya está listo para la [localización](#).

Localización

Localización es el proceso de adaptar software para una región o idioma específico añadiendo componentes específicos de la localidad y traduciendo el texto. El término Localización normalmente se contrae como "l10n" porque en idioma inglés hay 10 letras entre la "L" y la "n".

La traducción del texto es una importante tarea de localización. Durante la internacionalización, los textos como las etiquetas de los componentes GUI y los mensajes de error son almacenados fuera del código fuente para ser recuperados en tiempo de ejecución. Antes de que el texto pueda recuperarse debe ser traducido. Como el texto no está dentro del código fuente, el programa no requiere ninguna modificación. Los traductores trabajan con ficheros de texto que son leídos por el programa, no están dentro de él. Así, el mismo ejecutable funciona en cualquier parte del mundo.

Las convenciones de formateo de fechas, números y monedas varían con el idioma y la región. Los localizadores podrían necesitar especificar algunos patrones de formateo. O, el programa podría proporcionarlos automáticamente. En cualquier caso, los localizadores deben probar el software para verificar que las convenciones de formateo están según los requerimientos locales.

Otros tipos de datos, como sonidos e imágenes también requieren localización [sensible a la cultura](#).

Datos Sensibles a la Cultura

Texto

Aunque las aplicaciones multi-media existen desde hace años, casi todos los programas primarios utilizan texto para comunicarse con el usuario final. Esto es especialmente cierto en la comunidad de negocios. Si estás leyendo esto on-line, mira la cantidad de texto mostrado por las otras aplicaciones que se están ejecutando en tu sistema. ¿Has visto todos esos botones y menús? Todos necesitan ser traducidos si las aplicaciones van a ser vendidas en todo el mundo. No solo esto, también los mensajes de estado, las informes impresos y las pantallas de ayuda en línea requieren traducción. Como las aplicaciones muestran y generan tanto texto, las cuentas de traducción forman la mayor parte del costo de la localización.

Si manejas adecuadamente los elementos textuales de tu programa, podrás reducir el coste de la traducción. Deberías mover el texto traducible a ficheros de propiedades, donde puedan ser cargados en objetos ResourceBundle. Aprenderás como hacer esto en la sección: [Utilizar ficheros de Propiedades](#).

Números

Las convenciones de formateo de números varía con el país. Se podrían utilizar diferentes caracteres para marcar el punto decimal y para separar los millares. La siguiente tabla muestra sólo unas pocas formas diferentes de formatear un número particular:

País	Número Formateado
Francia	123 456,78
Alemania	123.456,78
U.S.A.	123,456.78

La lección [Formateo de Números y Moneda](#) te muestra como crear formatos de números específicos de la localidad.

Moneda

Las unidades de moneda varían con el país. y también el formato de la cantidad. La siguiente tabla ilustra algunos ejemplos:

País	Moneda	Ejemplo
España	Peseta	1.234,56Pts
Italia	Lira	L. 1.234,56
U.S.A.	Dollar	\$1,234.56

Puedes ver [Formateo de Números y Moneda](#) para más información.

Fechas y Horas

El formato de fechas y horas varía con el país. La siguiente tabla muestra algunos ejemplos:

País	Fecha	Hora
Canada	30/4/98	20:15
Alemania	30.4.1998	20:15 Uhr
U.S.A.	4/30/98	10:15 PM

[Formateo de Fechas y Horas](#) explica cómo realizar un formateo de acuerdo a varias convenciones culturales.

Imágenes

Con el advenimiento de las aplicaciones GUI, las imágenes aparecen en cualquier lugar de nuestras pantallas. Las encontramos en iconos, gráficos, fotografías, dibujos y banners. Aunque la utilización de imágenes es universal, su significado no lo es. Por ejemplo, podrías estar tentado a utilizar señales de tráfico para ayudar a los usuarios finales a navegar a través de la aplicación. Pero como las señales de tráfico varían de un país a otro, tu programa debería mostrar diferentes versiones de los iconos en diferentes países. Afortunadamente, puedes manejar objetos Image para las diferentes regiones y aislarlos en un ListResourceBundle. Este proceso se describe en la lección [Utilizar un ListResourceBundle](#).

Colores

Los colores tienen diferentes significados a lo largo del mundo. En U.S.A. el color blanco significa pureza, pero en Japón significa muerte. En Egipto, el color rojo representa la muerte, pero en China sugiere felicidad. Al igual que los objetos Image, los objetos Color dependientes de la cultura pueden ser manejados si son almacenados en un ListResourceBundle.

Sonidos

Si tus aplicaciones generan sonidos, debes tener en cuenta que el mismo sonido podría no ser reconocido en todo el mundo. Por ejemplo, las sirenas de policía son diferentes en U.S.A y en Alemania. Por su puesto, si tu aplicación da instrucciones verbales, las instrucciones deben ser traducidas. Puedes seguir la pista de de objetos AudioClip dependientes de la cultura si los almacenas en objetos ListResourceBundle.

Un ejemplo Rápido

Si eres nuevo internacionalizando software, esta lección es para tí. Utilizando un ejemplo sencillo, veremos como internacionalizar un programa que muestra mensajes de texto en el idioma apropiado. En esta lección aprenderás cómo trabajan juntos los objetos Locale y ResourceBundle, y cómo utilizar los ficheros de propiedades.

[Antes de la Internacionalización](#)

En la primera versión del código fuente, codificamos las versiones inglesas de los mensajes que queríamos mostrar. Esta NO es la forma de escribir software internacionalizado.

[Después de la Internacionalización](#)

Una pequeña visión de lo que será nuestro código fuente después de la internacionalización.

[Ejecutar el programa de Ejemplo](#)

Para ejecutar el programa de ejemplo, se especifica el idioma y el país en la línea de comandos. Esta sección muestra varios ejemplos.

[Cómo hemos internacionalizado el programa de Ejemplo](#)

Internacionalizar el programa de ejemplo sólo requiere unos pocos pasos. Te sorprenderás de lo fácil que es.

Antes de la Internacionalización

Supongamos que hemos escrito un programa que muestra tres mensajes:

```
System.out.println("Hello.");  
System.out.println("How are we?");  
System.out.println("Goodbye.");
```

Hemos decidido que este programa necesita mostrar estos tres mensajes para la gente que vive en Francia y en Alemania. Desafortunadamente tu personal de programación no es multi-lingüe., por eso necesitas ayuda para traducir los mensajes al Francés y al Alemán. Cómo los traductores no son programadores, tenemos que sacar los mensajes fuera del código fuente a ficheros de texto que puedan ser editados por los traductores. También queremos que el programa sea lo suficientemente flexible para poder mostrar los mensajes en otros idiomas, pero ahora mismo no sabemos qué idiomas. Por lo tanto, queremos que el usuario final especifique su idioma en el momento de la ejecución.

Parece que este programa necesita ser internacionalizado.

Ozito

Después de la Internacionalización

Abajo puedes ver el código fuente del programa internacionalizado. Observa que el texto de los mensajes no está codificado.

```
import java.util.*;

public class I18NSample {

    static public void main(String[] args) {

        if (args.length != 2) {
            System.out.println("Please specify language and country codes.");
            System.out.println("For example: java I18NSample fr FR");
            System.exit(-1);
        }

        Locale currentLocale;
        ResourceBundle messages;
        String language = new String(args[0]);
        String country = new String(args[1]);

        currentLocale = new Locale(language, country);

        messages =
            ResourceBundle.getBundle("MessagesBundle", currentLocale);

        System.out.println(messages.getString("greetings"));
        System.out.println(messages.getString("inquiry"));
        System.out.println(messages.getString("farewell"));
    }
}
```

Ejecutar el Programa de Ejemplo

Nuestro programa internacionalizado es flexible, porque permite que el usuario final pueda especificar el idioma y el país en la línea de comandos. En el siguiente ejemplo, el programa muestra los mensajes en Francés, porque el código de lenguaje es fr (Francés), y el código de país es FR (Francia):

```
% java I18NSample fr FR
Bonjour.
Comment allez-vous?
Au revoir.
```

En el siguiente ejemplo, el código de idioma es en (Inglés) y el código de país es US (Estados Unidos):

```
% java I18NSample en US
Hello.
How are you?
Goodbye.
```

¿Cómo hemos internacionalizado el programa de ejemplo?

Si echaste un vistazo al código fuente internacionalizado, observarías que los mensajes en inglés dentro del código han desaparecido. Como estos mensajes ya no están dentro del código, y como el código del idioma se especifica en el momento de la ejecución, el mismo ejecutable puede ser distribuido por todo el mundo. No se requieren recompilaciones para su localización, Nuestro programa ha sido internacionalizado.

Te podrías preguntar que ha sucedido con el texto de los mensajes, o que entendemos por los códigos de idioma y de país. No te preocupes. Te explicaremos todos estos conceptos cuando pasemos a través del proceso de internacionalización del programa de ejemplo:

[Crear el Fichero de Propiedades](#)

[Definir la Localidad](#)

[Crear un objeto ResourceBundle](#)

[Recuperar el texto del ResourceBundle](#)

Crear el Fichero de Propiedades

Un fichero de propiedades almacena información sobre las características de un programa o un entorno. Estos ficheros de propiedades tienen formato de texto plano. Se pueden crear con cualquier editor de textos.

En nuestro ejemplo, el fichero de propiedades almacena los textos traducibles de los mensajes que deseamos mostrar. Antes de que nuestro programa fuera internacionalizado, las versiones inglesas de los mensajes fueron codificadas dentro de sentencias `System.out.println`. Nuestro fichero de propiedades por defecto, que se llama `MessagesBundle.properties`, contiene las siguientes líneas:

```
greetings = Hello
farewell = Goodbye
inquiry = How are you?
```

Ahora que los mensajes están en un fichero de propiedades, podemos traducirlos a varios idiomas. No se necesita modificar el código fuente. Nuestro traductor de Francés, ha creado un fichero de propiedades llamado `MessagesBundle_fr_FR.properties`, que contiene estas líneas:

```
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

Observa que los valores situados a la derecha de los signos igual "=" han sido traducidos, pero las claves del lado izquierdo del signo no han cambiado. Estas claves no deben cambiar, porque el programa se referirá a ellas cuando recupere el texto traducido.

El nombre del fichero de propiedades es importante. El nombre del fichero `MessagesBundle_fr_FR.properties` contiene el código del idioma `fr` y el código del país `FR`. Estos códigos también se utilizan cuando se crea un objeto `Locale`.

Definir la Localidad

Un objeto `Locale` define un idioma particular y un país. La siguiente sentencia define un objeto `Locale` para el idioma Inglés y el país de Estados Unidos.

```
aLocale = new Locale("en","US");
```

El siguiente ejemplo crea objetos `Locale` para el idioma francés en Canada y Francia:

```
caLocale = new Locale("fr","CA");  
frLocale = new Locale("fr","FR");
```

Queremos mantener flexible el programa de ejemplo, por ello, en vez de codificar los códigos de idioma y de país en el código fuente, los obtenemos de la línea de comandos en el momento de la ejecución:

```
String language = new String(args[0]);  
String country = new String(args[1]);  
currentLocale = new Locale(language, country);
```

Los objetos `Locale` son sólo identificadores. Después de definir un objeto `Locale`, se lo pasas a otro objeto que realiza las tareas útiles, como formatear fechas y números. Estos objetos se llaman sensibles a la localidad, porque su comportamiento varía de acuerdo a la localidad. Un `ResourceBundle` es un ejemplo de objeto sensible a la localidad.

Crear el objeto ResourceBundle

Los objetos ResourceBundle contienen objetos específicos de una localidad. Estos objetos ResourceBundle se utilizan para aislar los datos sensibles a la localidad, como texto traducible, etc. En nuestro programa de ejemplo, el ResourceBundle está constituido por los ficheros de propiedades que contienen los mensajes que queremos mostrar.

Nuestro objeto ResourceBundle se crea de esta forma:

```
message = ResourceBundle.getBundle("MessagesBundle",currentLocale)
```

Los argumentos pasados al método getBundle identifican a los ficheros de propiedades a los que queremos acceder. El primer argumento, MessagesBundle, se refiere a esta familia de ficheros de propiedades:

```
MessagesBundle_en_US.properties  
MessagesBundle_fr_FR.properties  
MessagesBundle_de_DE.properties
```

El objeto Locale, que es el segundo argumento de getBundle, especifica el fichero elegido de MessagesBundle. Cuando se creó el objeto Locale, se le pasaron al constructor los códigos del idioma y del país. Observa que estos códigos forman parte del nombre de los ficheros de propiedades de MessagesBundle.

Ahora, todo lo que tenemos que hacer es traducir los mensajes de ResourceBundle.

Buscar el Texto dentro de ResourceBundle

Los ficheros de propiedades contienen parejas clave/valor. Los valores consisten en texto traducido que nuestro programa mostrará. Especificaremos las claves cuando querramos utilizar los mensajes traducidos del ResourceBundle con el método `getString`. Por ejemplo, para recuperar el mensaje identificado por la clave "greetings", llamaríamos a `getString` de esta forma:

```
String msg1 = messages.getString("greetings");
```

En nuestro ejemplo, utilizamos la clave `greetings` porque refleja el contenido del mensaje, podrías haber utilizado cualquier otro String, como `s1` o `msg1`. Sólo recordar que la clave debe estar escrita en tu programa y que debe estar presente en los ficheros de propiedades. Si tu traductor modifica accidentalmente las claves de los ficheros de propiedades, `getString` no podrá encontrar los mensajes.

Esto es todo. Como puedes ver, internacionalizar un programa no es demasiado difícil. Sólo requiere algo de planificación, y un poco de código extra, pero los beneficios son enormes. El ejemplo que hemos cubierto en esta sección ha sido intencionadamente sencillo porque queríamos proporcionarte una introducción al proceso de internacionalización. El API de Java ofrece muchas más capacidades de internacionalización que las descritas en esta sección. Explicaremos esos tópicos en mayor detalle en las siguientes secciones.

Seleccionar la Localidad

Un programa internacionalizado muestra información diferente a lo ancho del mundo. Por ejemplo, el programa mostrará diferentes mensajes en París, Tokio o Nueva York. Si el proceso de localización está bien ajustado, el programa mostrará diferentes mensajes en Nueva York y en Londres, teniendo en cuenta las diferencias entre el Inglés americano y el británico. ¿Cómo puede un programa internacionalizado identificar el idioma y la región de sus usuarios finales? Es sencillo, refiriéndose a un objeto `Locale`.

Un objeto `Locale` es un identificador para una combinación particular de idioma, región y cultura. Si una clase varía su comportamiento de acuerdo con un objeto `Locale`, se dice que es sensible a la localidad. Por ejemplo, la clase `NumberFormat` es sensible a la localidad porque el formateo de números depende de la localidad. `NumberFormat` podría devolver un número como 902 300 (Francia), o 902.300 (Alemania), o 902,300 (U.S.). Los objetos `Locale` son sólo identificadores. El trabajo real, como el formateo o la detección de límites de palabras lo realizan los métodos de las clases sensibles a la localidad.

En esta lección aprenderás cómo trabajar con objetos `Locale`, y cómo realizar las siguientes tareas:

Crear un Objeto Locale

Cuando se crea un objeto `Locale`, se debe especificar un código de idioma y un código de país. Existe un tercer parámetro opcional, la variante.

Identificar Localidades Disponibles

Las clases sensibles a la localidad sólo soportan ciertas definiciones de localidades. Esta sección muestra cómo determinar que definiciones de localidades están soportadas.

La Localidad por Defecto

Si no se asigna explícitamente un objeto `Locale` a un objeto sensible a la localidad, se utilizará la localidad por defecto. Afortunadamente, se puede seleccionar la `Locale` por defecto.

El ámbito de una Localidad

En la plataforma Java, no se especifica un objeto `Locale` global seleccionando una variable de entorno antes de ejecutar la aplicación. En su lugar, se asigna un objeto `Locale` a cada objeto sensible a la localidad.

Crear un objeto Locale

Para crear un objeto Locale, se especifica el código del idioma y el código del país. Por ejemplo para especificar el idioma Francés y el país Canada, se llamará al constructor de esta forma:

```
aLocale = new Locale("fr", "CA");
```

En el siguiente ejemplo, creamos objetos Locale para el idioma Inglés en U.S.A. y Gran Bretaña:

```
bLocale = new Locale("en", "US");  
cLocale = new Locale("en", "GB");
```

El primer argumento es el código del lenguaje, un par de letras minúsculas conformes a la norma ISO-639. Puedes encontrar una lista completa de códigos ISO-639 en: <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>

El segundo argumento es el código de país. Que consiste en dos letras mayúsculas, conforme a la norma ISO3166. Puedes encontrar una copia de esta norma en: http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

Además, si necesitas distinguir todavía más tu Locale, puedes especificar un tercer parámetro, llamado código de variante. Si existen variaciones en el idioma utilizado dentro del mismo país, podrías querer especificar una variante. Por ejemplo, en el Sur de Estados Unidos, la gente suele decir "y'all," cuando en el Norte dicen "you all." Se podría crear diferentes objetos Locale de esta forma:

```
nLocale = new Locale("en", "US", "NORTH");  
sLocale = new Locale("en", "US", "SOUTH");
```

Los códigos de variante no conforman un estándar. Son arbitrarios y específicos de la aplicación. Si se crean objetos Locale con los códigos de variante NORTH y SOUTH, como en el ejemplo anterior, sólo nuestra aplicación sabrá como tratarlos.

Normalmente, se especifican códigos de variante para identificar diferencias cuasadas por la plataforma de ordenador. Por ejemplo, las diferencias de fuentes podría forzar a utilizar caracteres diferentes en Windows y en UNIX. Se podría definir los objetos Locale con estos códigos de variante:

```
xLocale = new Locale("de", "DE", "UNIX");  
yLocale = new Locale("de", "DE", "WINDOWS");
```

El código de país y el código de variante son opcionales. Se podría crear un objeto Locale para el idioma Inglés de esta forma:

```
enLocale = new Locale("en", "");
```

Sin embargo, si se omite el código de país, la aplicación no podrá adaptarse a las diferencias regionales del idioma. Por ejemplo, un programa que utilice el objeto `enLocale` no podrá mostrar la palabra "colour" en U.K. y la palabra "color" en U.S.A.

Por conveniencia, la clase `Locale` proporciona constantes para algunos idiomas y países. Por ejemplo, se puede crear un objeto `Locale` especificando las constantes `JAPANESE` o `JAPAN`. Los objetos creados por las dos sentencias siguientes son equivalentes:

```
j1Locale = Locale.JAPAN;  
j2Locale = new Locale("ja", "JA");
```

Cuando se especifica una constante de idioma, la porción del país del objeto `Locale` no se define. Las siguientes sentencias crean objetos `Locale` equivalentes:

```
j3Locale = Locale.JAPANESE;  
j4Locale = new Locale("ja", "");
```

Identificar las Localidades Disponibles

Se puede crear un objeto `Locale` con cualquier combinación de códigos válidos de idioma y de país, pero eso no significa que se pueda utilizar. Recuerda, un objeto `Locale` es sólo un identificador. El objeto `Locale` se pasa a otros objetos que realizan el trabajo verdadero. Estos otros objetos, que llamamos sensibles a la localidad, no saben como tratar todas las posibles definiciones de `Locale`.

Para encontrar los tipos de definiciones de `Locale` que reconoce una clase sensible a la localidad, se llama al método `getAvailableLocales`. Por ejemplo, para encontrar las definiciones de localidades soportadas por la clase `DateFormat`, se podría escribir una rutina como ésta:

```
import java.util.*;
import java.text.*;

public class Available {

    static public void main(String[] args) {

        Locale list[] = DateFormat.getAvailableLocales();

        for (int i = 0; i < list.length; i++) {
            System.out.println
                (list[i].getLanguage() + " " + list[i].getCountry());
        }
    }
}
```

La Localidad por Defecto

Si no se asigna un objeto `Locale` a un objeto sensible a la localidad, depende del objeto `Locale` devuelto por el método `getDefault`. Se puede seleccionar el objeto `Locale` por defecto de dos formas:

- Seleccionar las propiedades del sistema `user.language` y `user.region`. La clase `Locale` selecciona el valor por defecto recuperando los valores de estas propiedades.
- Llamando al método `setDefault`.

El siguiente ejemplo muestra estas dos técnicas de seleccionar el objeto `Locale` por defecto:

```
import java.util.*;

public class DefaultLocale {

    static public void main(String[] args) {

        Properties props = System.getProperties();
        props.put("user.language", "ja");
        props.put("user.region", "JA");
        System.setProperties(props);

        Locale aLocale = Locale.getDefault();
        System.out.println(aLocale.toString());

        aLocale = new Locale("fr", "FR");
        Locale.setDefault(aLocale);
        System.out.println(aLocale.toString());
    }
}
```

Aquí está la salida de este programa:

```
ja_JA
fr_FR
```

No dependas de la localidad por defecto a menos que la selecciones antes con uno de los dos métodos mostrados arriba. Si no lo haces podrías encontrarte que la localidad por defecto devuelta por `getDefault` podría no ser la misma en todas las plataformas Java.

El Ambito de la Localidad

No existe algo parecido a una Localidad global en el lenguaje de programación Java. Se puede especificar una localidad por defecto, como se describió en la página anterior, pero no es necesario que se utilice la misma localidad a lo largo de todo el programa. Si se desea, se puede asignar un objeto Locale diferente para cada objeto sensible a la localidad de nuestro programa. Este es el caso cuando se escriben aplicaciones multi-idioma, que pueden mostrar información en distintos idiomas. Pero para la mayoría de las aplicaciones se selecciona el mismo objeto Locale para todos los objetos sensibles a la localidad.

La programación distribuida alcanza algunas cotas interesantes. Por ejemplo, supongamos que hemos diseñado una aplicación servidor que recibe peticiones de clientes de distintos países. Si el objeto Locale de cada cliente es diferente, ¿cual debería ser el objeto Locale del Servidor? Quizás el servidor sea multi-hilo, y cada thread seleccione su objeto Locale para los servicios de su cliente. O quizás todos los datos pasados entre el servidor y los clientes deberían ser independientes de la localidad.

¿Qué diseño deberíamos utilizar? La respuesta depende de los requerimientos específicos de la aplicación, si están involucrados sistemas legales, y de lo complejo que se quiera el sistema. Teóricamente, se podrían seleccionar todos los objetos sensibles a la localidad tanto en el cliente como en el servidor con un objeto Locale diferente. Por supuesto, esto no podría ser en la práctica, pero demuestra la flexibilidad de los objetos Locale en el lenguaje de programación Java.

Aislar los objetos específicos de la localidad en un ResourceBundle

Los datos específicos de la localidad deben ser creados de acuerdo a las convenciones del idioma y la región del usuario final. El texto mostrado por un interface de usuario es el ejemplo más obvio de datos específicos de la localidad. Por ejemplo, una aplicación con un botón "Cancel" en los Estados Unidos, tendrá un botón "Abbrechen" en Alemania. En otros países este botón tendrá otras etiquetas. Obviamente, tu no quieres codificar la etiqueta de este botón. ¿No sería bonito poder obtener automáticamente la etiqueta correcta para una Localidad dada? Afortunadamente, se puede, asilando los objetos específicos de la localidad en un ResourceBundle.

En esta lección, aprenderás cómo crear, cargar y acceder a objetos ResourceBundle. Si tienes prisa por examinar algún código fuente, adelantate y chequea las dos últimas lecciones. Luego podrás volver a las dos primeras para obtener alguna información conceptual sobre los objetos ResourceBundle.

La clase ResourceBundle

Los objetos ResourceBundle contienen objetos específicos de la localidad. Cuando se necesita uno de estos objetos, se busca en el ResourceBundle, que devuelve el objeto que corresponde con la Localidad del usuario final. En esta sección, explicaremos como un ResourceBundle se relaciona con una Locale. También describiremos las subclases de ResourceBundle, y cuando se pueden utilizar.

Preparar el uso de un ResourceBundle

Antes de crear y cargar objetos ResourceBundle, se debería planificar un poco. Primero, identificar los objetos específicos de la localidad de nuestro programa. Luego, organizar estos objetos en categorías y almacenarlos en diferentes objetos ResourceBundle.

Utilizar Ficheros de Propiedades

Si nuestra aplicación contiene objetos String que necesitan ser traducidos a diferentes idiomas, deberíamos almacenar estos objetos String en un PropertyResourceBundle, que está constituido por un conjunto de ficheros de propiedades. Como los ficheros de propiedades son sencillos ficheros de texto, pueden ser creados y mantenidos por tus traductores. No necesitas cambiar el código fuente. En esta sección aprenderás cómo seleccionar los ficheros de propiedades que constituyen un PropertyResourceBundle.

Utilizar un ListResourceBundle

La clase `ListResourceBundle`, que es una subclase de `ResourceBundle`, maneja objetos específicos de la localidad con una lista. Un `ListResourceBundle` está constituido por un fichero de clase, lo que significa que se deberá codificar y compilar un nuevo fichero fuente cada vez que querramos soportar una nueva localidad. Por lo tanto, no se debería utilizar un `ListResourceBundle` para aislar objetos `String` que deban ser traducidos a otros idiomas. Para aislar texto traducible, se debería utilizar un `PropertyResourceBundle` porque está constituido por un conjunto de ficheros de propiedades editables. Sin embargo, los objetos `ListResourceBundle` son útiles, porque al contrario que los ficheros de propiedades, pueden almacenar cualquier tipo de objeto específico de la localidad. Pasando a través de un programa de ejemplo, en esta sección veremos cómo utilizar un `ListResourceBundle`.

La clase ResourceBundle

Cómo se relaciona un ResourceBundle con una Locale

Conceptualmente hablando, un ResourceBundle es un conjunto de subclases relacionadas que comparten el mismo nombre base. La siguiente lista muestra un conjunto de subclases relacionadas. El nombre base es ButtonLabel. Los caracteres que siguen al nombre base son el código del idioma, el código del país y el código de la variante de un Locale. Por ejemplo, ButtonLabel_en_GB corresponde con la localidad especificada por el código del idioma Inglés, (en) y el código del país para Gran Bretaña (GB).

```
ButtonLabel
ButtonLabel_de
ButtonLabel_en_GB
ButtonLabel_fr_CA_UNIX
```

Para seleccionar el ResourceBundle apropiado, se llama al método `getBundle`. Los siguientes ejemplo seleccionan el ButtonLabel ResourceBundle para localidad que corresponden con el idioma Francés , el país Canada y la plataforma UNIX:

```
Locale currentLocale = new Locale("fr", "FR", "UNIX");
```

```
ResourceBundle introLabels =
    ResourceBundle.getBundle("ButtonLabel", currentLocale);
```

Si no existe una clase ResourceBundle para la localidad especificada. `getBundle` trata de encontrar la correspondencia más cercana. Por ejemplo, si no existiera una clase para ButtonLabel_fr_CA_UNIX, `getBundle` buscará las clases en el siguiente orden:

```
ButtonLabel_fr_CA_UNIX
ButtonLabel_fr_CA
ButtonLabel_fr
ButtonLabel
```

Si `getBundle` no encuentra una correspondencia en la lista anterior, intentará una búsqueda similar utilizando la localidad por defecto. Si vuelve a fallar, `getBundle` lanzará una `MissingResourceException`.

Siempre se debe proporcionar una clase base sin sufijos. En el ejemplo anterior, si existiera una clase llamada ButtonLabel, `getBundle` no lanzaría `MissingResourceException`.

Las subclases ListResourceBundle y PropertyResourceBundle

La clase abstracta `ResourceBundle` tiene dos subclases: `ListResourceBundle` y `PropertyResourceBundle`. La subclase elegida depende de cómo estén localizados los datos.

Un `PropertyResourceBundle` está constituido por uno o más ficheros de propiedades. Se deberían almacenar objetos `String` traducibles en ficheros de propiedades. Como los ficheros de propiedades son sólo ficheros de texto y no forman parte del código fuente Java, pueden ser creados y actualizados por los traductores. No se requiere experiencia en programación. Un traductor puede añadir soporte para una localidad adicional con sólo crear un nuevo fichero de propiedades. No se necesita un nuevo fichero de clase. Los ficheros de propiedades sólo pueden contener valores para objetos `String`. Si necesitamos otros tipos de objetos, se debe utilizar un `ListResourceBundle`. [Construir un ResourceBundle con Ficheros de Propiedades](#) te muestra como utilizar `PropertyResourceBundle`.

La clase `ListResourceBundle` maneja recursos con una lista. Cada `ListResourceBundle` está constituido por un fichero de clase. Se puede almacenar cualquier objeto específico de la localidad en un `ListResourceBundle`. Para añadir soporte para nuevas localidades, se debe crear otro fichero fuente, y compilarlo. Como los traductores normalmente no son programadores, no se deberían almacenar objetos `String` que requieran traducción en un `ListResourceBundle`. [Utilizar un ListResourceBundle](#) contiene código de ejemplo que puede resultar útil.

La clase `ResourceBundle` es flexible. Si primero decidimos cargar nuestros objetos `Strings` específicos de la localidad en un `ListResourceBundle`, y luego decidimos utilizar un `PropertyResourceBundle`, el impacto en nuestro código será limitado. Por ejemplo, la siguiente llamada a `getBundle` recuperará un `ResourceBundle` para la localidad apropiada, tanto si `ButtonLabel` está constituido por una clase o por un fichero de propiedades:

```
ResourceBundle introLabels =  
    ResourceBundle.getBundle("ButtonLabel", currentLocale);
```

Parejas Clave-Valor

Los objetos `ResourceBundle` contienen un array de parejas clave-valor. La clave, que debe ser un `String`, es lo que se especificará cuando querramos recuperar el valor desde el `ResourceBundle`. El valor es el objeto específico de la localidad. En el siguiente ejemplo, las claves son los objetos `String` "OkKey" y "CancelKey":

```
class ButtonLabel_en extends ListResourceBundle {  
    // English version  
    public Object[][] getContents() {  
        return contents;  
    }  
}
```

```
static final Object[][] contents = {  
    {"OkKey", "OK"},  
    {"CancelKey", "Cancel"},  
};  
}
```

Para recuperar el String "OK" desde el ResourceBundle, deberíamos especificar la clave apropiada cuando llamemos a getString:

```
String okLabel = ButtonLabel.getString("OkKey");
```

El ejemplo precedente es muy simple, porque los valores String están codificados en el código fuente. Esto no es una buena práctica, porque tus traductores necesitan trabajar con ficheros de propiedades que están separados del código fuente.

Un fichero de propiedades contiene parejas de clave-valor. La clave está en el lado izquierdo del signo igual y el valor en el lado derecho. Cada pareja está en una línea independiente. Las claves sólo pueden estar representadas por objetos String. El siguiente ejemplo muestra el contenido de un fichero de propiedades llamado ButtonLabel.properties:

```
OkKey = OK  
CancelKey = Cancel
```

Preparar para Utilizar un ResourceBundle

Identificar los Objetos Específicos de la Localidad

Si la aplicación tiene un interface de usuario, contendrá muchos objetos específicos de la localidad. Deberíamos empezar leyendo el código y buscándo objetos que varíen con la localidad. La lista podría incluir objetos ejemplarizados de las siguientes clases:

- String
- Component
- Graphics
- Image
- Color
- AudioClip

Observarás que esta lista no contiene objetos que representen números, fechas, horas o monedas. El formato de esos objetos varía con la localidad pero no así los propios objetos. Por ejemplo, se formatea un objeto Date de acuerdo a la localidad, pero se sigue utilizando el mismo objeto Date sin importar la localidad. En vez de aislar estos objetos en un ResourceBundle, se deben formatear con clases especiales de formateo sensible a la localidad. Veremos como hacer esto en [Formateo de Fechas y Horas](#).

En general, los objetos almacenados en un ResourceBundle están predefinidos y se venden con el producto. Estos objetos no se modifican mientras el programa se está ejecutando. Por ejemplo, se debería almacenar la etiqueta de un Menu en un ResourceBundle porque es específico de la localidad, y no cambia durante la sesión del programa. Sin embargo, no se deberían aislar en un ResourceBundle los objetos String introducidos por el usuario final en un TextField. Los datos de este estilo podrían variar de día a día. Son específicos de la sesión del programa, no de la localidad en que se está ejecutando el programa.

Normalmente, la mayoría de los objetos que se necesita aislar en un ResourceBundle son objetos String. Sin embargo, no todos los objetos String son específicos de la Localidad. Por ejemplo, si un String es un elemento de protocolo utilizado en un proceso de inter-comunicación, no necesita ser localizado porque el usuario final nunca lo verá. La decisión de cuando localizar algunos objetos String no siempre está clara. Los ficheros LOG son un buen ejemplo. Si un fichero Log es escrito por un programa y leído por otro, ambos programas están utilizando el fichero Log como un buffer de comunicación. Supongamos que el usuario final chequea ocasionalmente el contenido de este fichero. ¿Debería estar localizado este fichero Log? Por otro lado, si el fichero es raramente chequeado por los usuarios finales, el coste de la traducción podría no merecer la pena. La decisión de localizar este fichero Log depende de un número de factores: diseño del programa, facilidad de utilización, coste de la traducción y soportabilidad.

Organizar Objetos ResourceBundle

Puedes organizar tus objetos ResourceBundle cargando cada uno de ellos en una categoría diferente de objetos. Por ejemplo, podríamos querer cargar todos las etiquetas de Button dentro de un ResourceBundle llamada ButtonLabelsBundle. Cargar los objetos relacionados en diferentes objetos ResourceBundle tiene varias ventajas:

- El código es fácil de leer y de mantener.
- Recuperar un objeto del ResourceBundle es más rápido si este contiene un pequeño número de objetos.
- Los Traductores encontrarán más sencillo el trabajar con ficheros de propiedades más pequeños.

La mayoría de los datos específicos de la localidad consisten en objetos String. Si estos objetos necesitan ser traducidos, se almacenan en un objeto ResourceBundle que está constituido por ficheros de propiedades. Si se utilizan ficheros de propiedades, los traductores pueden añadir soporte para idiomas adicionales creando nuevos ficheros de propiedades.

Construir un ResourceBundle con Ficheros de Propiedades

Esta sección pasea a través de un programa de ejemplo llamado PropertiesDemo. El código fuente del programa está en [PropertiesDemo.java](#). También podrías encontrar útil examinar la [salida](#) generada por este programa.

1. Crear el Fichero de Propiedades por defecto

Un fichero de propiedades es un sencillo fichero de texto. Los ficheros de propiedades se pueden crear y mantener con un sencillo editor de texto.

Siempre se debe crear un fichero de propiedades por defecto. El nombre de este fichero empieza con el nombre base del ResourceBundle y termina con el sufijo .properties. En el programa PropertiesDemo, el nombre base es LabelsBundle. Por lo tanto, el fichero de propiedades por defecto se llama LabelsBundle.properties. Este fichero contiene las siguientes líneas:

```
# This is the default LabelsBundle.properties file
s1 = computer
s2 = disk
s3 = monitor
s4 = keyboard
```

En este fichero se puede observar que las líneas de comentarios empiezan con una almohadilla (#). Las otras líneas contienen parejas de clave-valor. Las claves están en el lado izquierdo del signo igual y los valores en el lado derecho. Por ejemplo, "s2" es la clave que corresponde con el valor "disk". Esta clave es arbitraria. Podríamos haberla llamado algo como "msg5" o "diskID.", por ejemplo. Sin embargo, una vez definida, la clave no debería cambiar porque es referenciada dentro del código fuente. De echo, cuando los localizadores crean un nuevo fichero de propiedades para acomodar idiomas adicionales, traducirán los valores a diferentes idiomas, pero no las claves.

2. Crear Ficheros de Propiedades Adiciones si son Necesarios

Para soportar una nueva Locale, los localizadores crearán un nuevo fichero de propiedades que contenga los valores traducidos. No se necesita cambiar el código fuente, ya que el programa referencia las claves, no los valores.

Por ejemplo, para añadir soporte para el idioma Alemán, los localizadores tendrán que traducir los valores de LabelsBundle.properties y situarlos en un fichero llamado LabelsBundle_de_DE.properties. Observa que el nombre de este fichero, al igual que el fichero por defecto, empieza con el nombre base LabelsBundle y termina con el sufijo .properties. Sin embargo, como el fichero se ha creado para una localidad específica, el nombre base es seguido por el código

del idioma (de) y el código del país (DE). El contenido de LabelsBundle_de_DE.properties es éste:

```
# This is the LabelsBundle_de_DE.properties file
s1 = Computer
s2 = Platte
s3 = Monitor
s4 = Tastatur
```

Hemos lanzado tres ficheros de propiedades con el programa de ejemplo PropertiesDemo:

```
LabelsBundle.properties
LabelsBundle_de_DE.properties
LabelsBundle_fr.properties
```

3. Especificar la Localidad

En el programa PropertiesDemo hemos creado los objetos Locale de esta forma:

```
Locale[] supportedLocales = {
    new Locale("fr", "FR"),
    new Locale("de", "DE"),
    new Locale("en", "US")
}

Locale currentLocale = new Locale("fr", "FR");
```

Para cada uno de estos objetos hemos especificado un código de idioma y un código de país. Estos códigos corresponden con los ficheros de propiedades creados en los pasos anteriores. Por ejemplo, el Locale creado con los códigos de y DE corresponde con el fichero LabelsBundle_de_DE.properties.

4. Crear el ResourceBundle

Este es el paso que muestra como se relacionan, la localidad, los ficheros de propiedades y el ResourceBundle. Para crear el ResourceBundle, llamamos al método getBundle, especificando el nombre base y la localidad:

```
ResourceBundle labels =
    ResourceBundle.getBundle("LabelsBundle",currentLocale);
```

El método getBundle primero busca un fichero de clase que corresponda con el nombre base. Si no puede encontrar el fichero de clase, comprueba los ficheros de propiedades. En el programa PropertiesDemo, hemos constituido el ResourceBundle con ficheros de propiedades en vez de ficheros de clases. Cuando el método getBundle localiza el fichero de propiedades correcto, devuelve un objeto PropertyResourceBundle cargado con las parejas clave-valor del

fichero de propiedades.

Si no existe un fichero de propiedades para la localidad especificada, `getBundle` selecciona el fichero de propiedades con la correspondencia más cercana. La siguiente tabla identifica los ficheros de propiedades que buscará el programa `PropertiesDemo` para cada localidad:

Parámetros Locale	Fichero de Propiedades	Explicación
de DE	LabelsBundle_de_DE.properties	Correspondencia Exacta.
fr FR	LabelsBundle_fr.properties	LabelsBundle_fr_FR.properties no existe, pero es la correspondencia más cercana.
en US	LabelsBundle.properties	Se selecciona el fichero por defecto porque los parámetros de la localidad no corresponden.

En lugar de llamar a `getBundle`, podríamos haber creado el objeto `PropertyResourceBundle` llamando a su constructor, que acepta un `InputStream` como argumento. Para crear el `InputStream` debemos especificar el nombre exacto del fichero de propiedades en la llamada al constructor de `FileInputStream`. Crear el `PropertyResourceBundle` llamando al método `getBundle` es más flexible, porque buscará los ficheros de propiedades con la correspondencia más cercana a la localidad especificada.

5. Obtener el Texto Localizado

Para recuperar los valores traducidos desde el `ResourceBundle`, llamamos al método `getString`:

```
String value = labels.getString(key);
```

El `String` devuelto por `getString` corresponde con la clave que hemos especificado. El `String` está en el idioma apropiado, proporcionado por un fichero de propiedades existente para la localidad especificada. Como las claves no cambian, los localizadores añaden ficheros de propiedades adicionales posteriormente,. Nuestra llamada a `getString` no necesita cambiar.

6. Iterar a través de todas las Claves

Si queremos recuperar los valores para todas las claves de un `ResourceBundle`, necesitamos llamar al método `getKeys`. Este método devuelve una `Enumeration` con todas las claves de un `ResourceBundle`. Se puede iterar a través de la `Enumeration` y recuperar cada valor con el método `getString`. Las siguientes líneas de código del programa `PropertiesDemo`, muestran como se hace esto:

```
ResourceBundle labels =  
    ResourceBundle.getBundle("LabelsBundle",currentLocale);
```

```
Enumeration bundleKeys = labels.getKeys();

while (bundleKeys.hasMoreElements()) {
    String key = (String)bundleKeys.nextElement();
    String value = labels.getString(key);
    System.out.println("key = " + key + ", " +
        "value = " + value);
}
```

Ozito

Utilizar un ListResourceBundle

Esta sección ilustra el uso de un objeto ListResourceBundle con un programa de ejemplo llamado ListDemo. Explicaremos cada paso involucrado en la creación del programa ListDemo, junto con las subclases de ListResourceBundle que soporta. El código fuente del programa está en [ListDemo.java](#). También podrías querer examinar la [salida](#) producida por el programa.

1. Crear las Subclases de ListResourceBundle

Un ListResourceBundle está constituido por un fichero de clase. Por lo tanto, nuestro primer paso es crear el fichero de clase para cada Localidad soportada. En el programa ListDemo, el nombre base del ListResourceBundle es StatsBundle. Cómo ListDemo soporta tres localidades diferentes, requiere los siguientes ficheros de clases:

```
StatsBundle_en_CA.class  
StatsBundle_fr_FR.class  
StatsBundle_ja_JA.class
```

La clase StatsBundle para Japón está definida en el código fuente de la siguiente forma. Observa que el nombre de la clase está construido añadiendo el código de idioma y del país al nombre base del ListResourceBundle. Dentro de la clase, se inicializa un array de dos dimensiones contents con parejas de clave valor. Las claves son el primer elemento de cada pareja: GDP, Population, y Literacy. Las claves deben ser objetos String, y deben ser iguales en cada fichero class del conjunto StatsBundle. Los valores pueden ser cualquier tipo de objeto. En este ejemplo, los valores son dos objetos Integer y un objeto Float.

```
import java.util.*;  
  
public class StatsBundle_ja_JA extends ListResourceBundle {  
  
    public Object[][] getContents() {  
        return contents;  
    }  
  
    private Object[][] contents = {  
        {"GDP", new Integer(21300)},  
        {"Population", new Integer(125449703)},  
        {"Literacy", new Double(0.99)},  
    };  
}
```

2. Especificar la Localidad

En el programa ListDemo, hemos definido los siguientes objetos Locale:

```
Locale[] supportedLocales = {  
    new Locale("en", "CA"),  
    new Locale("ja", "JA"),  
    new Locale("fr", "FR")  
};
```

Cada objeto Locale corresponde a una clase de StatsBundle. Por ejemplo, la localidad Japonesa, que fue definida con los códigos ja y JA, corresponde con StatsBundle_ja_JA.class.

3. Crear el ResourceBundle

Para crear el ListResourceBundle, llamamos al método getBundle. En la siguiente línea de código, observa que hemos especificado el nombre base de la clase (StatsBundle) y la Localidad.

```
ResourceBundle stats =  
    ResourceBundle.getBundle("StatsBundle", currentLocale);
```

El método getBundle buscará una clase cuyo nombre empiece con StatsBundle y esté seguido por los códigos de idioma y de país especificados por la Locale. Por ejemplo, si currentLocale se crea con los códigos ja y JA, getBundle devuelve un objeto ListResourceBundle cargado con la clase StatsBundle_ja_JA.

4. Recuperar Objetos Localizados

Ahora que tenemos un ListResourceBundle para la Localidad apropiada, recuperaremos los objetos localizados por sus claves. En la siguiente línea de código, recuperaremos el ratio de alfabetización llamando al método getObject con el parámetro "Literacy". Como getObject devuelve un objeto, debemos tiparlo a Double:

```
Double lit = (Double)stats.getObject("Literacy");
```

Formateo de Números y Monedas

Los programas almacenan y operan con números de una forma independiente de la Localidad. Antes de mostrar o imprimir un número el programa debe convertirlo a un String que esté en un formato sensible a la Localidad. Por ejemplo, en Francia, el número 123456.78 debería ser formateado como 123 456,78, y en Alemania debería aparecer como 123.456,78. En esta lección, aprenderás como hacer que tus programas sean independientes de las convenciones de la localidad para los puntos decimales, los separadores de millares, y otras propiedades de formateo.

Utilizar Formatos Predefinidos

Utilizando los métodos de factoría proporcionados por la clase `NumberFormat`, se pueden obtener formatos específicos de la localidad para números, monedas y porcentajes.

Formatear con Patrones

Con la clase `DecimalFormat` se especifica un formato de número con patrón. La clase `DecimalFormatSymbols` permite modificar los símbolos de formateo como separadores decimales o el signo negativo.

Utilizar Formatos Predefinidos

Llamando a los métodos de la clase [NumberFormat](#) se pueden formatear números, monedas y porcentajes de acuerdo a la Localidad. Sin embargo, hay un error: NumberFormat podría no soportar la Localidad especificada. Para conocer las definiciones de Locale soportadas por NumberFormat, llamamos al método `getAvailableLocales`:

```
Locale[] locales = NumberFormat.getAvailableLocales();
```

Si NumberFormat no soporta la localidad que necesitamos, podemos crear nuestro propio formateador. Explicaremos este procedimiento en la siguiente sección, [Formatear con Patrones](#).

El siguiente material muestra como obtener formateadores específicos de la localidad para números, monedas y porcentajes. Los códigos de ejemplo son de un programa llamado [NumberFormatDemo.java](#).

Números

Se pueden utilizar los métodos de factoria de NumberFormat para formatear números de tipos de datos primitivos, como double, y sus correspondientes objetos, como Double.

En el siguiente fragmento de código, formatearemos un Double de acuerdo a la Localidad. Primero, obtendremos un ejemplar de NumberFormat específico de la Localidad, llamando a `getNumberInstance`. Luego llamamos al método `format` con el Double como argumento. El método `format` devuelve el número formateado en un String, que está listo para ser mostrado.

```
Double amount = new Double(345987.246);
NumberFormat numberFormatter;
String amountOut;

numberFormatter = NumberFormat.getNumberInstance(currentLocale);
amountOut = numberFormatter.format(amount);
System.out.println(amountOut + "    " + currentLocale.toString());
```

La salida de este ejemplo muestra como el formato del mismo número varía con la Localidad:

```
345 987,246    fr_FR
345.987,246    de_DE
345,987.246    en_US
```

Monedas

Si estás escribiendo aplicaciones de negocios, necesitarás formatear y mostrar valores en monedas. Las monedas se formatean de la misma forma que los números, excepto en que se llama a `getCurrencyInstance` para crear el formateador. Cuando se llama al método `format`, devuelve un `String` que incluye el número formateado y el signo de moneda apropiado.

El siguiente código muestra como formatear moneda de una forma específica de la Localidad:

```
Double currency = new Double(9876543.21);
NumberFormat currencyFormatter;
String currencyOut;

currencyFormatter = NumberFormat.getCurrencyInstance(currentLocale);
currencyOut = currencyFormatter.format(currency);
System.out.println(currencyOut + "    " + currentLocale.toString());
```

La salida genera por las líneas precedentes sería esta:

```
9 876 543,21 F    fr_FR
9.876.543,21 DM   de_DE
$9,876,543.21    en_US
```

A primera vista, esta salida podría parecer errónea, porque los valores numéricos son iguales. Por supuesto, 9 876 543,21 F no es equivalente a 9.876.543,21 DM. Sin embargo, recuerda que la clase `NumberFormat` no puede cambiar monedas. Los métodos pertenecientes a la clase `NumberFormat` formatean monedas, pero no las convierten.

Porcentajes

También se pueden utilizar los métodos de la clase `NumberFormat` para formatear porcentajes. Para obtener el formateador específico de la localidad se llama al método `getPercentInstance`. Con este formateador, un fracción como 0.75 se mostrará como 75%.

El siguiente código muestra como formatear porcentajes:

```
Double percent = new Double(0.75);
NumberFormat percentFormatter;
String percentOut;

percentFormatter = NumberFormat.getPercentInstance(currentLocale);
percentOut = percentFormatter.format(percent);
```

Formatear con Patrones

Se puede utilizar la clase `DecimalFormat` para formatear números decimales en cadenas específicas de la Localidad. Esta clase permite controlar los ceros iniciales y finales, los sufijos y prefijos, separadores (millares), y el separador decimal. Si se quiere cambiar un símbolo del formateo como el saperador decimal, se puede utilizar `DecimalFormatSymbols` en conjunción con la clase `DecimalFormat`. Estas clases ofrecen una gran flexibilidad en el formateo de números, pero hacen el código más complejo. Siempre que sea posible, se debería utilizar la clase `NumberFormat`, que se describió en la sección anterior, en vez de `DecimalFormat` y `DecimalFormatSymbols`.

Utilizando ejemplos, las siguientes líneas mostrarán como utilizar las clases `DecimalFormat` y `DecimalFormatSymbols`. Los fragmentos de código se han extraído de un programa llamado [DecimalFormatDemo.java](#).

Construir Patrones

Las propiedades de formateo de `DecimalFormat` se especifican con un `String` patrón. El patrón determina la apariencia del número formateado. Para una descripción completa de la sintaxis de los patrones puedes ver la página [Sintaxis de los Patrones de Formateo de Números](#).

En el siguiente ejemplo, hemos creado un formateador, pasándole un patrón al constructor de `DecimalFormat`. Luego, le hemos pasado un valor `double` al método `format`, que devuelve un `String` formateado:

```
DecimalFormat myFormatter = new DecimalFormat(pattern);
String output = myFormatter.format(value);
System.out.println(value + " " + pattern + " " + output);
```

La salida de las líneas de código precedentes se describe en la siguiente tabla. El valor es el número, un `double`, que será formateado. El Patrón es el `String` que especifica las propiedades de formateo. La Salida, que es un `String`, representa el número formateado:

Valor	Patrón	Salida	Explicación
123456.789	###,###.###	123,456.789	El signo Almohadilla (#) indica un dígito, la coma la posición del separador de millares y el punto la posición del separador decimal.
123456.789	###.##	123456.79	El valor tiene tres dígitos a la derecha del punto decimal, pero el patrón sólo tiene dos. El método <code>format</code> maneja esto redondeando.
123.78	000000.000	000123.780	El patrón especifica relleno con ceros, porque se utiliza el caracter Cero en vez de la almohadilla (#).
12345.67	\$###,###.###	\$12,345.67	El primer carácter del patrón es el signo del dólar (\$). Observa que este signo precede inmediatamente al dígito más a la izquierda de la salida formateada.

12345.67	\u00a5###,###.###	¥12,345.67	El patrón especifica el signo del Yen Japonés (¥) con su valor Unicode \u00a5.
----------	-------------------	------------	--

Formateo Sensible a la Localidad

El ejemplo anterior creaba un objeto `DecimalFormat` para la Localidad por defecto. Si se quiere un objeto `DecimalFormat` para una localidad distinta, se ejemplariza un `NumberFormat` y luego se fuerza a `DecimalFormat`. Entonces, el objeto `DecimalFormat` formateará los patrones definidos de una forma sensible a la localidad. Aquí se puede ver un ejemplo:

```
NumberFormat nf = NumberFormat.getNumberInstance(loc);
DecimalFormat df = (DecimalFormat)nf;
df.applyPattern(pattern);
String output = df.format(value);
System.out.println(pattern + " " + output + " " + loc.toString());
```

Abajo podemos ver el resultado de la ejecución de este código. Los números formateados, en la segunda columna, varían con la Localidad:

```
###,###.###  123,456.789  en_US
###,###.###  123.456,789  de_DE
###,###.###  123 456,789  fr_FR
```

Por eso, los patrones de formateo que hemos utilizado hasta ahora seguían las convenciones del Inglés Norteamericano. Por ejemplo, en el patrón `"###,###.###"` la coma es el separador de millares y el punto representa el punto decimal. Esta convención está bien, sabiendo que los usuarios finales no están expuestos a ellas. Sin embargo, algunas aplicaciones, como las hojas de cálculo y los generadores de informes, permiten al usuario final definir sus propios patrones de formateo. Para esas aplicaciones, los patrones de formateo especificados por el usuario final deberían utilizar la notación localizada. En estos casos querremos llamar al método `applyLocalizedPattern` sobre el objeto `DecimalFormat`.

Modificar los Símbolos de Formateo

Con la clase [DecimalFormatSymbols](#) se pueden modificar los símbolos que aparecen en los números formateados producidos por el método `format`. Estos símbolos incluyen el punto decimal, el separador de millares, el signo menos, y el signo de porcentaje además de algunos más.

En el siguiente ejemplo podemos ver el uso de `DecimalFormatSymbols` aplicando un formato inusual a un número. Empezamos ejemplarizando `DecimalFormatSymbols` sin argumentos, lo que devuelve un objeto para la Localidad por defecto, (Como `DecimalFormatSymbols` es una clase sensible a la Localidad, podríamos haber especificado una Localidad cuando llamamos al constructor.) Luego, modificamos el separador decimal y el de millares. Luego especificamos el `DecimalFormatSymbols` cuando ejemplarizamos la clase `DecimalFormat`. Sólo para hacer las cosas más complicadas, cambiamos la numero de agrupamiento del formateador de tres a cuatro. Finalmente llamamos al método `format`.

Aquí tienes el código de Ejemplo:

```
DecimalFormatSymbols unusualSymbols = new DecimalFormatSymbols();
unusualSymbols.setDecimalSeparator('|');
unusualSymbols.setGroupingSeparator('^');

String strange = "#,##0.###";
DecimalFormat weirdFormatter = new DecimalFormat(strange, unusualSymbols);
weirdFormatter.setGroupingSize(4);

String bizarre = weirdFormatter.format(12345.678);
System.out.println(bizarre);
```

Esto es lo que aparecerá cuando imprimamos este extraño número formateado

1^2345|678

Formateo de Fechas y Horas

Los objetos Date representan fechas y horas. No se puede mostrar o imprimir un objeto Date sin convertirlo primero a un String que esté en el formato apropiado. Pero, ¿Cuál es el formato adecuado? Primero, el formato debería estar conforme a las convenciones de la Localidad del usuario final. Por ejemplo, los alemanes reconocerán esto 9.4.98 como una fecha válida, pero los norteamericanos esperarán que la misma fecha aparezca como 4/9/98. Segundo, el formato debería incluir la información necesaria. Por ejemplo, un programa que mida el rendimiento de una red podría mostrar milisegundos. Probablemente un calendario de citas on-line no mostrará milisegundos, pero si que mostrará los días de la semana.

Esta lección explica como formatear fechas y horas en diferentes formas, y de una manera sensible a la Localidad. Si sigues las técnicas descritas en esta lección, tus programas no sólo mostrarán las fechas y horas en la Localidad apropiada, sino que el código fuente permanecerá independiente de cualquier Localidad especificada.

Utilizar Formato Predefinidos

La clase DateFormat proporciona estilos de formateo predefinidos que son específicos de la Localidad y fáciles de utilizar.

Formatear con Patrones

Con la clase SimpleDateFormat, se pueden crear formatos personalizados específicos de la Localidad.

Cambiar los Símbolos de Formateo

Utilizando la clase DateFormatSymbols, se pueden cambiar los símbolos que representan los nombres de los meses, de los días de la semana, y otros elementos de formateo.

Utilizar Formatos Predefinidos

La clase [DateFormat](#) permite formatear fechas y horas con estilos predefinidos de una forma sensible a la Localidad, DateFormat no soporta todas las posibles definiciones de Localidades. Para ver las definiciones de Locale que reconoce DateFormat, llamamos al método `getAvailableLocales`:

```
Locale[] locales = DateFormat.getAvailableLocales();
```

En las siguientes secciones, veremos cómo formatear fechas y horas con la clase DateFormat. Los ejemplos se han sacado el programa [DateFormatDemo.java](#).

Fechas

Formatear fechas con la clase DateFormat es un proceso en dos pasos. Primero se crea un formateador con el método `getDateInstance`. Segundo, se llama al método `format`, que devuelve un String que contiene la fecha formateada. En el siguiente ejemplo, hemos formateado la fecha de hoy llamando a estos dos métodos:

```
Date today;
String dateOut;
DateFormat dateFormatter;

dateFormatter =
    DateFormat.getDateInstance(DateFormat.DEFAULT, currentLocale);
today = new Date();
dateOut = dateFormatter.format(today);

System.out.println(dateOut + "    " + currentLocale.toString());
```

Aquí está la salida generada por este código. Observa que los formatos de fechas varían con la Localidad. Como DateFormat es sensible a la Localidad, tiene cuidado de los detalles de formateo para cada Localidad.

```
9 avr 98    fr_FR
9.4.1998    de_DE
09-Apr-98    en_US
```

En el ejemplo de código anterior, hemos especificado el estilo de formato DEFAULT. Este estilo es sólo uno de los estilos de formato predefinidos que proporciona la clase DateFormat:

- DEFAULT
- SHORT
- MEDIUM

- LONG
- FULL

La siguiente tabla muestra cómo se formatea cada estilo para las Localidades de U.S. y Francia:

Estilo	Localidad U.S.	Localidad Francia
DEFAULT	10-Apr-98	10 avr 98
SHORT	4/10/98	10/04/98
MEDIUM	10-Apr-98	10 avr 98
LONG	April 10, 1998	10 avril 1998
FULL	Friday, April 10, 1998	vendredi, 10 avril 1998

Horas

Los objetos Date representan tanto fechas como horas. Formatear horas con la clase DateFormat es similar al formateo de fechas, excepto en que el formateador se crea con el método getInstance:

```
DateFormat timeFormatter =
    DateFormat.getInstance(DateFormat.DEFAULT, currentLocale);
```

La siguiente tabla muestra los estilos de formatos predefinidos para las horas en las localidades de U.S. y Alemania:

Estilo	Localidad U.S.	Localidad Alemania
DEFAULT	3:58:45 PM	15:58:45
SHORT	3:58 PM	15:58
MEDIUM	3:58:45 PM	15:58:45
LONG	3:58:45 PM PDT	15:58:45 GMT+02:00
FULL	3:58:45 oclock PM PDT	15.58 Uhr GMT+02:00

Fechas y Horas

Para mostrar la fecha y la hora en la misma String, se crea el formateador con el método getDateTimeInstance. El primer parámetro es el estilo de fecha y el segundo es el estilo de la hora. El tercer parámetro es nuestro viejo amigo Locale. Aquí tienes un ejemplo rápido:

```
DateFormat formatter;

formatter = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                           DateFormat.LONG,
                                           currentLocale);
```

Para la localidad de U.S., el formateado de este ejemplo, formateará la fecha y la

hora de esta forma:

April 10, 1998 4:05:54 PM PDT

Si el ejemplo del formateador se hubiera creado con la Localidad Francia, el String devuelto por el método format sería:

11 avril 1998 01:05:54 GMT+02:00

Ozito

Formatear con Patrones

En la sección anterior, [Utilizar formatos predefinidos](#), describimos los estilos de formatos proporcionados por la clase DateFormat. En la mayoría de los casos, estos formatos predefinidos son suficiente. Sin embargo, si se quieren crear formatos personalizados, se debe utilizar la clase [SimpleDateFormat](#).

En la siguientes líneas, proporcionaremos varios ejemplos de código que demuestran el uso de los métodos de la clase SimpleDateFormat. Podrás encontrar el código completo en el fichero llamado [SimpleDateFormatDemo.java](#).

Sobre los Patrones

Cuando se crea un objeto SimpleDateFormat, se especifica un patrón en un String. Los contenidos de este patrón determinan el formato de la fecha y de la hora. Para una descripción completa sobre la sintaxis de los patrones puedes ver las tablas de la sección [Sintaxis de los Patrones de Formato de Fechas](#). Podremos ver algunos ejemplos de patrones en los siguientes ejemplos.

En el siguiente código hemos especificado el patrón cuando se crea el objeto SimpleDateFormat y luego llamamos al método format. El String devuelto por el método format contiene la fecha y hora formateadas y está lista para mostrarse.

```
Date today;
String output;
SimpleDateFormat formatter;

formatter = new SimpleDateFormat(pattern, currentLocale);
today = new Date();
output = formatter.format(today);

System.out.println(pattern + "    " + output);
```

La tabla siguiente muestra la salida generada por el código anterior cuando se especifica la localidad de U.S.:

patrón	Salida
dd.MM.yy	09.04.98
yyyy.MM.dd G 'at' hh:mm:ss z	1998.04.09 AD at 06:15:55 PDT
EEE, MMM d, 'yy	Thu, Apr 9, '98
h:mm a	6:15 PM
H:mm	18:15
H:mm:ss:SSS	18:15:55:624
K:mm a,z	6:15 PM,PDT

Patrones y Localidades

La clase SimpleDateFormat es sensible a la localidad. Si se ejemplariza SimpleDateFormat sin un parámetro Locale, formateará la fecha y hora de acuerdo a la Localidad por defecto. Tanto el patrón como la Localidad determinan el formato. Para el mismo patrón, la clase SimpleDateFormat podría formatear la fecha y la hora de forma diferente si varía la Localidad.

En el siguiente ejemplo, el patrón está codificado en la sentencia que crea el objeto SimpleDateFormat:

```
Date today;  
String result;  
SimpleDateFormat formatter;  
  
formatter = new SimpleDateFormat("EEE d MMM yy", currentLocale);  
today = new Date();  
result = formatter.format(today);
```

```
System.out.println("Locale: " + currentLocale.toString());  
System.out.println("Result: " + result);
```

El código anterior genera esta salida. Aunque el patrón esté codificado, el formato de la fecha cambia cada vez que se especifica una Localidad diferente:

```
Locale: fr_FR  
Result: ven 10 avr 98
```

```
Locale: de_DE  
Result: Fr 10 Apr 98
```

```
Locale: en_US  
Result: Thu 9 Apr 98
```


Cambiar los Símbolos de Formateo de Fechas

El método `format` de `SimpleDateFormat` devuelve un `String` formado por dígitos y símbolos. Por ejemplo, en el `String` "Friday, April 10, 1998," los símbolos son "Friday" y "April." Si los símbolos encapsulados en `SimpleDateFormat` no cubren tus necesidades, puedes cambiarlos con la clase [DateFormatSymbols](#). Se pueden cambiar los símbolos que representan los nombres de los meses, de los días de la semana, de las zonas horarias. etc.

Echemos un vistazo a un ejemplo que modifica los nombres cortos de los días de la semana. Podrás encontrar el código fuente de este ejemplo en el fichero llamado [DateFormatSymbolsDemo.java](#). En este ejemplo, empezamos creando un objeto `DateFormatSymbol` para la Localidad de U.S. Tenemos curiosidad sobre las abreviaturas que encapsula `DateFormatSymbol` para los días de la semana, por eso llamamos al método `getShortWeekdays`. Hemos decidido crear versiones en mayúsculas de estas abreviaturas en el array de `String` llamado `capitalDays`. Luego aplicamos el nuevo conjunto de símbolos de `capitalDays` al objeto `DateFormatSymbol` con el método `setShortWeekdays`. Finalmente, ejemplarizamos la clase `SimpleDateFormat`, especificando el `DateFormatSymbol` que tenía los nuevos nombres. Aquí está el código fuente:

```
Date today;
String result;
SimpleDateFormat formatter;
DateFormatSymbols symbols;
String[] defaultDays;
String[] modifiedDays;

symbols = new DateFormatSymbols(new Locale("en","US"));
defaultDays = symbols.getShortWeekdays();

for (int i = 0; i < defaultDays.length; i++) {
    System.out.print(defaultDays[i] + " ");
}
System.out.println();

String[] capitalDays = {
    "", "SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"};
symbols.setShortWeekdays(capitalDays);

modifiedDays = symbols.getShortWeekdays();
for (int i = 0; i < modifiedDays.length; i++) {
    System.out.print(modifiedDays[i] + " ");
}
```

```
System.out.println();
System.out.println();

formatter = new SimpleDateFormat("E", symbols);
today = new Date();
result = formatter.format(today);
System.out.println(result);
```

La salida generada por este código se muestra abajo. La primera línea contiene los nombres cortos de los días de la semana antes de cambiarlos. La segunda línea contiene los nombres en mayúsculas que hemos aplicado con el método `setShortWeekdays`. Estas dos primeras líneas parecen idénticas, porque la primera cadena del array de nombres es `null`. La última línea muestra el resultado devuelto por el método `SimpleDateFormat.format`.

Sun	Mon	Tue	Wed	Thu	Fri	Sat
SUN	MON	TUE	WED	THU	FRI	SAT

WED

Ozito

Formateo de Mensajes

A todos nos gusta utilizar programas que nos cuenten lo que están haciendo. Los programas que nos mantienen informados normalmente lo hacen mostrando mensajes de estado o de error. Por supuesto, estos mensajes necesitan ser traducidos para que puedan ser entendidos por los usuarios finales de todo el mundo. Explicamos la traducción de mensajes de textos en [Aislar Objetos Específicos de la Localidad en un ResourceBundle](#). Normalmente con mover el mensaje String dentro de un ResourceBundle, ya está hecho. Sin embargo, si un mensaje contiene datos variables tendremos que hacer unos pasos extra para preparar su traducción.

En la siguiente lista de mensajes, hemos subrayado los datos variables:

```
The disk named MyDisk contains 300 files.  
The current balance of account #34-98-222 is $2,745.72.  
405,390 people have visited your website since January 1, 1998.  
Delete all files older than 120 days.
```

Podríamos estar tentados a construir el último mensaje de la lista concatenando frases y variables:

```
double numDays;  
ResourceBundle mssgBundle;  
.  
.  
String message = mssgBundle.getString("deleteolder") +  
                  numDays.toString() +  
                  mssgBundle.getString("days");
```

Esta aproximación funciona bien en Inglés, pero no funciona en idiomas donde los verbos aparezcan al final de la sentencia. Como el orden de las palabras de este mensaje está codificado, los localizadores no podrán crear traducciones gramáticamente correctas para todos los idiomas.

¿Cómo podemos hacer que nuestros programas sean fácilmente localizables si necesitamos utilizar mensajes concatenados? Podemos hacerlo utilizando la clase MessageFormat, que es el punto principal de esta lección.

Tratar con Mensajes Concatenados

Un mensaje concatenado podría contener varios tipos de variables: fechas, horas, cadenas, números, monedas y porcentajes. Para formatear un mensaje concatenado de un forma independiente de la Localidad, se construye un patrón que luego se aplica a un objeto MessageFormat.

Manejar Plurales

Las palabras de un mensaje pueden variar si son en plural o singular. Con la clase `ChoiceFormat`, se puede mapear un número a una palabra o frase, permitiendo construir mensajes que sean gramáticamente correctos.

Tratar con Mensajes Concatenados

En esta sección pasaremos a través de un programa de ejemplo para demostrar cómo internacionalizar un mensaje concatenado. El programa de ejemplo hace uso de la clase [MessageFormat](#). El código fuente completo los puedes encontrar en [MessageFormatDemo.java](#).

1. Identificar las Variables del Mensaje

La versión inglesa del mensaje que queremos internacionalizar es esta:

At 1:15 PM on April 13, 1998, we detected 7 spaceships on the planet Mars.

Date

Date

Number

String

Observa que hemos subrayado las variables, y hemos identificado qué tipo de objetos representará estos datos.

2. Aislar el Patrón del Mensaje en un ResourceBundle

Vamos a almacenar el mensaje en un ResourceBundle llamado MessageBundle. Aquí tienes el código de creación del ResourceBundle:

```
ResourceBundle messages =  
    ResourceBundle.getBundle("MessageBundle",currentLocale);
```

Este ResourceBundle está compuesto por un fichero de propiedades para cada Localidad. Como nuestro ResourceBundle se llama MessageBundle, el fichero de propiedades para la Localidad U.S. English se llamará MessageBundle_en_US.properties. Aquí tienes los contenidos de este fichero de propiedades:

```
template = At {2,time,short} on {2,date,long}, we detected {1,number,integer}  
spaceships on the planet {0}.  
planet = Mars
```

Hemos especificado el patrón en la primera línea del fichero de propiedades. Si comparamos este patrón con el mensaje de texto mostrado en el paso 1, veremos que se ha reemplazado cada variable del mensaje con un argumento encerrado entre corchetes. Cada argumento empieza con un dígito llamado el número de argumento, que corresponde con el índice de un elemento en un array de Object que contiene los valores de los argumentos. Observa que en el patrón, estos argumentos no están en ningún orden particular. Se pueden situar los argumentos en cualquier orden dentro del patrón. El único requerimiento es que el número del argumento tenga un elemento correspondiente en el array de valores de los argumentos. En el siguiente paso explicaremos el array de valores de argumentos, pero primero echemos un vistazo a todos los argumentos del patrón. La siguiente tabla proporciona algunos detalles sobre los argumentos.

Argumento	Descripción
{ 2,time,short}	La parte horaria de un objeto Date. El estilo "short" especifica el estilo de formato DateFormat.SHORT.

{ 2,date,long}	La parte de la fecha de un objeto Date. El mismo objeto Date se utiliza para las dos variables de fecha y hora. En el array de argumentos Object el índice que contiene el objeto Date es el 2.
{ 1,number,integer}	Un objeto Number, además cualificado con el estilo numérico "integer".
{ 0}	El String del ResourceBundle que corresponde con la clave "planet".

Para una descripción completa de la sintaxis de argumentos, puedes ver la documentación del API para la clase [MessageFormat](#).

3. Seleccionar los Argumentos del Mensaje

En las siguientes líneas de código, asignamos los valores para cada argumento del patrón. Los índices de los elementos del array `messageArguments` corresponden con los números de los argumentos del patrón. Por ejemplo, el elemento de índice 1, que es un `Integer(7)`, corresponde con el argumento `{ 1,number,integer}` del patrón. Extraemos los objetos `String`, que son los elementos 0 y 3, del `ResourceBundle` con `getString`, porque deben ser traducidos. El array de argumentos del mensaje se define de esta forma:

```
Object[] messageArguments = {
    messages.getString("planet"),
    new Integer(7),
    new Date()
};
```

4. Crear el Formateador

Luego, creamos un objeto `MessageFormat`. Seleccionamos la Localidad porque nuestro mensaje contiene objetos `Date` y `Number`, que deberían ser formateados de una manera sensible a la Localidad. Por ejemplo en inglés Norteamericano la fecha 4/13/98 está en el formato correcto, pero en francés debería formatearse como 13/04/98. Creamos el formateador de mensajes de la siguiente forma:

```
MessageFormat formatter = new MessageFormat("");
formatter.setLocale(currentLocale);
```

5. Formatear el Mensaje utilizando el Patrón y los Argumentos

En este paso, demostraremos como trabajan juntos, el patrón, los argumentos del mensaje y el formateador. Primero, extraemos el `String` del patrón del `ResourceBundle` con el método `getString`. La clave para el patrón es "template." Pasamos el patrón al formateador con el método `applyPattern`. Luego, formateamos el mensaje utilizando el array de argumentos del mensaje llamando al método `format`. El `String` devuelto por este método ya está listo para utilizar. Todo esto se realiza con sólo dos líneas de código:

```
formatter.applyPattern(messages.getString("template"));
String output = formatter.format(messageArguments);
```

6. Ejecutar el Programa de Demostración

Veamos que sucede cuando el programa se ejecuta con la Localidad U.S. English:

```
% java MessageFormatDemo en US
```

```
currentLocale = en_US
```

At 1:15 PM on April 13, 1998, we detected 7 spaceships on the planet Mars.

Cuando se ejecuta el programa para la Localidad Alemana, observamos que la fecha y la hora han sido localizadas:

```
% java MessageFormatDemo de DE
```

```
currentLocale = de_DE
```

Um 13.15 Uhr am 13. April 1998 haben wir 7 Raumschiffe auf dem Planeten Mars entdeckt.

Ozito

Manejar Plurales

En Inglés, las formas plural y singular de una palabra normalmente son diferentes. Esto puede representar un problema cuando se construyen mensajes que se refieren a cantidades. Por ejemplo, si el mensaje informa del número de ficheros en un disco, son posibles las siguientes variaciones:

```
There are no files on XDisk.  
There is one file on XDisk.  
There are 2 files on XDisk.
```

La forma más rápida de resolver este problema es crear un patrón de MessageFormat como éste:

```
There are {0,number} file(s) on {1}.
```

Desafortunadamente, este patrón resulta gramáticamente incorrecto:

```
There are 1 file(s) on XDisk.
```

Podemos hacer algo mejor que esto utilizando la clase [ChoiceFormat](#). En esta sección, veremos como trazar con plurales en un mensaje, pasando a través de un programa de ejemplo llamado [ChoiceFormatDemo.java](#). Este programa también hace uso de la clase MessageFormat que se describió en la sección anterior, [Tratar con Mensajes Concatenados](#).

1. Definir el Patrón del Mensaje

Primero, identifiquemos las variables de nuestro mensaje:

There	are no files	on	XDisk	.
There	is one file	on	XDisk	.
There	are 2 files	on	XDisk	.

	^		^	
	variable		variable	

Luego, reemplazamos las variables del mensaje con argumentos, creando un patrón que puede aplicarse a un objeto MessageFormat:

```
There {0} on {1}.
```

Es muy sencillo trabajar con el argumento para el nombre del disco, que está representado por {1}. Lo trataremos igual que cualquier otra variable String en un patrón MessageFormat. Este argumento corresponde con el elemento 1 del array de valores (Ver paso 7).

Tratar con el argumento {0} es más complejo por un par de razones:

- Primero, la frase que reemplaza este argumento varía con el número de ficheros. Para construir esta frase en tiempo de ejecución, necesitamos mapear el número de ficheros en un String particular. Por ejemplo, el número 1, mapeará el String "is one file." La clase ChoiceFormat permite realizar el mapeado necesario.
- Segundo, si el disco contiene varios ficheros, la frase incluirá un entero. La clase MessageFormat nos permite insertar números en una frase.

2. Crear un ResourceBundle

Aislaremos el texto del mensaje en un ResourceBundle porque debe ser traducido:

```
ResourceBundle bundle =  
    ResourceBundle.getBundle("ChoiceBundle",currentLocale);
```

Hemos decidido construir nuestro ResourceBundle con ficheros de propiedades. El fichero ChoiceBundle_en_US.properties contiene las siguientes líneas:

```
pattern = There {0} on {1}.  
noFiles = are no files  
oneFile = is one file  
multipleFiles = are {2} files
```

El contenido de este fichero de propiedades muestra cómo se construirán y formatearán los mensajes. La primera línea contiene el patrón para MessageFormat, que explicamos en el paso anterior. Las otras líneas contienen frases que reemplazarán el argumento {0} en el patrón. La frase para la clave "multipleFiles" contiene el argumento {2}, que será reemplazado por un número

Aquí podemos ver la versión francesa del fichero de propiedades ChoiceBundle_fr_FR.properties:

```
pattern = Il {0} sur {1}.  
noFiles = n' y a pas des fichiers  
oneFile = y a un fichier  
multipleFiles = y a {2} fichiers
```

3. Crear un formateador de Mensaje

En este paso, ejemplarizamos MessageFormat y seleccionamos su Localidad:

```
MessageFormat messageForm = new MessageFormat("");  
messageForm.setLocale(currentLocale);
```

4. Crear un formateador de Choice

El objeto `ChoiceFormat` nos permite elegir, basándose en un número `double`, un `String` particular. El rango de números `double` y los objetos `String` con los que se mapean, se especifican en arrays:

```
double[] fileLimits = {0,1,2};

String [] fileStrings = {
    bundle.getString("noFiles"),
    bundle.getString("oneFile"),
    bundle.getString("multipleFiles")
};
```

`ChoiceFormat` mapea cada elemento del array `double` con el elemento del array `String` que tiene el mismo índice. En nuestro código de ejemplo, el 0 mapea el `String` devuelto por la llamada a `bundle.getString("noFiles")`. Por coincidencia, en nuestro ejemplo, el índice es el mismo que el valor en el array `fileLimits`. si hubieramos seleccionado `fileLimits[0]` a 7, `ChoiceFormat` mapearía el número 7 con `fileStrings[0]`.

Especificamos los arrays `double` y `String` cuando ejemplarizamos `ChoiceFormat`:

```
ChoiceFormat choiceForm = new ChoiceFormat(fileLimits, fileStrings);
```

5. Aplicar el Patrón

¿Recuerdas el patrón que construimos en el paso 1? Ahora es el momento de recuperar el patrón del `ResourceBundle` y aplicarlo al objeto `MessageFormat`:

```
String pattern = bundle.getString("pattern");
messageForm.applyPattern(pattern);
```

6. Asignar lo formatos

En este paso, asignamos el objeto `ChoiceFormat` creado en el paso 4 al objeto `MessageFormat`:

```
Format[] formats = {choiceForm, null, NumberFormat.getInstance()};
messageForm.setFormats(formats);
```

El método `setFormats` asigna objetos `Format` a los argumentos del patrón del mensaje. Debemos llamar al método `applyPattern` antes del llamar al método `setFormats`. La siguiente tabla muestra cómo el array `Format` corresponde con los argumentos del patrón del mensaje:

Elemento del Array	Argumento del Patrón

choiceForm	{ 0}
null	{ 1}
NumberFormat.getInstance()	{ 2}

7. Seleccionar los Argumentos y el Formato del Mensaje

En tiempo de ejecución, asignamos las variables al array de argumentos que pasamos al objeto MessageFormat. Los elementos del array corresponden con los argumentos del patrón. Por ejemplo, messageArgument[1] mapea al argumento {1} del patrón, que es un String que contiene el nombre del disco. En el paso anterior, asignamos un objeto ChoiceFormat al argumento {0} del patrón. Por lo tanto, el número asignado a messageArgument[0] el String seleccionado por el objeto ChoiceFormat. Si messageArgument[0] es mayor o igual que 2, el String "are {2} files" reemplaza al argumento {0} del patrón. El número asignado a messageArgument[2] será substituido en lugar del argumento {2} del patrón. Intentaremos hacer esto con las siguientes líneas de código:

```
Object[] messageArguments = {null, "XDisk", null};

for (int numFiles = 0; numFiles < 4; numFiles++) {
    messageArguments[0] = new Integer(numFiles);
    messageArguments[2] = new Integer(numFiles);
    String result = messageForm.format(messageArguments);
    System.out.println(result);
}
```

8. Ejecutar el Programa de Demostración

Ejecutemos el programa para la Localidad U.S. English:

```
% java ChoiceFormatDemo en US
```

```
currentLocale = en_US
```

```
There are no files on XDisk.
There is one file on XDisk.
There are 2 files on XDisk.
There are 3 files on XDisk.
```

Compara los mensajes mostrados por el programa con las frases del ResourceBundle del paso 2. Observa que el objeto ChoiceFormat selecciona la frase correcta, que el objeto MessageFormat utiliza para construir el mensaje apropiado.

La versión francesa del mensaje también parece correcta:

```
% java ChoiceFormatDemo fr FR
```

```
currentLocale = fr_FR
```

```
Il n' y a pas des fichiers sur XDisk.
```

```
Il y a un fichier sur XDisk.
```

```
Il y a 2 fichiers sur XDisk.
```

```
Il y a 3 fichiers sur XDisk.
```

Ozito

Trabajar con Excepciones

En la lección, [Manejo de Errores utilizando Excepciones](#), mostramos como lanzar y capturar excepciones. Como podrás recordar, los mensajes de excepciones mostrados por los ejemplos de esta lección están en Inglés. Frecuentemente, los mensajes de excepciones sólo se utilizan para propósitos de depurado y nunca son vistos por lo usuarios finales. Sin embargo, si estos usuarios están expuestos a mensajes de excepciones, deberemos asegurarnos de que no están codificados en cualquier otro idioma.

En esta lección, daremos algunos trucos para hacer los mensajes de excepciones independientes de la Localidad. Los ejemplos de código de esta sección incluyen objetos ResourceBundle y MessageFormat. Si no lo has hecho anteriormente, deberías leer primero [Aislar Objetos Específicos de la Localidad en un ResourceBundle](#), y [Formateo de Mensajes](#).

Manejar Mensajes de Excepciones Codificados

Si el texto del mensaje de un subclase de Exception ha sido codificado, no puede ser localizado, Pero existe un atajo, que explicaremos en esta sección.

Crear Subclases de Exception Independientes de la Localidad

Si creamos una subclase de Exception, deberíamos asegurarnos de que no contiene un mensajes codificado en su interior. Nuestras subclases, deberían implementar el método `getLocalizedMessage`.

Manejar Mensajes de Excepciones Codificados Internamente

Supongamos que llamamos a un método que lanza una excepción, y queremos mostrar el mensaje de la excepción cuando ocurra el error. Si este método pertenece a una clase de un paquete escrito por otras personas, no tendremos ningún control sobre el texto del mensaje a menos que tengamos acceso al código fuente. Este puede ser un grave problema si el texto del mensaje ha sido codificado, porque no podrá ser traducido a otros idiomas. En el siguiente código de ejemplo, encontramos este problema cuando abrimos un fichero que no existe:

```
static public void defaultMessage() {  
  
    try {  
        FileInputStream in = new FileInputStream("vapor.txt");  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("e.getMessage = " + e.getMessage());  
        System.out.println("e.getLocalizedMessage = " + e.getLocalizedMessage());  
        System.out.println("e.toString = " + e.toString());  
    }  
}
```

La clausula Catch del método defaultMessage genera la siguiente salida:

```
e.getMessage = vapor.txt  
e.getLocalizedMessage = vapor.txt  
e.toString = java.io.FileNotFoundException: vapor.txt
```

Estos mensajes son inadecuados por varias razones. Primero, el texto del mensaje está codificado dentro de la clase FileNotFoundException y no puede ser localizado. Intentamos imprimir el String devuelto por getLocalizedMessage, pero como getLocalizedMessage no ha sido implementada, sólo devuelve el mismo mensaje que el método getMessage. Además, getMessage sólo devuelve el nombre del fichero, que por sí mismo no es un mensaje traducible. Aunque podríamos encontrar útil el tener el nombre del fichero durante la depuración, nuestros usuarios finales necesitarán un mensaje más informativo.

En general, los mensajes proporcionados por subclases de Exception, como FileNotFoundException, se han creado para programadores, no para usuarios finales. Por ejemplo, un procesador de textos no debería mostrar, "java.io.FileNotFoundException: vapor.txt," cuando el usuario trata de abrir un fichero que no existe. Este mensaje, aunque es técnicamente correcto, no tiene sentido para la mayoría de la gente. En su lugar, cuando el procesador de textos capturara la FileNotFoundException, debería mostrar este mensaje: "Cannot find the file named vapor.txt. Make sure the file exists."

En una aplicación como un procesador de textos, el texto devuelto por FileNotFoundException.getMessage no es apropiado para los usuarios finales. Deberíamos proporcionar un mensaje que sea fácil de entender, y que pueda ser traducido. Veremos como hacer esto con el siguiente ejemplo.

Como el texto del mensaje necesita ser traducido, lo aislaremos en un ResourceBundle. Almacenaremos nuestros mensajes en un fichero de propiedades, construiremos un

ResourceBundle llamado ExceptionBundle. Aquí tenemos el mensaje del fichero ExceptionBundle_en_US.properties:

```
template = Cannot find the file named {0}. Make sure the file exists.
```

En el siguiente código mostramos un mensaje sensible a la localidad cuando se captura FileNotFoundException. Primero creamos un ResourceBundle para la Localidad apropiada. Luego recuperamos el patrón del mensaje traducido desde el ResourceBundle. Finalmente, aplicamos el patrón del formato para insertar el nombre del fichero en el mensaje:

```
static public void customMessage(Locale currentLocale) {

    System.out.println("Locale: " + currentLocale.toString());
    String fileName = "vapor.txt";

    try {
        FileInputStream in = new FileInputStream(fileName);
    }
    catch (FileNotFoundException e) {
        ResourceBundle messages =
            ResourceBundle.getBundle("ExceptionBundle",currentLocale);
        Object[] messageArguments = {fileName};
        MessageFormat formatter = new MessageFormat("");
        formatter.setLocale(currentLocale);
        formatter.applyPattern(messages.getString("template"));
        String errorOut = formatter.format(messageArguments);
        System.out.println(errorOut);
    }
}
```

La salida del método customMessage está bastante mejor que:
"java.io.FileNotFoundException: vapor.txt."

```
Locale: en_US
Cannot find the file named vapor.txt. Make sure the file exists.
```

```
Locale: de_DE
Die Datei vapor.txt konnte nicht gefunden werden.
Stellen Sie sicher, daß die Datei existiert.
```

Crear Subclases de Excepciones Independientes de la Localidad

Si queremos crear un subclase de `Exception` cuando necesitamos manejar errores específicos de nuestras aplicaciones, podemos crear nuestras subclases independientes de la Localidad sobrescribiendo su método `getLocalizedMessage`. Cada método que sobrescribamos debería aislar el mensaje que devuelve en un `ResourceBundle`. Esto permitirá que el mensaje sea traducido a varios idiomas durante la localización.

En el siguiente ejemplo, veremos cómo crear una subclase de `Exception` que muestra un mensaje independiente de la Localidad. El código fuente de este programa está en estos dos ficheros: [OverLimitException.java](#) y [Account.java](#).

El programa `Account` simula una cuenta de crédito. Si este programa se excede el límite de crédito, recupera un mensaje localizado desde un `ResourceBundle` llamado `ExceptionBundle`, y luego muestra el mensaje. En el fichero `ExceptionBundle_en_US.properties`, hemos especificado este mensaje:

```
pattern = Negative balance of {0,number,currency} is not allowed.
```

El programa `Account` recupera el mensaje con el método `getLocalizedMessage`, que hemos implementado en una subclase de `Exception` llamada `OverLimitException`. El método `getLocalizedMessage` acepta un objeto `Locale` como parámetro. Especificamos la Localidad cuando recuperamos el patrón del mensaje desde el `ResourceBundle`, y también cuando definimos el objeto `MessageFormat`. Por lo tanto, nuestra clase `OverLimitException` es sensible a la Localidad:

```
public class OverLimitException extends Exception {

    private double detail;

    public OverLimitException (double amount) {
        detail = amount;
    }

    public String getMessage() {
        return getLocalizedMessage(Locale.getDefault());
    }

    public String getLocalizedMessage(Locale currentLocale) {
        ResourceBundle messages =
            ResourceBundle.getBundle("ExceptionBundle",currentLocale);
        Object[] messageArguments = {new Float(detail)};
```



```

        MessageFormat formatter = new MessageFormat("");
        formatter.setLocale(currentLocale);
        formatter.applyPattern(messages.getString("pattern"));
        return formatter.format(messageArguments);
    }
}

```

En la clase Account, el método withdraw lanza un OverLimitException si el nuevo balance excede el límite de crédito. En el siguiente código, observa que hemos pasado el parámetro value al constructor OverLimitException en la sentencia throw. El constructor asigna este parámetro al campo detail de la clase OverLimitException. El método getLocalizedMessage inserta una representación String del campo detail en el mensaje devuelto. El método withdraw es el siguiente:

```

public void withdraw(double amount) throws OverLimitException {
    double value = balance - amount;
    if (value < creditLimit) {
        throw new OverLimitException(value);
    }
    else {
        balance = value;
    }
}

```

En el método main de la clase Account, llamamos al método withdraw y captura el OverLimitException siempre que se lance. La clausula catch imprime el String devuelto por getLocalizedMessage. Aquí tienes el método main:

```

static public void main(String[] args) {

    Locale[] locales = {
        new Locale("en", "US"),
        new Locale("de", "DE")
    };

    Account credit = new Account();
    credit.deposit(20.00f);

    for (int k = 0; k < locales.length; k++) {
        try {
            credit.withdraw(1000.00f);
        }
        catch (OverLimitException e) {
            System.out.println("Locale: " + locales[k].toString());
            System.out.println(e.getLocalizedMessage(locales[k]));
        }
    }
}

```

```
    }  
  } // for  
}
```

El programa Account muestra dos mensajes localizados. No sólo el texto se muestra en el idioma correcto, si no que también el formato de moneda es el adecuado para cada Localidad.:

```
% java Account
```

```
Locale: en_US  
Negative balance of ($980.00) is not allowed.
```

```
Locale: de_DE  
Ihr Konto um -980,00 DM zu überziehen ist nicht gestattet.
```

Comparar Strings

Las aplicaciones que buscan u ordenan texto realizan frecuentes comparaciones. Por ejemplo, un navegador necesita comparar la igualdad de dos cadenas cuando los usuarios finales buscan algún texto. Un escritor de informes realiza comparaciones de cadenas cuando ordena un lista de cadenas en orden alfabético.

Si la audiencia de nuestra aplicación está limitada a personas de habla inglesa, probablemente podamos hacer las comparaciones con el método `String.compareTo`. Este método realiza una comparación binaria de caracteres Unicode dentro de las cadenas. Para muchos idiomas, esta comparación binaria no es suficiente para ordenar cadenas, porque los valores Unicode no corresponden con el orden relativo de los caracteres.

Afortunadamente, la clase [Collator](#) permite realizar comparaciones de cadenas en diferentes idiomas. En esta lección, veremos cómo utilizar la clase `Collator` para buscar y ordenar texto.

Realizar Comparaciones Independientes de la Localidad

Las reglas de comparación definen la secuencia de ordenación de los strings. Estas reglas pueden variar con la Localidad, porque los distintos idiomas naturales ordenan sus palabras de forma diferente. Utilizando las reglas predefinidas de la clase `Collator`, podemos ordenar cadenas de una forma independiente de la Localidad.

Personalizar la Reglas de Comparación

En algunos casos, las reglas predefinidas de comparación proporcionadas por la clase `Collator` podrían no ser suficiente. Por ejemplo, podríamos querer ordenar strings en un idioma cuya localidad no estuviera soportada por `Collator`. En esta situación, podemos definir nuestras propias reglas de comparación, y asignarlas a un objeto `RuleBasedCollator`.

Aumentar el Rendimiento de la Comparación

Con la clase `CollationKey`, podremos incrementar la eficiencia de la comparación de cadenas. Esta clase convierte objetos `String` a claves cortas que siguen las reglas de un objeto `Collator` dado.

Realizar Comparaciones Independientes de la Localidad

Se utiliza la clase `Collator` para realizar comparaciones independientes de la Localidad. Esta clase es sensible a la Localidad. Para ver las localidades soportadas por la clase `Collator`, se llama al método `getAvailableLocales`:

```
Locale[] locales = Collator.getAvailableLocales();
```

Para ejemplarizar la clase `Collator`, se llama al método `getInstance` y se especifica una Localidad:

```
Collator myCollator = Collator.getInstance(new Locale("en", "US"));
```

El método `getInstance` realmente devuelve un `RuleBasedCollator`, que es una subclase concreta de `Collator`. El objeto `RuleBasedCollator` contiene un conjunto de reglas que determinan el orden de ordenación de las cadenas para una localidad especificada. Estas reglas están predefinidas para cada localidad. Como estas reglas están encapsuladas dentro de `RuleBasedCollator`, los programas no necesitarán rutinas especiales para tratar con las reglas de comparación para varios idiomas.

Se llama al método `Collator.compare` para realizar comparaciones de cadenas independientes de la Localidad. Este método devuelve un entero menor que, igual que o mayor que cero cuando la cadena del primer argumento sea menor que, igual que o mayor que la cadena del segundo argumento.

```
System.out.println(myCollator.compare("abc", "def"));
System.out.println(myCollator.compare("rtf", "rtf"));
System.out.println(myCollator.compare("xyz", "abc"));
```

Aquí puedes ver la salida del programa anterior:

```
-1
0
1
```

Utilizaremos el método `compare` para realizar operaciones de ordenación. El programa de ejemplo, llamado [CollatorDemo.java](#) utiliza el método `compare` para ordenar un array de palabras Inglesas y Francesas. En este programa, veremos lo que sucede cuando se ordena un array con dos objetos `Collator` diferentes:

```
Collator fr_FRCollator = Collator.getInstance(new Locale("fr", "FR"));
Collator en_USCollator = Collator.getInstance(new Locale("en", "US"));
```

Nuestro método para ordenar, llamado `sortStrings`, puede ser utilizado con cualquier `Collator`. Observa que el método `sortStrings` llama al método

compare:

```
public static void sortStrings(Collator collator, String[] words) {  
    String tmp;  
    for (int i = 0; i < words.length; i++) {  
        for (int j = i + 1; j < words.length; j++) {  
            // Compare elements of the array two at a time.  
            if (collator.compare(words[i], words[j] ) > 0 ) {  
                // Swap words[i] and words[j]  
                tmp = words[i];  
                words[i] = words[j];  
                words[j] = tmp;  
            }  
        }  
    }  
}
```

El Collator Inglés ordena las palabras de esta forma:

```
peach  
pêche  
péché  
sin
```

De acuerdo a las reglas de comparación del idioma Francés, la lista anterior sería errónea. En Francés, "pêche" debería seguir a "péché" en una lista ordenada. Nuestro Collator Francés ordena del array de forma correcta:

```
peach  
péché  
pêche  
sin
```

Personalizar las Reglas de Comparación

En la sección anterior, explicamos como realizar comparaciones de cadenas utilizando diferentes reglas para cada localidad. Estas reglas de comparación determinan el orden de ordenación de las cadenas. Si las reglas predefinidas no son suficientes, podemos diseñar nuestras propias reglas y asignarlas a un objeto `RuleBasedCollator`.

Las reglas de comparación personalizadas están contenidas en un objeto `String` que se pasa al constructor de `RuleBasedCollator`. Aquí tenemos un ejemplo:

```
String simpleRule = "< a < b < c < d";
RuleBasedCollator simpleCollator = new RuleBasedCollator(simpleRule);
```

Para el objeto `simpleCollator` del ejemplo anterior, "a" es menor que "b," que es menor que "c," y así sucesivamente. El método `simpleCollator.compare` se referirá a estas reglas cuando compare cadenas. La sintaxis completa utilizada para construir una regla de comparación es más flexible y completa que ese sencillo ejemplo. Para una completa descripción de la sintaxis, puedes ver a la documentación del API para la clase [RuleBasedCollator](#).

En el siguiente ejemplo ordenamos una lista de palabras españolas con dos objetos `Collators`. El código completo de este ejemplo está en el fichero llamado [RulesDemo.java](#).

Empezamos definiendo nuestras propias reglas de ordenación para Inglés y Español. Hemos decidido ordenar las palabras en Español de la forma tradicional. Cuando se ordena de esta forma, las letras "ch," y "ll," y sus mayúsculas equivalentes. tienen su propia posición en el orden de ordenación. Estas parejas de caracteres se comparan como si fueran un sólo caracter. Por ejemplo, "ch" ordena com una sólo letra después de "cz". Observa las diferencias para los dos `Collators`:

```
String englishRules =
    ("< a,A < b,B < c,C < d,D < e,E < f,F " +
     "< g,G < h,H < i,I < j,J < k,K < l,L " +
     "< m,M < n,N < o,O < p,P < q,Q < r,R " +
     "< s,S < t,T < u,U < v,V < w,W < x,X " +
     "< y,Y < z,Z");
```

```
String smallnTilde = new String("\u00F1");
String capitalNTilde = new String("\u00D1");
```

```
String traditionalSpanishRules =
    ("< a,A < b,B < c,C " +
     "< ch, cH, Ch, CH " +
     "< d,D < e,E < f,F " +
```

```
"< g,G < h,H < i,I < j,J < k,K < l,L " +
"< ll, lL, Ll, LL " +
"< m,M < n,N " +
"< " + smallnTilde + ", " + capitalNTilde + " " +
"< o,O < p,P < q,Q < r,R " +
"< s,S < t,T < u,U < v,V < w,W < x,X " +
"< y,Y < z,Z");
```

En las siguientes líneas de código, creamos los Collators y llamamos a nuestra rutina de ordenación:

```
try {
    RuleBasedCollator enCollator =
        new RuleBasedCollator(englishRules);
    RuleBasedCollator spCollator =
        new RuleBasedCollator(traditionalSpanishRules);

    sortStrings(enCollator, words);
    printStrings(words);

    System.out.println();

    sortStrings(spCollator, words);
    printStrings(words);
}
catch (ParseException pe) {
    System.out.println("Parse exception for rules");
}
```

La rutina de ordenación, llamada sortStrings, es genérica. Ordena cualquier array de palabras de acuerdo a las reglas de cualquier objeto Collator:

```
public static void sortStrings(Collator collator, String[] words) {
    String tmp;
    for (int i = 0; i < words.length; i++) {
        for (int j = i + 1; j < words.length; j++) {
            // Compare elements of the words array
            if( collator.compare(words[i], words[j] ) > 0 ) {
                // Swap words[i] and words[j]
                tmp = words[i];
                words[i] = words[j];
                words[j] = tmp;
            }
        }
    }
}
```

Cuando ordenamos con las reglas de ordenación Inglesas el array aparecerá de esta forma:

```
chalina  
curioso  
llama  
luz
```

Compara la lista anterior con la siguiente, que está ordenada de acuerdo a las reglas de ordenación del Español tradicional:

```
curioso  
chalina  
luz  
llama
```

Ozito

Aumentar el Rendimiento de la Ordenación

La ordenación de largas listas de palabras consume mucho tiempo. Si el algoritmo de ordenación compara cadenas repetidamente podemos acelerar el proceso utilizando la clase `CollationKey`.

Un objeto [CollationKey](#) representa una clave de ordenación para un `String` y un `Collator` dados. Comparar dos objetos `CollationKey` envuelve una amplica comparación de claves cortas que es más rápido que comparar objetos `String` con el método `Collator.compare`. Sin embargo, generar objetos `CollationKey` también necesita tiempo. Por lo tanto, si un `String` sólo va a ser comparado una vez, `Collator.compare` ofrece mejor rendimiento.

En el siguiente ejemplo, utilizamos un objeto `CollationKey` para ordenar un array de palabras. El código completo de este ejemplo está en el fichero llamado [KeysDemo.java](#).

Creamos un array de objetos `CollationKey` en el método `main`. Para crear un `CollationKey`, se llama al método `getCollationKey` sobre un objeto `Collator`. No se pueden comparar dos objetos `CollationKey` que no sean originales del mismo objeto `Collator`. Aquí podemos ver el método `main`:

```
static public void main(String[] args) {

    Collator enUSCollator = Collator.getInstance(new Locale("en","US"));

    String [] words = {
        "peach",
        "apricot",
        "grape",
        "lemon"
    };

    CollationKey[] keys = new CollationKey[words.length];

    for (int k = 0; k < keys.length; k ++) {
        keys[k] = enUSCollator.getCollationKey(words[k]);
    }

    sortArray(keys);
    printArray(keys);
}
```

El método `sortArray` llama al método `CollationKey.compareTo`. Este método devuelve un entero menor que, igual que, o mayor que cero si el objeto `keys[i]` es menor que, igual que o mayor que el objeto `keys[j]`. Observa que hemos comparado los objetos `CollationKey`, no los objetos `String` del array original de

palabras. Aquí puedes ver el código del método `sortArray`:

```
public static void sortArray(CollationKey[] keys) {  
  
    CollationKey tmp;  
  
    for (int i = 0; i < keys.length; i++) {  
        for (int j = i + 1; j < keys.length; j++) {  
            // Compare the keys  
            if( keys[i].compareTo( keys[j] ) > 0 ) {  
                // Swap keys[i] and keys[j]  
                tmp = keys[i];  
                keys[i] = keys[j];  
                keys[j] = tmp;  
            }  
        }  
    }  
}
```

Hemos ordenado un array de objetos `CollationKey`, pero nuestro objetivo original era ordenar un array de objetos `String`. Para recuperar el `String` que representa a cada objeto `CollationKey`, llamamos al método `getSourceString` en el método `displayWords`:

```
static void displayWords(CollationKey[] keys) {  
  
    for (int i = 0; i < keys.length; i++) {  
        System.out.println(keys[i].getSourceString() + " ");  
    }  
}
```

El método `displayWords` imprime las siguientes líneas:

```
apricot  
grape  
lemon  
peach
```

Detectar Límites de Texto

Las aplicaciones que manipulan texto necesitan localizar límites dentro del propio texto. Por ejemplo, consideremos algunas de las funciones comunes en un procesador de textos: iluminar un caracter, cortar una palabra, mover el cursor al párrafo siguiente, y cortar las palabras al final de la línea. Para realizar cada una de estas funciones, el procesador de textos debe poder detectar los límites lógicos del texto. Afortunadamente, no tenemos que escribir nuestras propias rutinas para realizar análisis de límites. En su lugar, podemos aprovecharnos de los métodos proporcionados por la clase [BreakIterator](#).

[Sobre la clase BreakIterator](#)

Esta sección explica la ejemplarización de métodos y los cursores imaginarios de la clase `BreakIterator`.

[Límite de Caracter](#)

En esta sección aprenderás la diferencia entre los caracteres de usuario y los caracteres Unicode, y como localizar los caracteres de usuario con un `BreakIterator`.

[Límite de Palabra](#)

Si la aplicación necesita seleccionar o localizar palabras dentro del texto, encontraremos útil el uso de un `BreakIterator`.

[Límites de Párrafo](#)

Determinar los límites de párrafo puede ser problemático, debido a la utilización ambigua de terminadores de sentencias en muchos idiomas escritos. Esta sección examina algunos de los problemas que podremos encontrarnos, y como tratarlos con la clase `BreakIterator`.

[Límite de Línea](#)

Esta sección describe como localizar las rupturas de línea potenciales en una cadena de texto con un `BreakIterator`.

Sobre la clase BreakIterator

La clase BreakIterator es sensible a la Localidad, porque los límites de texto pueden variar con el idioma. Por ejemplo, la reglas para ruptura de líneas no son las mismas para todos los idiomas. Para determinar las Localidades soportadas por la clase BreakIterator se llama al método `getAvailableLocales`:

```
Locale[] locales = BreakIterator.getAvailableLocales();
```

Se pueden analizar cuatro tipos diferentes de límites con la clase BreakIterator: caracter, palabra, párrafo y ruptura potencial de línea. Cuande se ejemplariza un BreakIterator, se debe llamar al método de creacción apropiado:

- `getCharacterInstance`
- `getWordInstance`
- `getSentenceInstance`
- `getLineInstance`

Cada ejemplar de BreakIterator sólo puede detectar un tipo de límite. Si se quiere localizar el límite de caracter y de palabra, por ejemplo, se necesitará crear dos ejemplares separados.

Un BreakIterator tiene un cursor imaginario que apunta al límite actual en una cadena de texto. Se puede mover este cursor dentro del texto con los métodos `previous` y `next`. Por ejemplo, si hemos creado un BreakIterator con `getWordInstance`, cada vez que llamemos al método `next` el cursor de moverá al siguiente límite de palabra dentro del texto. Los métodos de movimiento del cursor devuelven un entero indicando la posición del límite. Esta posición es el índice del caracter de la cadena de texto que sigue al límite. Al igual que los Strings indexados, los límites están basados en cero. El primer límite es el 0, y el último límite es la longitud de la cadena.

Se debería utilizar la clase BreakIterator sólo con texto natural. No se debe utilizar con lenguajes de programación.

En las siguientes secciones, proporcionaremos ejemplos de cada tipo de límite. Los ejemplos de código se han extraído de un fichero llamado [BreakIteratorDemo.java](#).

Límite de Caracter

Se necesitará localizar los límites de carácter si nuestra aplicación permite al usuario final seleccionar caracteres individuales. o mover el cursor a través del texto carácter a carácter. Para crear un `BreakIterator` que localice los límites de carácter se llama al método `getCharacterInstance`:

```
BreakIterator characterIterator =  
    BreakIterator.getCharacterInstance(currentLocale);
```

Este tipo de `BreakIterator` detecta límites entre caracteres de usuario, no sólo caracteres Unicode. Los caracteres de usuario varían con el idioma, pero la clase `BreakIterator` puede reconocer estas diferencias porque es sensible a la Localidad. Un carácter de usuario podría estar compuesto por uno o más caracteres Unicode. Por ejemplo, el carácter de usuario ü podría componerse combinando los caracteres Unicode `'\u0075'` (u) `'\u00a8'` (¨). Sin embargo, este no es el mejor ejemplo, porque el carácter ü también podría ser representado por un sólo carácter Unicode `'\u00fc'`. Veremos el idioma Árabe para un ejemplo más realista.

En Árabe, la palabra para casa es:



Aunque esta palabra contiene tres caracteres de usuario, está compuesta por seis caracteres Unicode:

```
String house = "\u0628" + "\u064e" + "\u064a" +  
    "\u0652" + "\u067a" + "\u064f";
```

Los caracteres Unicode de las posiciones 1,3 y 5 en la cadena `house` son diacríticos. En Árabe los diacríticos son necesarios, porque pueden alterar el significado de las palabras. Los diacríticos en nuestro ejemplo son caracteres no blancos ya que aparecen sobre los caracteres base. En un procesador de textos árabe, no podemos mover el cursor sobre cada carácter Unicode de la cadena. En su lugar, debemos movernos sobre cada carácter de usuario, que podría estar compuesto por más de un carácter Unicode. Por lo tanto, debemos utilizar un `BreakIterator` para scanear los caracteres de usuario en la cadena.

En nuestro ejemplo, [BreakIteratorDemo.java](#), hemos creado un `BreakIterator` para scanear caracteres árabicos. Luego pasamos este `BreakIterator`, junto con el objeto `String` creado anteriormente, al método llamado `listPositions`:

```
BreakIterator arCharIterator =  
    BreakIterator.getCharacterInstance(new Locale ("ar", "SA"));
```

```
listPositions (house,arCharIterator);
```

El método `listPositions` utiliza un `BreakIterator` para localizar los límites de caracter dentro de la cadena. Observa que asignamos un string particular al `BreakIterator` con el método `setText`. Recuperamos el primer límite de caracter con el método `first`, luego llamamos al método `next` hasta que se devuelva la constante `BreakIterator.DONE`. Aquí podemos ver el código de esta rutina:

```
static void listPositions(String target, BreakIterator iterator) {  
  
    iterator.setText(target);  
    int boundary = iterator.first();  
  
    while (boundary != BreakIterator.DONE) {  
        System.out.println (boundary);  
        boundary = iterator.next();  
    }  
}
```

El método `listPositions` imprime las siguientes posiciones de límites para los caracteres de usuario de la cadena `house`. Las posiciones de los diacríticos (1, 3, 5) no se listan:

```
0  
2  
4  
6
```

Límite de Palabra

Se invoca al método `getWordIterator` para ejemplarizar un `BreakIterator` que detecte límites de palabra:

```
BreakIterator wordIterator =  
    BreakIterator.getWordInstance(currentLocale);
```

Querremos crear un `BreakIterator` como éste cuando nuestras aplicaciones necesiten realizar operaciones con palabras individuales. Estas operaciones podrían ser las funciones comunes de los procesadores de textos como seleccionar, cortar, pegar y copiar. O nuestras aplicaciones podrían buscar palabras, y para hacer esto necesitan poder distinguir entre palabras completas.

Cuando se realizan análisis de límites de palabra, un `BreakIterator` diferencia entre las palabras y los caracteres que no forman parte de las palabras. Estos caracteres, que incluyen espacios, tabuladores, marcas de puntuación, y algunos símbolos, tienen límites de palabras en ámbos lados.

En el siguiente ejemplo, extraído del programa [BreakIteratorDemo.java](#), queremos marcar los límites de palabras en algún texto. Primero creamos el `BreakIterator` y luego llamamos a un método que hemos escrito llamado `markBoundaries`:

```
Locale currentLocale = new Locale ("en","US");  
  
BreakIterator wordIterator =  
    BreakIterator.getWordInstance(currentLocale);  
  
String someText = "She stopped.  " +  
    "She said, \"Hello there,\" and then went on.";   
  
markBoundaries(someText, wordIterator);
```

El propósito de este método es marcar los límites de palabras en un string con un caracter ('^'). Cada vez que `BreakIterator` detecta un límite palabra, insertamos este caracter en el buffer `markers`. Scaneamos el string en un bucle, llamando al método `next` hasta que devuelva `BreakIterator.DONE`. Aquí tenemos el código de la rutina `markBoundaries`:

```
static void markBoundaries(String target, BreakIterator iterator) {  
  
    StringBuffer markers = new StringBuffer();  
    markers.setLength(target.length() + 1);  
    for (int k = 0; k < markers.length(); k++) {  
        markers.setCharAt(k, ' ');  
    }  
}
```

```

iterator.setText(target);
int boundary = iterator.first();

while (boundary != BreakIterator.DONE) {
    markers.setCharAt(boundary, '^');
    boundary = iterator.next();
}

System.out.println(target);
System.out.println(markers);
}

```

El método `markBoundaries` imprime el string `target` y el buffer `markers`. Obseva donde ocurren los caracteres ('^') en relación con las marcas de puntuación y los espacios:

```

She stopped.  She said, "Hello there," and then went on.
^  ^^      ^^ ^  ^^  ^^^^  ^^  ^^^^  ^^  ^^  ^^  ^

```

El `BreakIterator` hace sencilla la selección de palabras dentro de un texto. No tenemos que escribir nuestras propias rutinas para manejar las reglas de puntuación de los distintos idiomas, porque la clase `BreakIterator` lo hace por nosotros. Aquí podemos ver una subrutina que extrae e imprime las palabras de una cadena dada:

```

static void extractWords(String target, BreakIterator wordIterator) {

    wordIterator.setText(target);
    int start = wordIterator.first();
    int end = wordIterator.next();

    while (end != BreakIterator.DONE) {
        String word = target.substring(start,end);
        if (Character.isLetterOrDigit(word.charAt(0))) {
            System.out.println(word);
        }
        start = end;
        end = wordIterator.next();
    }
}

```

En nuestro ejemplo, llamamos a `extractWords`, le pasamos la misma cadena que en el ejemplo anterior. El método `extractWords` imprimirá la siguiente lista de palabras:

```

She

```


stopped
She
said
Hello
there
and
then
went
on.

Ozito

Límite de Sentencia

En muchos idiomas el terminador de sentencia es un punto. En Inglés, también se utiliza un punto para especificar el separador decimal, para indicar una marca de elipsis, y para terminar abreviaturas. Cómo el punto tiene más de un propósito, no podemos determinar el límite de sentencia con total seguridad.

Primero, echemos un vistazo a un caso en el que si funciona la detección de límite de sentencia. Empezaremos creando un `BreakIterator` con el método `getSentenceInstance`:

```
BreakIterator sentenceIterator =  
    BreakIterator.getSentenceInstance(currentLocale);
```

Para demostrar los límites de sentencias, utilizaremos el método [markBoundaries](#), que se explicó en la sección anterior. El método `markBoundaries` imprime caracteres ('^') en un string para indicar las posiciones de los límites. En el siguiente ejemplo, los límites de sentencias están definidos apropiadamente:

```
She stopped.  She said, "Hello there," and then went on.  
^            ^                                     ^
```

También se pueden localizar los límites de sentencias que terminen en interrogaciones o puntos de exclamación.

```
He's vanished!  What will we do?  It's up to us.  
^              ^                  ^          ^
```

Utilizar el punto como separador decimal no provoca ningún error:

```
Please add 1.5 liters to the tank.  
^                                     ^
```

Una marca de elipsis (puntos suspensivos) indica la omisión de texto dentro de un pasaje entrecomillado. En el siguiente ejemplo los puntos suspensivos generan límites de sentencia:

```
"No man is an island . . . every man . . . "  
^              ^ ^                  ^ ^ ^
```

Las abreviaturas también podrían provocar errores. Si el punto es seguido por un espacio en blanco y una letra mayúscula, el `BreakIterator` detecta un límite de sentencia:

```
My friend, Mr. Jones, has a new dog.  The dog's name is Spot.  
^              ^                  ^          ^
```

Límite de Línea

Las aplicaciones que formatean texto o realizan ruptura de líneas deben localizar las rupturas de líneas potenciales. Se pueden encontrar estas rupturas de línea, o límites, con un `BreakIterator` que haya sido creado con el método `getLineInstance`:

```
BreakIterator lineIterator =  
    BreakIterator.getLineInstance(currentLocale);
```

Este `BreakIterator` determina la posición en que se puede romper una línea para continuar en la siguiente línea. Las posiciones detectadas por el `BreakIterator` son rupturas de líneas potenciales. La ruptura de línea real mostrada en la pantalla podría no ser la misma.

En los siguientes ejemplos, utilizamos el método [markBoundaries](#) para ver los límites de línea detectados por un `BreakIterator`. Este método imprime marcas en los límites de línea de la cadena fuente.

De acuerdo al `BreakIterator`, un límite de línea ocurre después del final de una secuencia de caracteres blancos, (space, tab, newline). En el siguiente ejemplo, podemos romper la línea en cualquiera de los límites detectados:

```
She stopped.  She said, "Hello there," and then went on.  
^      ^      ^      ^      ^      ^      ^      ^      ^
```

Las rupturas de líneas potenciales también ocurren inmediatamente después de un guión:

```
There are twenty-four hours in a day.  
^      ^      ^      ^      ^      ^      ^      ^      ^
```

En el siguiente ejemplo, rompemos una cadena en líneas de la misma longitud con un método llamado `formatLines`. Utilizamos un `BreakIterator` para localizar las rupturas de líneas potenciales. Para romper una línea, ejecutamos un `System.out.println()` siempre que la longitud de la línea actual alcance el valor del parámetro `maxLength`. El método `formatLines` es corto, sencillo, y gracias al `BreakIterator`, independiente de la Localidad. Aquí puedes ver su código fuente:

```
static void formatLines(String target, int maxLength,  
                        Locale currentLocale) {  
  
    BreakIterator boundary = BreakIterator.getLineInstance(currentLocale);  
    boundary.setText(target);  
    int start = boundary.first();  
    int end = boundary.next();  
    int lineLength = 0;  
  
    while (end != BreakIterator.DONE) {
```

```

        String word = target.substring(start,end);
        lineLength = lineLength + word.length();
        if (lineLength >= maxLength) {
            System.out.println();
            lineLength = word.length();
        }
        System.out.print(word);
        start = end;
        end = boundary.next();
    }
}

```

En el programa [BreakIteratorDemo.java](#) llamamos a formatLines de esta forma:

```

String moreText = "She said, \"Hello there,\" and then " +
                  "went on down the street.  When she stopped " +
                  "to look at the fur coats in a shop window, " +
                  "her dog growled.  \"Sorry Jake,\" she said. " +
                  " \"I didn't know you would take it personally.\"";

formatLines(moreText, 30, currentLocale);

```

Aquí podemos ver la salida de la llamada a formatLines:

```

She said, "Hello there," and
then went on down the
street.  When she stopped to
look at the fur coats in a
shop window, her dog
growled.  "Sorry Jake," she
said.  "I didn't know you
would take it personally."

```

Convertir Texto no-Unicode

En el lenguaje de programación Java, los valores `char` representan caracteres Unicode. Unicode es una codificación de caracteres de 16 bits que soporta la mayoría de los idiomas del mundo. Podrás aprender más sobre el estándar Unicode en la web site de [Unicode Consortium](http://unicode.org).

Pocos editores de texto soportan actualmente texto Unicode. El editor de texto que hemos utilizado para escribir los códigos de ejemplo de esta lección sólo soporta caracteres ASCII, que están limitados a 7 bits. Para indicar un carácter Unicode que no puede ser representado en ASCII, como "ö," hemos utilizado la secuencia de escape `\u00F6`. Cada "d" en la secuencia de escape es un dígito hexadecimal. El siguiente ejemplo muestra cómo indicar el carácter "ö" con una secuencia de escape:

```
String str = "\u00F6";  
char c = '\u00F6';  
Character letter = new Character('\u00F6');
```

No tenemos que especificar al secuencia de escape Unicode para caracteres ASCII. Cuando se leen ficheros ASCII o ISO Latin-1, el entorno de ejecución de Java convierte automáticamente los caracteres a Unicode. Sin embargo, si se quiere convertir texto desde otras codificaciones a Unicode, debemos realizar las conversiones nosotros mismos.

En esta lección se explica el API que se utiliza para traducir texto no-Unicode a Unicode. Antes de utilizar estos APIs, deberíamos verificar que la codificación que queremos convertir está soportada. La lista de codificaciones soportadas no forma parte de la especificación del lenguaje de programación Java. Por lo tanto, podría variar con las plataformas. Para ver las codificaciones soportadas por el JDK, puedes ver la sección "Supported Encodings" en la documentación [Internationalization Overview](http://docs.oracle.com/javase/6/docs/api/java/util/Charset.html).

Las siguientes secciones describen dos técnicas para convertir texto no-Unicode:

[Bytes Codificados y Strings](#)

Esta sección muestra cómo convertir arrays de bytes no-Unicode en objetos Strings, y vice-versa.

[Streams de Caracteres y de Bytes](#)

En esta sección aprenderás como traducir entre streams de caracteres Unicode y streams de bytes con texto no-Unicode.

Bytes Codificados y Strings

Si el texto no-Unicode está almacenado en un array de bytes, se pueden convertir a Unicode con uno de los métodos constructores de String. Inversamente, se puede convertir un objeto String en un array de bytes de caracteres no-Unicode con el método `String.getBytes`. Cuando se llama a estos métodos se debe especificar el identificador de la codificación como uno de los parámetros.

En los siguientes ejemplos convertiremos caracteres entre UTF8 y Unicode. UTF8 es una forma binaria compacta para codificar caracteres Unicode de 16-bits en 8 bits. El código fuente de este ejemplo está en el fichero llamado [StringConverter.java](#).

Primero creamos un String que contiene los caracteres Unicode:

```
String original = new String("A" + "\u00ea" + "\u00f1"  
                             + "\u00fc" + "C");
```

Cuando imprimimos el String llamado original aparecerá como:

AêñüC

Para convertir el objeto String a UTF8, llamamos al método `getBytes` y le especificamos el identificador de codificación apropiado como un parámetro. El método `getBytes` devuelve un array de bytes en formato UTF8. Para crear un objeto String desde un array de bytes no-Unicode, llamamos al constructor de String con el parámetro de la codificación. El código que hace estas llamadas está encerrado en un bloque try, para el caso de que la codificación especificada no estuviera soportada:

```
try {  
    byte[] utf8Bytes = original.getBytes("UTF8");  
    byte[] defaultBytes = original.getBytes();  
  
    String roundTrip = new String(utf8Bytes, "UTF8");  
    System.out.println("roundTrip = " + roundTrip);  
  
    System.out.println();  
    printBytes(utf8Bytes, "utf8Bytes");  
    System.out.println();  
    printBytes(defaultBytes, "defaultBytes");  
}  
catch (UnsupportedEncodingException e) {  
    e.printStackTrace();  
}
```

Imprimimos los valores de los arrays `utf8Bytes` y `defaultBytes` para demostrar

un punto importante. La longitud del texto convertido podría no ser la misma que la longitud del texto original. Algunos caracteres Unicode se traducen en bytes sencillos, y otros en parejas de bytes. Nuestra rutina para mostrar los arrays de bytes es esta:

```
public static void printBytes(byte[] array, String name) {
    for (int k = 0; k < array.length; k++) {
        System.out.println(name + "[" + k + "] = " + "0x" +
            UnicodeFormatter.byteToHex(array[k]));
    }
}
```

Aquí está la salida de los métodos printBytes. Observa que sólo el primer y último caracteres la "A" y la "C", son los mismos en los dos arrays:

```
utf8Bytes[0] = 0x41
utf8Bytes[1] = 0xc3
utf8Bytes[2] = 0xaa
utf8Bytes[3] = 0xc3
utf8Bytes[4] = 0xb1
utf8Bytes[5] = 0xc3
utf8Bytes[6] = 0xbc
utf8Bytes[7] = 0x43
```

```
defaultBytes[0] = 0x41
defaultBytes[1] = 0xea
defaultBytes[2] = 0xf1
defaultBytes[3] = 0xfc
defaultBytes[4] = 0x43
```


Streams de Caracteres y de Bytes

El paquete `java.io` proporciona clases que permiten conversiones entre streams de caracteres y streams de bytes de texto no-Unicode. Con la clase [InputStreamReader](#), se pueden convertir streams de bytes a streams de caracteres. Se utiliza la clase [OutputStreamWriter](#) para traducir streams de caracteres a streams de bytes.

Cuando se crean objetos `InputStreamReader` y `OutputStreamWriter`, se debe especificar la codificación que se quiere convertir. Por ejemplo, si queremos traducir un fichero de texto en formato UTF8 a Unicode, deberíamos crear un `InputStreamReader` de la siguiente forma:

```
FileInputStream fis = new FileInputStream("output.txt");
InputStreamReader isr = new InputStreamReader(fis, "UTF8");
```

Si se omite el identificador de codificación, `InputStreamReader` y `OutputStreamWriter` utilizarán la codificación por defecto. Al igual que la lista de codificaciones soportadas, la codificación por defecto puede variar con la plataforma Java. En la versión 1.1 del JDK, la codificación por defecto es 8859_1 (ISO-Latin-1). Este valor se selecciona en la propiedad del sistema `file.encoding`. Puedes determinar la codificación que utilizará un `InputStreamReader` o un `OutputStreamWriter` llamando al método `getEncoding`. En el siguiente ejemplo, llamamos al método para determinar que la codificación por defecto de nuestra plataforma sea 8859_1:

```
InputStreamReader defaultReader = new InputStreamReader(fis);
System.out.println(defaultReader.getEncoding());
```

Se especifica un `InputStream` cuando se crea un `InputStreamReader`, y un `OutputStream` cuando se construye un `OutputStreamWriter`. `InputStream` y `OutputStream` son superclases abstractas y todas sus entradas y salidas son streams de bytes, Esto permite realizar conversiones de cualquiera de los streams de bytes que pertenezcan a sus subclases. Por ejemplo, con un `InputStreamReader` se puede convertir texto no-Unicode desde un `FileInputStream` o un `PipedInputStream`, porque ambas son subclases de `InputStream`.

En el siguiente ejemplo. veremos como realizar conversiones con las clases `InputStreamReader` y `OutputStreamWriter`. El código fuente de este ejemplo lo puedes encontrar en el fichero [StreamConverter.java](#). En este ejemplo, convertimos una secuencia de caracteres Unicode desde un objeto `String` en un `FileOutputStream` de bytes codificado en UTF8. El método que realiza la conversión se llama `writeOutput`:

```
static void writeOutput(String str) {
```

```

try {
    FileOutputStream fos = new FileOutputStream("output.txt");
    Writer out = new OutputStreamWriter(fos, "UTF8");
    out.write(str);
    out.close();
}
catch (IOException e) {
    e.printStackTrace();
}
}

```

readInput, leemos los bytes codificados en UTF8 desde el fichero creado con el método **writeOutput**. Utilizamos un **InputStreamReader** para convertir los bytes de UTF8 a Unicode, y devolver el resultado en un **String**. Aquí puedes ver el método **readInput**:

```

static String readInput() {

    StringBuffer buffer = new StringBuffer();
    try {
        FileInputStream fis = new FileInputStream("output.txt");
        InputStreamReader isr = new InputStreamReader(fis, "UTF8");
        Reader in = new BufferedReader(isr);
        int ch;
        while ((ch = in.read()) > -1) {
            buffer.append((char)ch);
        }
        in.close();
        return buffer.toString();
    }
    catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

```

En el método **main** de nuestro programa ejemplo, llamamos al método **writeOutput** para crear un fichero de bytes codificado en UTF. Luego leemos el mismo fichero, convirtiendo de nuevo los bytes a Unicode. Aquí puedes ver el código del método **main**:

```

public static void main(String[] args) {

    String jaString =
        new String("\u65e5\u672c\u8a9e\u6587\u5b57\u5217");
}

```

```
writeOutput(jaString);  
String inputString = readInput();  
String displayString = jaString + " " + inputString;  
new ShowString(displayString, "Conversion Demo");  
}
```

La cadena original (jaString) debería ser idéntica a la recién creada (inputString). Para ver si las dos cadenas son iguales, las concatenamos y las mostramos con un objeto ShowString. La clase ShowString muestra un string con el método Graphics.drawString. El código fuente de esta clases está en el fichero [ShowString.java](#). Cuando ejemplarizamos un ShowString en nuestro programa ejemplo, aparecen las siguientes ventanas. La repetición de los caracteres mostrados verifica que las dos cadenas son idénticas.



Una Lista para Internacionalizar un Programa Existente

Muchos programas no son internacionalizados cuando se escriben por primera vez. Estos programas podrían haber empezado como prototipos, o quizás no se hubiera pensado en su distribución internacional. Si se necesita internacionalizar un programa existente, se deberían realizar las siguientes tareas:

- Identificar los datos dependientes de la cultura.

Los mensajes de texto son la forma más obvia de datos que varían con la cultura, porque deben ser traducidos. Sin embargo, hay otros tipos de datos que podrían variar con la región o el idioma. La siguiente lista contiene ejemplos de datos dependientes de la cultura:

- mensajes
- etiquetas de componentes GUI
- ayuda online
- sonidos
- colores
- graficos
- iconos
- fechas
- horas
- números
- monedas
- medidas
- números de teléfono
- honores y títulos
- direcciones
- distribuciones de página

Para más información puedes ver [Datos sensibles a la Cultura](#)

- Aislar texto traducible en fardos de recursos.

La traducción es costosa. Se puede ayudar a reducir los costes aislando el texto que debe ser traducido en objetos ResourceBundle. El texto traducible incluye mensajes de estado, mensajes de error, entradas de diario, y etiquetas de componentes GUI. Este texto está codificado dentro de los programas que no han sido internacionalizados. Se necesitará localizar las ocurrencias del texto escrito que se muestra a los usuarios finales. Por ejemplo, se necesitará limpiar código como éste:

```
String buttonLabel = "OK";  
...  
Button okButton = new Button(buttonLabel);
```

Puedes ver [Aislar Objetos específicos de la Localidad en un ResourceBundle](#).

- Tratar con mensajes compuestos

Los mensajes compuestos contienen datos variables. En el mensaje, "El disco contiene 1100 ficheros," el entero 1100 podría variar. Este mensaje es difícil de traducir porque la

posición del entero en la sentencia no es la misma en todos los idiomas. El siguiente mensaje no es traducible porque el orden de los elementos de la sentencia está codificado por concatenación:

```
Integer fileCount;
...
String diskStatus = "The disk contains " + fileCount.toString() + " files.";
```

Siempre que sea posible, se debería evitar construir mensajes compuestos porque son difíciles de traducir. Sin embargo, si la aplicación necesita mensajes compuestos se pueden manejar con las técnicas descritas en [Formatear Mensajes](#).

- Formatear Números y Monedas

Si la aplicación muestra números y monedas, necesitaremos formatearlas de una forma independiente de la Localidad. El siguiente código todavía no está internacionalizado porque no muestra correctamente los números en todos los países:

```
Double amount;
TextField amountField;
...
String displayAmount = amount.toString();
amountField.setText(displayAmount);
```

Necesitaremos reemplazar el código anterior con una rutina que formatee correctamente los números. El lenguaje de programación Java proporciona varias clases que formatean números y monedas. Estas clases se explicaron en la sección, [Formatear Números y Moneda](#).

- Formatear Fechas y Horas

La fecha y la hora varían con la región y el idioma, Si el código contiene sentencias como la siguiente, necesitaremos cambiarlas:

```
Date currentDate = new Date();
TextField dateField;
...
String dateString = currentDate.toString();
dateField.setText(dateString);
```

Si utilizamos las clases de formateo de fechas nuestra aplicación podrá mostrar la fecha y la hora de la forma correcta en todo el mundo. Para ejemplos e instrucciones, puedes ver [Formatear Fechas y Horas](#).

- Manejar mensajes de Excepción

Deberíamos evitar a nuestros usuarios finales los mensajes de excepción codificados en nuestro propio idioma. La sección [Trabajar con Excepciones](#) muestra como atajar este problema.

- Comparación de Cadenas

Cuando se ordena o se busca texto, se necesita comparar cadenas. Si el texto se muestra, no deberíamos utilizar los métodos de comparación de la clase String. Un programa que no haya sido internacionalizado podría comparar las cadenas de esta forma:

```
String target;
String candidate;
...
if (target.equals(candidate)) {
...
if (target.compareTo(candidate) < 0) {
...

```

Los métodos `String.equals` y `String.compareTo` realizan comparaciones binarias, que no son efectivas cuando se ordena o se busca en algunos idiomas. En su lugar, deberíamos utilizar la clase `Collator`, que se describe en la sección [Comparar Strings](#).

- Utilizar Atributos de Caracteres Unicode

Los desarrolladores acostumbrados a programar en otros lenguajes podrían determinar los atributos de un carácter comparándolo con un carácter constante. Por ejemplo, podrían escribir código como éste:

```
char ch;
...
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {
    // ch is a letter
...
if (ch >= '0' && ch <= '9') {
    // ch is a digit
...
if ((ch == ' ') || (ch == '\n') || (ch == '\t')) {
    // ch is a whitespace

```

Este tipo de código no funciona en todos los idiomas. Deberíamos reemplazar esta comparación de caracteres con llamadas a los métodos proporcionados por la clase [Character](#). Por ejemplo, podríamos reemplazar el código anterior con las siguientes sentencias:

```
char ch;
...
if (Character.isLetter(ch)) {
...
if (Character.isDigit(ch)) {
...
if (Character.isSpaceChar(ch)) {

```

- Convertir Texto No-Unicode

Los caracteres en el lenguaje de programación Java están codificados en Unicode. Si nuestras aplicaciones manejan texto no-Unicode, debemos traducirlo a Unicode. Para más información, puedes ver [Convertir texto No-Unicode](#).

- Reservar espacio suficiente en el GUI

Normalmente el texto se expande cuando se traduce del inglés a otros idiomas. También la anchura de los caracteres no es la misma para todos los juegos de caracteres y tipos de fuentes. Debemos asegurarnos de diseñar el GUI con espacio de sobra.

El Formato de Ficheros Java (JAR)

Versión original de Alan Sommerer

El formato de ficheros de 'Archivos Java™' te permite empaquetar varios ficheros en un sólo archivo. Típicamente un fichero JAR contendrá los ficheros de clases y los recursos auxiliares asociados con los applets y aplicaciones. Estos recursos auxiliares podrían incluir, por ejemplo, ficheros de imagen y sonido que sean utilizados por un applet.

Nota: El formato de fichero JAR fue presentado en la versión 1.1 del JDK, y la versión 1.2 incluye muchas mejoras en la funcionalidad de los ficheros JAR. A menos que se diga lo contrario las características cubiertas en esta lección pertenecen a las dos versiones. Si una característica o ejemplo pertenece sólo a una de las versiones, se informará de ello con una anotación como esta "versión 1.2."

El formato de ficheros JAR proporciona muchos beneficios:

- **Seguridad:** Puedes firmar digitalmente el contenido de un fichero JAR. Los usuarios que reconozcan tu firma pueden permitir a tu software privilegios de seguridad que de otro modo no tendría.
- **Disminuir el tiempo de descarga:** Si tus applets están empaquetados en un fichero JAR, los ficheros de clases y los recursos asociados pueden ser descargados por el navegador en una sólo transacción HTTP sin necesidad de abrir una nueva conexión para cada fichero.
- **Compresión:** El formato JAR permite comprimir tus ficheros para ahorrar espacio.
- **Empaquetado por extensiones (versión 1.2):** El marco de trabajo de las extensiones proporciona un significado por el cual puedes añadir funcionalidad al corazón de la plataforma Java. Y el formato JAR define el empaquetado por extensiones. [Java 3D™](#) y [JavaMail](#) son ejemplos de extensiones desarrolladas por Sun™. Mediante el uso del formato JAR también puedes convertir tu software en extensiones.
- **Empaquetado sellado (versión 1.2):** Los paquetes almacenados en ficheros JAR pueden ser sellados opcionalmente para que el paquete puede reforzar su consistencia. El sellado de un paquete dentro de un fichero JAR significa que todas las clases definidas en ese paquete deben encontrarse dentro del mismo fichero JAR.
- **Empaquetado versionado (versión 1.2):** Un fichero JAR puede contener datos sobre los ficheros que contiene, como información sobre el vendedor o la versión.
- **Portabilidad:** El mecanismos para manejar los ficheros JAR son una parte estándar del corazón del API de la plataforma Java.

[Utilizar ficheros JAR: básico](#)

Muestra cómo realizar las operaciones básicas sobre ficheros JAR, y cómo ejecutar software que está contenido en estos ficheros. Esta lección también presenta el concepto de 'manifiesto' de los ficheros JAR, que juega un papel importante en las funcionalidades avanzadas del formato JAR.

[Firmar y Autenticar ficheros JAR](#)

Muestra como utilizar las herramientas del JDK™ para firmar digitalmente ficheros JAR y verificar las firmas de ficheros JAR firmados.

Ozito

Utilizar Ficheros JAR: Básico

Esta lección te mostrará cómo realizar las operaciones básicas con ficheros JAR. Los ficheros JAR están empaquetados con el formato ZIP, por eso puedes utilizarlos en tareas del "estilo ZIP" como compresión de datos, archivado, descompresión y desempaquetado de archivos.

De echo, estos están entre los usos más comunes de los ficheros JAR, y se pueden obtener muchos beneficios de los ficheros JAR utilizando sólo estas características básicas. Por ejemplo, empaquetar applets multi-clases en un sólo fichero JAR puede reducir drásticamente el tiempo de descarga del applet.

Incluso si quieres aprovecharte de las funcionalidades avanzadas del formato JAR como la firma electrónica, primeros necesitarás familiarizarte con las operaciones fundamentales.

Para realizar las tareas básicas con ficheros JAR, debes utiliziar la Herramienta de Archivos Java™ proporcionada como parte del JDK. Como esta herramienta es invocada utilizando el comando jar, por conveniencia la llamaremos "herramienta Jar".

Esta lección te enseñará como utilizar la herramienta Jar, con ejemplos de cada una de las características básicas:

- [Crear un fichero JAR](#)
- [Ver el contenido de un fichero JAR](#)
- [Extraer el contenido de un fichero JAR](#)
- [Modificar un fichero de Manifiesto](#)

Además, esta lección contiene una sección [Ejecutar Software empaquetado en ficheros JAR](#) que te enseña como llamar a los applets y aplicaciones que están empaquetados en ficheros JAR.

Como sinopsis de los tópicos cubiertos en esta lección, la siguiente tabla resume las operaciones más comunes con ficheros JAR:

Operación	Comando
Para crear un fichero JAR	jar cf jar-file input-file(s)
Para ver el contenido de un fichero JAR	jar tf jar-file
Para extraer el contenido de un fichero JAR	jar xf jar-file
Para extraer ficherso específicos de un fichero JAR	jar xf jar-file archived-file(s)

Para ejecutar una aplicación empaquetada en un fichero JAR	<code>jre -cp app.jar MainClass</code>
Para llamar a un applet empaquetado en un fichero JAR	<pre><applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height> </applet></pre>

Algunas de las características más avanzadas ofrecidas por el formato JAR, como el sellado de paquetes y la firma electrónica se han hecho posibles gracias al manifiesto de los ficheros JAR, un fichero especial que contienen los ficheros JAR. En la sección final de esta lección, [Comprender el Manifiesto](#), encontrarás información básica sobre la estructura y utilización del fichero de manifiesto.

Referencias adicionales

La documentación de JDK incluye páginas sobre la herramienta Jar:

- [Jar tool reference for Windows platform](#)
- [Jar tool reference for Solaris platform](#)

Crear un fichero JAR

El comando de la herramienta Jar

El formato básico del comando para crear un fichero JAR es:

```
jar cf fichero-jar fichero(s)-de entrada
```

Echemos un vistazo a las opciones y argumentos utilizados en este comando:

- La opción **c** indica que quieres crear un fichero JAR.
- La opción **f** indica que quieres que la salida vaya a un fichero en vez de a stdout.
- **fichero-file** es el nombre que quieres para el fichero JAR resultante. Puedes utilizar cualquier nombre de fichero. Por convención, a los ficheros JAR se les da la extensión **.jar**, aunque no es obligatorio.
- El argumento **fichero(s)-de entrada** es una lista delimitada por espacios de uno o más ficheros que deben ser situados dentro de tu fichero JAR. Este argumento puede tener símbolo del comodín *****. Si alguno de los fichero(s)-de entrada, es un directorio, el contenido de dicho directorio se añadirá al fichero JAR recursivamente.

Las opciones **c** y **f** pueden aparecer en cualquier orden, pero no debe existir ningún espacio entre ellas.

Este comando generará un fichero JAR comprimido y lo situará en el directorio actual. El comando también generará un [fichero de manifiesto](#), por defecto.

META-INF/MANIFEST.MF, para el archivo JAR.

Podrías añadir cualquiera de estas opciones adicionales a las opciones **cf** del comando básico:

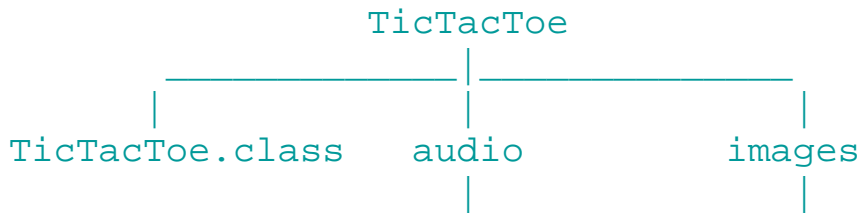
- **v** - produce un salida verbosa en stderr (en versión 1.1) o stdout (en versión 1.2) mientras se construye el fichero. La salida verbosa te dice el nombre de cada fichero añadido al fichero JAR.
- **O** - indica que no quieres que el fichero JAR sea comprimido.
- **M** - indica que no se debería producir el fichero de manifiesto por defecto.
- **m** - utilizada para incluir información de manifiesto desde un fichero de manifiesto existente. El formato utilizado por esta opción es:

```
jar cmf existing-manifest output-file input-file(s)
```

En la versión 1.1 el formato JAR sólo soporta nombres de ficherso ASCII. La versión 1.2 añade soporte para nombres codificados en UTF8.

Un ejemplo

Veamos un ejemplo. El JDK™ incluye una demo del applet TicTacToe. Esta demo contiene ficheros de clases, ficheros de audio e imágenes, todos almacenados en un directorio llamado TicTacToe que tiene esta estructura:



Los subdirectorios audio e images contienen ficheros GIF y de sonido utilizados por el applet.

Para empaquetar esta demo en un sólo fichero JAR llamado TicTacToe.jar, deberías ejecutar este comando desde el directorio TicTacToe:

```
jar cvf TicTacToe.jar TicTacToe.class audio images
```

Los argumentos audio e images representan directorios, por eso la herramienta JAR los situará recursivamente a ellos y sus contenidos en el fichero JAR. El fichero JAR generado TicTacToe.jar estará situado en el directorio TicTacToe. Como el comando utiliza la opción v para salida verbosa, podrás ver esta salida:

```
adding: TicTacToe.class (in=3825) (out=2222) (deflated 41%)
adding: audio/ (in=0) (out=0) (stored 0%)
adding: audio/beep.au (in=4032) (out=3572) (deflated 11%)
adding: audio/ding.au (in=2566) (out=2055) (deflated 19%)
adding: audio/return.au (in=6558) (out=4401) (deflated 32%)
adding: audio/yahoo1.au (in=7834) (out=6985) (deflated 10%)
adding: audio/yahoo2.au (in=7463) (out=4607) (deflated 38%)
adding: images/ (in=0) (out=0) (stored 0%)
adding: images/cross.gif (in=157) (out=160) (deflated -1%)
adding: images/not.gif (in=158) (out=161) (deflated -1%)
```

Puedes ver desde esta salida que el fichero JAR TicTacToe.jar está comprimido. La herramienta JAR comprime los ficheros por defecto. Puedes desactivar la compresión utilizando la opción O (cero), con lo que el comando se parecería a esto:

```
jar cvf0 TicTacToe.jar TicTacToe.class audio images
```

Podrías querer evitar la compresión, por ejemplo, para incrementar la velocidad a la que el fichero JAR podría ser cargado por un navegador. Los ficheros JAR sin comprimir generalmente pueden cargarse más rápido que los comprimidos porque no necesitan descomprimir los ficheros durante la descarga.

La herramienta JAR aceptará argumentos que utilicen el símbolo de comodín *. Como no existen ficheros en el directorio TicTacToe que no se quieran añadir, podrías utilizar este comando alternativo para construir el fichero JAR:

```
jar cvf TicTacToe.jar *
```

Una nota final: aunque la salida verbosa no lo indique, la herramienta Jar añade automáticamente un [fichero de manifiesto](#) al archivo JAR en la localización META-INF/MANIFEST.MF.

Ozito

Ver el contenido de un fichero JAR

El comando de la herramienta Jar

El formato del comando básico para ver el contenido de un fichero JAR es:

```
jar tf fichero-jar
```

Echemos un vistazo a las opciones y argumentos utilizados en este comando:

- La opción **t** indica que quieres ver la tabla de contenidos del fichero JAR.
- La opción **f** indica que el fichero JAR que se quiere ver esta especificado en la línea de comandos. Sin la opción **f**, la herramienta Jar esperaría un fichero en stdin.
- El argumento **fichero-jar** es el nombre del fichero (o path) del fichero JAR cuyo contenido se quiere visualizar.

Las opciones **t** y **f** pueden aparecer en cualquier orden, pero no puede existir ningún espacio entre ellas.

Este comando mostrará la tabla de contenidos del fichero JAR en stdout.

Opcionalmente puedes utilizar la opción **verbose**, **v**, para producir información adicional sobre el tamaño de los ficheros, y las fechas de modificación.

Un Ejemplo

Utilicemos la herramienta Jar para listar el contenido del fichero **TicTacToe.jar** que fue creado en la [página anterior](#):

```
jar tf TicTacToe.jar
```

Este comando mostrará el contenido del fichero JAR en stdout:

```
META-INF/MANIFEST.MF
TicTacToe.class
audio/
audio/beep.au
audio/ding.au
audio/return.au
audio/yahoo1.au
audio/yahoo2.au
images/
images/cross.gif
images/not.gif
```

El fichero JAR contiene el fichero class de TicTacToe y los directorios de audio e images, como se esperaba. La salida también muestra que el fichero JAR contiene un [fichero de manifiesto](#), META-INF/MANIFEST.MF, que fue situado automáticamente en el archivo por la herramienta JAR.

Todos los paths se muestran con barras invertidas, sin importar la plataforma o el sistema operativo que estés utilizando. Los paths en un fichero JAR siempre son paths relativos; por ejemplo, nunca verás un path que empiece con C:.

La herramienta Jar mostrará información adicional si utilizas la opción v:

```
jar tvf TicTacToe.jar
```

La salida verbosa del fichero JAR TicTacToe se podría parecer a esto:

```
256 Mon Apr 20 10:50:28 PDT 1998 META-INF/MANIFEST.MF
3885 Mon Apr 20 10:49:50 PDT 1998 TicTacToe.class
  0 Wed Apr 15 16:39:32 PDT 1998 audio/
4032 Wed Apr 15 16:39:32 PDT 1998 audio/beep.au
2566 Wed Apr 15 16:39:32 PDT 1998 audio/ding.au
6558 Wed Apr 15 16:39:32 PDT 1998 audio/return.au
7834 Wed Apr 15 16:39:32 PDT 1998 audio/yahoo1.au
7463 Wed Apr 15 16:39:32 PDT 1998 audio/yahoo2.au
  0 Wed Apr 15 16:39:44 PDT 1998 images/
157 Wed Apr 15 16:39:44 PDT 1998 images/cross.gif
158 Wed Apr 15 16:39:44 PDT 1998 images/not.gif
```

Extraer el contenido de un fichero JAR

El comando de la herramienta Jar

El formato básico del comando para extraer el contenido de un fichero JAR es:

```
jar xf fichero-jar [fichero(s)-archivados]
```

Echemos un vistazo a las opciones y argumentos de este comando:

- La opción **x** indica que quieres extraer los ficheros de un archivo JAR.
- La opción **f** indica que el fichero JAR que se quiere ver esta especificado en la línea de comandos. Sin la opción **f**, la herramienta Jar esperaría un fichero en stdin.
- El argumento **fichero-jar** es el nombre de fichero (o path y nombre) del fichero JAR del que quieres extraer los ficheros.
- **fichero(s)-archivados** es un argumento opcional que consiste en un lista delimitada por espacios de los ficheros que se quieren extraer del archivo. Si este argumento no está presente, la herramienta Jar extraerá todos los ficheros del archivo.

Como es normal, el orden en que aparezcan las opciones **x** y **f** no importa, pero no debe existir ningún espacio entre ellas.

Cuando se extraen ficheros, la herramienta Jar copia los ficheros deseados y los escribe en el directorio actual, reproduciendo la estructura de directorios que los ficheros tenían en el archivo. El archivo JAR original no se modifica.

Precaución: Cuando se extraen ficheros, la herramienta Jar sobrescribe cualquier fichero que tenga el mismo nombre y path que los ficheros extraídos.

Un ejemplo

Extraigamos algunos ficheros del archivo JAR TicTacToe que utilizamos en las secciones anteriores. Recordemos que el contenido de TicTacToe.jar era:

```
META-INF/MANIFEST.MF
TicTacToe.class
audio/
audio/beep.au
audio/ding.au
audio/return.au
audio/yahoo1.au
audio/yahoo2.au
images/
images/cross.gif
```


`images/not.gif`

Supongamos que queremos extraer los ficheros `TicTacToe.class` y `cross.gif`. Para hacer esto, puedes utilizar este comando:

```
jar xf TicTacToe.jar TicTacToe.class images/cross.gif
```

Este comando hace dos cosas:

- Sitúa una copia de `TicTacToe.class` en el directorio actual.
- Crea el directorio `images`, si no existe, y sitúa una copia de `cross.gif` en él.

El fichero JAR original no se modifica.

Se pueden extraer todos los ficheros que se quieran de la misma forma. Cuando el comando no especifica qué ficheros extraer, la herramienta JAR extrae todos los ficheros del archivo. Por ejemplo, puedes extraer todos los ficheros del archivo `TicTacToe` utilizando este comando:

```
jar xf TicTacToe.jar
```

Ozito

Modificar un Fichero de Manifiesto

El comando de la herramienta Jar

La herramienta Jar te permite mezclar el contenido de un fichero de manifiesto pre-existente con el fichero de manifiesto por defecto que se genera cuando se crea un fichero JAR.

Podrías querer producir un fichero de manifiesto que no fuera por defecto, por ejemplo, añadiendo una cabecera de propósito especial que permita a tu fichero JAR realizar una función particular. Puedes ver ejemplos de algunas cabeceras de propósito especial en la sección [Entender el Manifiesto](#).

El comando básico tiene este formato:

```
jar cmf manifest jar-file input-file(s)
```

Echemos un vistazo a las opciones y argumentos utilizados en este comando:

- La opción **c** indica que quieres crear un fichero JAR.
- La opción **m** indica que quieres mezclar información de un fichero de manifiesto existente en el fichero de manifiesto del fichero JAR que estás creando.
- La opción **f** indica que quieres que la salida vaya a un fichero (el fichero Jar que estás creando), en vez de a stdout.
- **manifest** es el nombre (o path y nombre) del fichero de manifiesto existente cuyo contenido quieres incluir en el manifiesto del fichero JAR.
- **jar-file** es el nombre que quieres para el fichero JAR resultante.
- El argumento **input-file(s)** es una lista delimitada por espacios de los ficheros que quieres añadir a tu fichero JAR.

Las opciones **c**, **m**, y **f** pueden aparecer en cualquier orden, pero no debe existir ningún espacio entre ellas.

Un ejemplo

En la versión 1.2 de la plataforma Java™, los paquetes dentro de ficheros JAR pueden sellarse opcionalmente, lo que significa que todas las clases definidas en un paquete deben estar archivadas en el mismo fichero JAR. Por ejemplo, podrías querer sellar un paquete, para asegurar la consistencia de versiones entre las clases de tu software.

Un paquete puede sellarse añadiendo la cabecera **Sealed**:

```
Name: myCompany/myPackage/  
Sealed: true
```

Para almacenar la cabecera Sealed en el manifiesto de un fichero JAR, primero necesitas escribir un fichero de manifiesto con las cabeceras apropiadas. Realmente, el fichero que escribas no tiene que ser un fichero de manifiesto completo; puede contener sólo suficiente información para que la herramienta Jar sepa dónde y qué información mezclar dentro del fichero de manifiesto por defecto.

Supongamos, por ejemplo, que tu fichero JAR es para contener estos cuatro paquetes:

```
myCompany/firstPackage
myCompany/secondPackage
myCompany/thirdPackage
myCompany/fourthPackage
```

y que quieres sellar firstPackage y thirdPackage. Para hacer esto, deberías escribir un fichero de manifiesto parcial con este contenido:

```
Name: myCompany/firstPackage/
Sealed: true
```

```
Name: myCompany/thirdPackage/
Sealed: true
```

Luego supongamos que:

- que has llamado a tu manifiesto parcial myManifest
- el fichero JAR que quieres crear se llamará myJar.jar
- El directorio actual es el directorio padre de myCompany

Podrías crear el fichero JAR con este comando:

```
jar cmf myManifest myJar.jar myCompany
```

El aspecto resultante del fichero de manifiesto de myJar.jar dependerá de la versión del JDK que estés utilizando. En cualquier caso, la información de sellado, será incluida para firstPackage y thirdPackage.

Ejecutar software empaquetado en un fichero JAR

Ahora que has aprendido a crear ficheros JAR, ¿Cómo ejecutarás realmente el código que has empaquetado? Existen dos escenarios a considerar:

- Tu fichero JAR contiene un applet para ser ejecutado dentro de un navegador.
- Tu fichero JAR contiene una aplicación que es llamada desde la línea de comandos.

Ficheros JAR en los Applets

Para llamar a un applet desde una página HTML y ejecutarlo dentro de un navegador, necesitas utilizar la etiqueta APPLET. (Puedes ver [Escribir Applets](#) para mas información sobre los applets.) Si el applet está empaquetado en un fichero JAR, lo unico diferente que tienes que hacer es utilizar el parámetro ARCHIVE para especificar el path relativo del archivo JAR.

Cómo ejemplo, utilizemos (de nuevo!) el applet TicTacToe que viene con el JDK de Java™. La etiqueta APPLET en el fichero HTML que llama a este applet se parecería a esto (ignorando la etiqueta ALT para más claridad):

```
<applet code=TicTacToe.class
        width=120 height=120>
</applet>
```

Si el applet TicTacToe estuviera empaquetado en un fichero JAR llamado TicTacToe.jar, podrías modificar la etiqueta APPLET con la sencilla adición de un parámetro ARCHIVE:

```
<applet code=TicTacToe.class
        archive="TicTacToe.jar"
        width=120 height=120>
</applet>
```

El parámetro ARCHIVE especifica el path relativo para el fichero JAR que contiene TicTacToe.class. Este ejemplo asume que el fichero JAR y el fichero HTML están en el mismo directorio. Si no fuera así, deberías especificar el path del fichero JAR.

Ficheros JAR para Aplicaciones - *JDK 1.1*

Puedes ejecutar aplicaciones que están empaquetadas en ficheros JAR utilizando la herramienta jre del JDK 1.1:

```
jre -cp app.jar MainClass
```

La opción -cp añade app.jar al classpath del sistema. MainClass identifica la clase

dentro del fichero JAR que es el punto de entrada de la aplicación. (Recuerda que en una aplicación, una de las clases debe tener un método con la firma: `public static void main(String[] args)` que sirve como punto de entrada o de arranque de la aplicación.)

Ficheros JAR para Aplicaciones - sólo JDK 1.2

En la versión 1.2 del JDK, puedes ejecutar aplicaciones empaquetadas en ficheros JAR con el intérprete Java. El comando básico es:

```
java -jar jar-file
```

La bandera `-jar` le dice al intérprete que la aplicación está empaquetada en un fichero JAR.

Nota: la opción `-jar` no está disponible en intérpretes anteriores a la versión 1.2 del JDK.

Sin embargo, para que este comando funcione, el intérprete necesita saber qué clase dentro del fichero JAR es el punto de entrada de la aplicación.

Para hacer esto, debes añadir una cabecera `Main-Class` al [manifiesto](#) del fichero JAR. La cabecera tiene esta forma:

```
Main-Class: classname
```

donde `classname` es el nombre de la clase que es el punto de entrada de la aplicación.

Para crear un fichero JAR que tenga un manifiesto con la cabecera apropiada `Main-Class`, puedes utilizar la opción `m` como se describió en la sección [anterior](#). Primero deberás preparar una plantilla de manifiesto que consista en una sola línea con la cabecera `Main-Class` y el valor. Por ejemplo, si tu aplicación fuera la aplicación de una sola clase `HelloWorld`, el punto de entrada, por su puesto, sería la clase `HelloWorld`, y tu plantilla de manifiesto podría ser como esta línea:

```
Main-Class: HelloWorld
```

Asumiendo que tu plantilla se encuentra en un fichero llamado `template`, la podrías mezclar con el fichero de manifiesto del fichero JAR con un comando como este:

```
jar cmf template app.jar HelloWorld.class
```

Con tu fichero JAR preparado de esta forma, puedes ejecutar el programa `HelloWorld` desde la línea de comandos:

```
java -jar app.jar
```


Entender el Manifiesto

Los ficheros JAR pueden soportar un amplio rango de funcionalidades, incluyendo la firma electrónica, el control de versiones, el sellado de paquetes, las extensiones, etc. ¿Qué le da a los ficheros JAR la habilidad para ser tan versátiles? La respuesta se encuentra dentro del manifiesto de los ficheros JAR.

El manifiesto es un fichero especial que puede contener información sobre los otros ficheros empaquetados en un fichero JAR. Personalizar la información "meta" del manifiesto, te permite utilizar los ficheros JAR para una gran variedad de propósitos.

Antes de ver algunas de las formas en que puede ser modificado el manifiesto para permitir funcionalidades especiales a los ficheros JAR, echemos un vistazo a la línea base del manifiesto por defecto.

El Manifiesto por defecto

Siempre que creas un fichero JAR, automáticamente recibe un fichero de manifiesto por defecto. Sólo puede haber un fichero de manifiesto en un fichero JAR, y siempre debe tener el path:

```
META-INF/MANIFEST.MF
```

Cuando un fichero JAR es creado con la versión 1.2 del JDK, el manifiesto por defecto es muy sencillo. Aquí tienes todo su contenido:

```
Manifest-Version: 1.0  
Created-By: Manifest JDK1.2
```

Como puedes ver en este ejemplo, las entradas de un fichero de manifiesto tienen la forma de parejas: "cabecera:valor". El nombre de una cabecera está separado de su valor por dos puntos.

El manifiesto mostrado arriba es conforme a la versión 1.0 de la especificación de manifiesto y ha sido creado con la versión 1.2 del JDK. Estas son propiedades del propio manifiesto, pero también puede contener información sobre otros ficheros empaquetados en el archivo.

La información exacta grabada en el fichero de manifiesto depende del uso previsto del fichero JAR. El fichero de manifiesto por defecto no asume nada sobre la información que debería almacenar sobre otros ficheros, por eso sólo contiene datos sobre él mismo.

El formato del fichero de manifiesto por defecto cambia de la versión 1.1 a la versión 1.2 del JDK. Si creas un fichero JAR para el paquete, java.math, por ejemplo, el fichero de manifiesto por defecto del JDK 1.1 se parecería a esto:

Manifest-Version: 1.0

Name: java/math/BigDecimal.class
Digest-Algorithms: SHA MD5
SHA-Digest: TD1GZt8G1ldXY2p4o1SZPc5Rj64=
MD5-Digest: z6z8xPj2AW/Q9AkRSPF0cg==

Name: java/math/BigInteger.class
Digest-Algorithms: SHA MD5
SHA-Digest: oBmrvIkBnSxdNZzPh5iLyF0S+bE=
MD5-Digest: wFymhDKjNreNZ4AzDWWglQ==

Al contrario que el manifiesto del JDK 1.2 el del JDK 1.1 tiene entradas para cada uno de los ficheros contenidos en el archivo, incluyendo los paths de los ficheros y valores digest. Estos últimos valores son solo importantes con respecto a la firma de ficheros JAR. De echo, el por qué la información digest no está en el fichero de manifiesto del JDK 1.2 - es porque nunca la necesita. Para aprender más sobre la firma, puedes ver la lección [Firmar y Autenticar ficheros JAR](#).

Cabeceras de Manifiesto para Propósitos Especiales

Dependiendo del papel que quieres que juegue tu fichero JAR, podrías modificar el manifiesto por defecto. Si sólo estas interesado en las características "ZIP" del fichero JAR como la compresión o el archivado, no tendrás que preocuparte del fichero de manifiesto. Este fichero no juega ningún papel en estas situaciones.

La mayoría de los usos de los ficheros JAR que van más allá del simple archivado y compresión necesitan que alguna información especial sea almacenada en el fichero de manifiesto. Abajo tienes una breve descripción de las cabeceras requeridas para algunas funciones de propósito especial de los ficheros JAR:

- [Aplicaciones empaquetadas en ficheros JAR](#)
- [Descarga de extensiones](#)
- [Sellado de paquetes](#)
- [Versionado de paquetes](#)

Aplicaciones empaquetadas en ficheros JAR - **sólo versión 1.2**

Si tienes una aplicación en un fichero JAR, necesitas indicar de alguna forma que clase es el punto de entrada de las que se incluyen en el fichero JAR. (Recuerda que el punto de entrada es una clase que tenga un método con la firma:
public static void main(String[] args).)

Esta información se proporciona con la cabecera Main-Class, que tiene esta forma general:


```
Main-Class: classname
```

donde classname es el nombre de la clase que es el punto de entrada de la aplicación.

Descarga de Extensiones - **sólo versión 1.2**

La descarga de extensiones son ficheros JAR que son referenciados por el fichero de manifiesto de otros ficheros JAR. En una situación típica, un applet estaría empaquetado en un fichero JAR cuyo manifiesto referenciara a un fichero JAR (o a varios ficheros JAR) que servirán como una extensión para los propósitos del applet. Las extensiones pueden referenciarse unas a otras de la misma forma.

La descarga de extensiones se especifica en el campo de cabecera Class-Path en el fichero de manifiesto del applet, aplicación, u otra extensión. Una cabecera Class-Path se podría parecer a esto, por ejemplo:

```
Class-Path: servlet.jar infobus.jar acme/beans.jar
```

Con esta cabecera, las clases de los ficheros servlet.jar, infobus.jar, y acme/beans.jar servirán como extensiones para los propósitos del applet o aplicación. Las URLs en la cabecera Class-Path son relativas a la URL del fichero JAR del applet o de la aplicación.

Sellado de Paquetes - **sólo versión 1.2**

Los paquetes almacenados en ficheros JAR pueden ser sellados opcionalmente para que el paquete pueda reforzar su consistencia. El sellado de un paquete dentro de un fichero JAR significa que todas las clases definidas en ese paquete deben encontrarse dentro del mismo fichero JAR.

Un paquete puede sellarse añadiendo la cabecera Sealed:

```
Name: myCompany/myPackage/  
Sealed: true
```

Versionado de Paquetes - **sólo versión 1.2**

La pagina [Especificación de Versionado de Paquetes](#) define varias cabeceras de manifiesto para contener información del versionado. Un conjunto de dichas cabeceras puede ser asignado a cada paquete. Las cabeceras de versionado deberían aparecer directamente debajo de la cabecera name del paquete. Este ejemplo muestra las cabeceras de versionado:

```
Name: java/util/  
Specification-Title: "Java Utility Classes"  
Specification-Version: "1.2"  
Specification-Vendor: "Sun Microsystems Inc."  
Implementation-Title: "java.util"  
Implementation-Version: "build57"
```

Implementation-Vendor: "Sun Microsystems. Inc."

Información Adicional

La [Especificación](#) del formato de manifiesto es forma parte de la documentación on-line del JDK.

Ozito

Firmado y Verificación de Ficheros JAR

Esta lección te enseña como utilizar las herramientas proporcionadas por el JDK de Java™ para firmar y verificar ficheros JAR:

- [Firmar ficheros JAR](#) te muestra como utilizar las herramientas del JDK™ para firmar digitalmente tus ficheros JAR.
- [Verificar ficheros JAR firmados](#) te enseña como verificar ficheros JAR.

Si no estás familiarizado con la firma, podrás encontrar alguna información en la página [Entender la Firma y Verificación](#), donde encontrarás definiciones de términos relevantes, explicaciones de algunos beneficios proporcionados por la firma, y una guía del mecanismo de firma utilizado por la plataforma Java y su relación con los ficheros JAR.

Firmar ficheros JAR

El JDK de Java™ contiene las herramientas que necesitas para firmar ficheros JAR. Dependiendo de la versión del JDK que tengas, utilizarás:

- JDK 1.2 [La herramienta JAR de Firma y Verificación](#)
 - JDK 1.1 [La herramienta de Seguridad JAVA](#)
-

La Herramienta JAR de Firma y Verificación del JDK 1.2

Esta herramienta se invoca utilizando el comando `jarsigner`, por lo tanto, nos referiremos a ella como "Jarsigner".

Para firmar un fichero JAR, primero debes tener una clave privada. Las claves privadas y sus correspondientes claves públicas están almacenadas en bases de datos protegidas llamadas keystores. Un keystore puede contener las claves de muchos firmantes potenciales. Cada clave del keystore puede ser identificada por un alias que suele ser el nombre del firmante propietario de la clave. Por ejemplo, la clave perteneciente a Rita Jones, podría tener el alias "rita".

La forma básica para el comando de firma de un fichero JAR es

```
jarsigner jar-file alias
```

En este comando:

- `jar-file` Es el nombre del fichero JAR a firmar.
- `alias` es el alias que identifica la clave privada que se utilizará para firmar el fichero JAR, y su certificado asociado.

La herramienta Jarsigner te pedirá las passwords para el keystore y el alias.

La forma básica de este comando asume que el keystore a utilizar es un fichero llamado `.keystore` en tu directorio. Creará la firma y el fichero de bloques de firmas con los nombres `x.SF` y `x.DSA` respectivamente, donde `x` son las primeras ocho letras del alias, todas en mayúsculas. Este comando básico, sobre-escribirá el fichero JAR original con el fichero JAR firmado.

En la práctica, podrías querer utilizar este comando en conjunción con una o más de estas funciones:

- `-keystore file` - especifica un fichero keystore por defecto si no quieres utilizar la base de datos por defecto `.keystore`.
- `-storepass password` - Te permite introducir la password del keystore en la línea de comandos en vez de pedírtelo.
- `-keypass password` - Te permite introducir la password del alias en la línea de comandos en vez de pedírtelo.

- -sigfile file - especifica el nombre base para los ficheros .SF y .DSA si no quieres que el nombre base se tome de tu alias. file debe estar compuesto solo por letras mayúsculas (A-Z), numerales (0-9), subrayado.
- -signedjar file - especifica el nombre del fichero JAR firmado a generar, si no quieres que el fichero original sea sobre-escrito con el fichero firmado.

Ejemplo

Veamos un par de ejemplos de firma de ficheros JAR con la herramienta Jarsigner. En estos ejemplos asumimos que:

- Tu alias es "johndoe".
- la keystore que quieres utilizar es un fichero llamado "mykeys" en el directorio de trabajo actual.
- la password del keystore es "abc123".
- la password de tu alias es "mypass".

Para firmar un fichero JAR llamado app.jar, podrías utilizar este comando:

```
jarsigner -keystore mykeys -storepass abc123  
-keypass mypass app.jar johndoe
```

Como este comando no utiliza la opción -sigfile, los ficheros .SF y .DSA se crearan con los nombres JOHNDOE.SF y JOHNDOE.DSA. Como el comando no utiliza la opción -signedjar, el fichero firmado resultante sobre-escribirá la versión original de app.jar.

Veamos que sucedería si utilizáramos una combinación de opciones diferente:

```
jarsigner -keystore mykeys -sigfile SIG  
-signedjar SignedApp.jar app.jar johndoe
```

Esta vez, te pedirá que introduzcas las passwords para el Keystore y el alias porque no se han especificado en la línea de comandos. (Por razones de seguridad, probablemente no es una buena idea especificar tus passwords en la línea de comandos.) Los ficheros de firma y bloque de firmas se llamarán SIG.SF y SIG.DSA, respectivamente, y el fichero firmado SignedApp.jar se situará en el directorio actual. El fichero JAR original permanece como estaba.

Página de Referencia de Jarsigner

Las páginas completas de referencia de la herramienta de firma y verificación de ficheros JAR están on-line:

- [Página de Referencia de Jarsigner con ejemplos Windows](#)
 - [Página de Referencia de Jarsigner con ejemplos Solaris](#)
-

La herramienta de Seguridad Java del JDK 1.1

Si estás utilizando la versión 1.1 del JDK, utilizarás la herramienta de seguridad Java para firmar ficheros JAR. Para llamar a esta herramienta se utiliza el comando `javakey`, por eso, la llamaremos "Javakey" para acortar.

La herramienta Javakey maneja una base de datos que contiene parejas de claves pública/privada y los certificados relacionados. Para firmar un fichero JAR con esta herramienta, necesitas tener una pareja de claves pública/privada en esta base de datos. La herramienta JavaKey buscará la base de datos en la localización especificada por la propiedad `identity.database` en el fichero de propiedades de seguridad, `java.security`. Normalmente la base de datos contiene pares de claves de muchos firmantes potenciales, cada par de claves está asociado con el nombre de usuario de un firmante.

Además de los pares de claves, la base de datos de Javakey contiene certificados para las claves públicas. Cuando se añade un certificado a la base de datos, Javakey le asigna un número único para propósitos de identificación.

Para firmar un fichero, debes proporcionar a Javakey esta información:

- El nombre de usuario del par de claves a utilizar.
- El número de certificado a utilizar.
- El nombre que deben tener los ficheros de firma y de bloque de firmas.
- El nombre del fichero JAR firmado.

Esta información se proporciona utilizando un `directive file`, que es básicamente un fichero de propiedades que lee Javakey cuando firma un fichero JAR. Aquí tienes un ejemplo:

```
# The signer property specifies the username corresponding to
# the key pair that Javakey is to use to sign the JAR file.
# In this example, Javakey will sign the file using the key pair
# belonging to user "rita".
```

```
signer=rita
```

```
# The cert property tells Javakey which certificate to use. Each
# certificate in Javakey's database is identified by a number.
# To see a list of all the certificates and associated numbers in
# the database, use the command 'javakey -ld'.
```

```
cert=1
```

```
# The signature.file property specifies the name that the signature
# file and signature block file are to have. In this example,
```

```
# the files will be named SIGFILE.SF and SIGFILE.DSA, respectively.
```

```
signature.file=sigfile
```

```
# The out.file property specifies the name that Javakey should give  
# to the signed JAR file it produces. This property is optional.  
# If it's not present, Javakey will give the signed file the name  
# of the original JAR file, but with a .sig filename extension.
```

```
out.file=rita.jar
```

Una vez que tu fichero directivo está listo, puedes firmar tu fichero JAR utilizando un comando con esta forma:

```
javakey -gs directive-file jar-file
```

En este comando:

- -gs es la opción que le dice a Javakey que firme un fichero JAR.
- directive-file es el path y el nombre del fichero deirectivo que Javakey debería utilizar.
- jar-file es el path y el nombre del fichero JAR que quieres firmar.

Javakey situará el fichero JAR firmado en el directivo actual.

Javakey puede realizar muchas otras funciones relacionadas con el manejo de bases de datos de claves y certificados. Puedes ver la documentación on-line del JDK para más información sobre JavaKey:

- [Página de referencia de Javakey](#) para Windows
- [Página de referencia de Javakey](#) para Solaris

Verificar Ficheros JAR Firmados

Normalmente, la verificación de ficheros JAR firmados será responsabilidad de tu entorno de ejecución Java™ . Asumiendo que utilizas el JDK 1.1 o posterior, tu navegador verificará los applets firmados que descargues. Y en la versión 1.2 de la plataforma Java, las aplicaciones Java firmadas llamadas con la opción -jar del intérprete serán verificadas por el entorno de ejecución.

Sin embargo, puedes verificar ficheros JAR firmados, utilizando la herramienta Jarsigner del JDK 1.2. Por ejemplo, podrías hacer esto para probar un fichero JAR firmado que tu has preparado. La herramienta Jarsigner puede verificar ficheros firmados con el propio Jarsigner o con la herramienta Javakey del JDK 1.1.

Nota: La versión 1.1 del JDK no proporciona una utilidad para verificar ficheros JAR firmados.

El comando básico para verificar un fichero JAR firmado es:

```
jarsigner -verify jar-file
```

Este comando verificará la firma del fichero JAR y se asegura que los ficheros del archivo no han cambiado desde que se firmó. Verás este mensaje:

```
jar verified.
```

si la verificación tuvo éxito. Si intentas verificar un fichero JAR sin firmar, verás el siguiente mensaje:

```
jar is unsigned. (signatures missing or not parsable)
```

Si falla la verificación, se mostrará el mensaje apropiado. Por ejemplo, si el contenido del fichero JAR ha cambiado desde que fue firmado, se mostrará un mensaje como éste si intentas verificar la firma del fichero:

```
jarsigner: java.lang.SecurityException: invalid SHA1  
signature file digest for test/classes/Manifest.class
```


Entender la Firma y la Verificación

La plataforma Java™ te permite firmar digitalmente tus ficheros JAR. Se firma digitalmente un fichero por la misma razón por la que se firma un documento de papel - para permitir a los lectores saber que tú has escrito el documento, o al menos que el documento tiene tu aprobación.

Por ejemplo, cuando firmas una carta, quien reconozca tu firma puede confirmar que tú has escrito la carta. De forma similar, cuando firmas un fichero digitalmente, aquel que "reconozca" tu firma digital sabe que el fichero viene de tí. El proceso de "reconocimiento" de firmas electrónicas se llama verificación.

¿Por qué es tan importante firmar y verificar los ficheros? Cuando descargas un fichero de un applet sin firmar de la web, la plataforma Java no le permite que realice operaciones sensibles a la seguridad como leer o escribir ficheros locales o ejecutar programas locales. Pero ¿qué sucede si dichas operaciones son una parte crucial de la funcionalidad del applet? y ¿qué pasa si quieres que tu applet realice una operación prohibida normalmente? Aquí es donde aparecen la firma y la verificación.

Si un applet está firmado, tú puedes determinar si viene de una fuente correcta. Una vez que tú (o tu navegador) hayas verificado que el applet viene de una fuente correcta, la plataforma puede relajar las restricciones de seguridad y permitir que el applet realice algunas operaciones que ordinariamente estarían prohibidas. En la versión 1.1 de la plataforma Java, un applet verdadero tiene la misma libertad que una aplicación local. En la versión 1.2 de la plataforma, un applet verdadero tiene las libertades especificadas en el fichero de policía.

La plataforma Java permite la firma y verificación utilizando números especiales llamados claves pública y privada. Estas claves vienen en parejas y juegan papeles complementarios.

La clave privada es el "lápiz" electrónico con el que tú firmas un fichero. Como su nombre indica, tu clave privada sólo la conoces tú, por eso nadie puede falsificar tu firma. Un fichero firmado con tu clave privada sólo puede ser verificado por la correspondiente clave pública.

Sin embargo, sólo las claves pública y privada, no son suficientes para verificar una firma. Una clave pública sólo puede verificar que un fichero firmado fue firmado por la clave privada correspondiente. Por eso aunque una clave pública puede decirte que una firma es "auténtica", no puede decirte de quien es la firma!

Por lo tanto, se requiere un elemento más para hacer el trabajo de firma y verificación. El elemento adicional es el certificado que los firmantes incluyen en un fichero JAR firmado. Un certificado es una sentencia firmada digitalmente por una autoridad en certificación que indica quien es el propietario de una firma particular. En el caso de los ficheros JAR firmados, el certificado indica a quién

pertenece la clave pública contenida en el fichero JAR.

Cuando firmas un fichero JAR tu clave pública se sitúa dentro del archivo junto con un certificado asociado para facilitar el uso por cualquiera que quiera verificar tu firma.

Para sumarizar la firma digital:

- El firmante firma el fichero JAR utilizando una clave privada.
- La correspondiente clave pública se sitúa dentro del fichero JAR, junto con su certificado, para que sea posible su uso por quien quiera verificar la firma.

El fichero de Firma

Cuando firmas un fichero JAR, se añade automáticamente un fichero de firma en el directorio META-INF del fichero JAR. Este es el mismo directorio que contiene el [manifiesto](#) del fichero JAR. Los ficheros de firma tienen la extensión .SF.

Aquí tienes un ejemplo del contenido de un fichero de firma:

```
Signature-Version: 1.0
SHA1-Digest-Manifest: hlyS+K9T7DyHtZrtI+LxvgqaMYM=
Created-By: SignatureFile JDK 1.2
```

```
Name: test/classes/Manifest.class
SHA1-Digest: fcav7ShIG6i86xPepmitOV04vWY=
```

```
Name: test/classes/SignatureFile.class
SHA1-Digest: xrQem9snnPhLySDiZyclMlsFdtM=
```

```
Name: test/images/manifest-concept.gif
SHA1-Digest: kdHbE7kL9ZHLgK7akHttYV4XIa0=
```

```
Name: test/images/manifest-schematic.gif
SHA1-Digest: mF0D5zpk68R4oaxEqoS9Q7nhm60=
```

El fichero de firma contiene entradas para cada nombre de fichero el valor de resumen. La línea SHA1-Digest-Manifest contiene el resumen del fichero de manifiesto .

Los valores de resumen son representaciones codificadas del contenido de los ficheros en el momento en que fueron firmados. El resumen de un fichero cambiará sólo si cambia el propio fichero. Cuando se verifica un fichero JAR firmado, se calcula el resumen de cada fichero y se compara con el resumen almacenado en el fichero de manifiesto para asegurarse de que el contenido del fichero JAR no ha cambiado desde que se firmó.

Puedes encontrar información adicional sobre la firma de ficheros en las páginas de

documentación del JDK [Formato del Manifiesto](#).

El fichero de bloques de firmas

Además del fichero de firma, cuando se firma un fichero JAR se añade automáticamente un fichero de bloque de firma en el directorio META-INF. Al contrario que el fichero de manifiesto o el fichero de firma, los ficheros de bloques de firmas no tienen un formato leible.

El fichero de bloque de firmas contiene dos elementos esenciales para la verificación:

- La firma digital para el fichero JAR que fue generado por la clave privada del firmante.
- La clave pública del firmante, junto con su certificado, para ser utilizadas por cualquiera que quiera verificar el fichero JAR.

Los nombres de los ficheros de bloques de firmas normalmente tienen la extensión, .DSA indicando que han sido creados por el Algoritmo de Firma Digital por defecto. Sin embargo, son posibles otras extensiones si las claves están asociadas con otros algoritmos estándar.

Documentación Relacionada

Para información adicional sobre las claves, los certificados, y autoridades de certificación, puedes ver

- [Las claves del JDK 1.2 y la herramienta de Manejo de Certificado](#)
- [Certificados X.509](#)

Para más información sobre la arquitectura de seguridad de la plataforma Java, puedes ver esta documentación relacionada:

- [Especificación y Referencia sobre el API Java de Criptografía](#)
- [Documentación sobre seguridad en el JDK 1.2](#)

Utilizar el JNI (Java Native Interface)

Las páginas de esta sección muestran como integrar código nativo en programas escritos en Java. También aprenderemos cómo escribir métodos nativos en lenguaje Java. Los métodos nativos están implementados en otros lenguajes como C. Además, el API Invocation nos permite incluir la Máquina Virtual Java en nuestras aplicaciones nativas.

Paso a Paso

Explica paso a paso un sencillo ejemplo (los métodos nativos del programa "Hello World!") para ilustrar cómo escribir, compilar y ejecutar un programa Java con métodos nativos.

El Interface de Programación Nativo de Java

Muestra cómo implementar el lado del lenguaje Java y el lado del lenguaje nativo de un método nativo. Esta lección incluye información sobre el paso de argumentos de distintos tipos de datos a un método nativo y cómo devolver distintos tipos de datos desde un método nativo. Esta lección también describe muchas funciones útiles que nuestro lenguaje Nativo puede utilizar para acceder a objetos Java y sus miembros, para crear objetos Java, lanzar excepciones, llamar a la Máquina Virtual Java, y mucho más.

Consideraciones de Seguridad: Observa que la posibilidad de la carga de librerías dinámicas está sujeta a la aprobación del controlador de seguridad. Cuando se trabaja con métodos nativos, se deben cargar librerías dinámicas. Algunos applets no podrán utilizar métodos nativos porque el navegador o visualizador en que se están ejecutando restringe la carga de librerías dinámicas. Puedes ver [Restricciones de Seguridad](#) para más información sobre las restricciones de seguridad de los applets.

Nota: Los programadores de MacOS deberán referirse a [MacOS Runtime para Java \(MRJ\)](#).

Paso a Paso:

Estas páginas pasean a través de los pasos necesarios para integrar código nativo en programas escritos en Java.

Esta lección implementa el consabido programa "Hello World!". Este programa tiene dos clases Java. La primera, llamada Main, implementa el método `main()` para todo el programa. La segunda, llamada HelloWorld, es un método, un método nativo, que muestra "Hello World!". La implementación para el método nativo se ha proporcionado en lenguaje C.

Paso 1: Escribir el Código Java

Crea un clase Java llamada HelloWorld que declara un método nativo. También, escribe el programa principal que crea el objeto HelloWorld y llama al método nativo.

Paso 2: Compilar el Código Java

Utiliza `javac` para compilar el código Java escrito en el Paso 1.

Paso 3: Crear el fichero .h

Utiliza `javah` para crear un fichero de cabecera (un fichero `.h`) al estilo JNI, a partir de la clase HelloWorld. El fichero de cabecera proporciona una definición de función para la implementación del método nativo `displayHelloWorld()`, que se ha definido en la clase HelloWorld.

Paso 4: Escribir la Implementación del Método Nativo

Escribe la implementación para el método nativo en un fichero fuente en el lenguaje nativo. La implementación será una función normal que será integrada con nuestra clase Java.

Paso 5: Crear una Librería Compartida

Utiliza el compilador C para compilar el fichero `.h` y el fichero `.c` que se han creado en los pasos 3 y 4 en una librería compartida. En terminología Windows 95/NT, una librería compartida se llama Librería de Carga Dinámica (DLL).

Paso 6: Ejecutar el Programa

Y finalmente, utiliza `java`, el intérprete del lenguaje Java, para ejecutar el programa.

Paso 1: Escribir el Código Java

El siguiente fragmento de código Java define una clase llamada [HelloWorld](#). Esta clase tiene un segmento de código estático:

```
class HelloWorld {  
    public native void displayHelloWorld();  
  
    static {  
        System.loadLibrary("hello");  
    }  
}
```

Definir un Método Nativo

Todos los métodos, tanto métodos Java, como métodos nativos, se deben definir dentro de una clase Java. Cuando se escribe la implementación de un método en un lenguaje de programación distinto de Java, se debe incluir la palabra clave `native` como parte de la definición del método dentro de la clase java. La clave `native` indica al compilador Java que la función es una función en lenguaje nativo. Es sencillo decir que la implementación del método `displayHelloWorld()` de la clase `HelloWorld` está escrita en otro lenguaje de programación porque la clave `native` aparece como parte de su definición de método:

```
public native void displayHelloWorld();
```

Esta definición de método en nuestra clase Java sólo proporciona la firma del método para `displayHelloWorld()`. No proporciona la implementación para el método. Se deberá proporcionar la implementación para `displayHelloWorld()` en un fichero fuente separado en el lenguaje nativo.

La definición de método para `displayHelloWorld()` también indica que el método es un método de ejemplar público, no acepta ningún argumento, y no devuelve ningún valor. Para más información sobre los argumentos y los valores de retorno desde métodos nativos puedes ver [Programación de Interface Nativo en Java](#).

Carga de la Librería

Se debe compilar el código de lenguaje nativo que implementa `displayHelloWorld` dentro de una librería compartida (se hará esto en el [Paso 5: Crear una Librería Compartida](#)). También se debe cargar la librería compartida dentro de la clase Java que la requiere. Esta carga de la librería compartida dentro de la clase Java mapea la implementación

del método nativo a su definición.

Se utiliza el método `System.loadLibrary` para cargar la librería compartida que se creo cuando compilamos la implementación del código. Este método se coloca con un inicializador `static`. El argumento de `System.loadLibrary` es el nombre de la librería. El sistema utiliza un estándar, pero específico de la plataforma, para convertir el nombre de la librería a un nombre de librería nativo. Por ejemplo, en Solaris convierte el nombre "hello" a `libhello.so`, mientras que en Win32 convierte el mismo nombre de librería a `hello.dll`.

El siguiente inicializador `static` de la clase `HelloWorld` carga la librería apropiada, llamada `hello`. El sistema de ejecución ejecuta un inicializador estático cuando carga la clase.

```
static {  
    System.loadLibrary("hello");  
}
```

Crear el Programa Principal

También se debe crear un fichero fuente separado con una aplicación Java que ejemplarize la clase que contiene la definición de método nativo. Esta aplicación Java también llamará al método nativo. Como es una aplicación, debe tener un método `main`. En un fichero fuente separado, que para este ejemplo hemos llamado [Main.java](#), se crea una aplicación Java que ejemplariza `HelloWorld` y llama al método nativo `displayHelloWorld()`.

```
class Main {  
    public static void main(String[] args) {  
        new HelloWorld().displayHelloWorld();  
    }  
}
```

Como se puede ver a partir del código anterior, se llama a un método nativo de la misma forma en que se llamaría a un método normal: sólo añadir el nombre del método al final del nombre del objeto, separado con un punto ('.'). Una pareja de paréntesis, (), sigue al nombre del método y encierra los argumentos. El método `displayHelloWorld()` no tiene ningún argumento.

Paso 2: Compilar el Código Java

Se utiliza el compilador del lenguaje Java para compilar la clase Java creada en el [paso anterior](#). En este momento también se debería compilar la aplicación Java que se escribió para probar el método nativo.

Ozito

Paso 3: Crear el fichero .h

En este paso se utiliza el programa de utilidad javah para generar un fichero de cabecera (un fichero .h) desde la clase Java HelloWorld. El fichero de cabecera proporciona un prototipo de función para la implementación del método nativo displayHelloWorld() definido en esta clase.

[Ejecuta javah](#) ahora sobre la clase HelloWorld que se creó en los pasos anteriores.

Por defecto, javah sitúa el nuevo fichero .h en el mismo directorio que el fichero .class. Se utiliza la opción -d para instruir a javah para que sitúe el fichero de cabecera en un directorio diferente.

El nombre del fichero de cabecera es el nombre de la clase java con extensión .h. Por ejemplo, el comando anterior generará un fichero llamado [HelloWorld.h](#).

La definición de Función

Mira el fichero de cabecera HelloWorld.h.

```
#includejava example-1dot1/HelloWorld.h
```

Java_HelloWorld_displayHelloWorld() Es la función que proporciona la implementación del método nativo de la clase HelloWorld, que se escribirá en el [Paso 4: Escribir la Implementación del Método Nativo](#). Se utiliza la firma de la función cuando se escribe la implementación del método nativo.

Si HelloWorld contuviera otros métodos nativos, sus firmas de función deberían aparecer aquí también.

El nombre de la función en el lenguaje nativo que implementa el método nativo consiste en el prefijo Java_, el nombre del paquete, el nombre de la clase, y el nombre del método nativo. Entre cada nombre de componente hay un subrayado "_" como separador. El nombre de paquete se omite cuando el método está en el paquete por defecto.

Así, el método nativo displayHelloWorld dentro de la clase HelloWorld se convierte en Java_HelloWorld_displayHelloWorld(). En nuestro ejemplo, no hay nombre de paquete, porque HelloWorld está en el paquete por defecto.

Observa que la implementación de la función en el lenguaje nativo, que aparece en el fichero de cabecera, acepta dos parámetros, aunque en su definición en lenguaje Java no aceptará ninguno. El JNI requiere que cualquier método nativo tenga estos dos parámetros. El primer parámetro es un puntero a un interface JNIEnv. A través de este puntero, el código nativo podrá acceder a los parámetros y objetos de la aplicación Java. El parámetro jobject es una referencia al propio objeto. Para un método nativo no-estático como el método displayHelloWorld de nuestro

ejemplo, este argumento es una referencia al objeto. Para métodos nativos estáticos, este argumento sería una referencia al método Java. Para aclararlo un poco, se puede pensar en el parámetro jobject como en la variable "this" de C++. Nuestro ejemplo ignora ámbos parámetros.

La siguiente lección [Programación del Interface Nativo en Java](#), describe como acceder a los datos utilizando el parámetro env.

Ozito

Paso 4: Escribir la Implementación del Método Nativo

Ahora podemos entrar en el negocio de escribir la implementación del método nativo en otro lenguaje distinto de java.

La función que escribamos debe tener la misma firma de función que la que se generó con javah dentro del fichero HelloWorld.h en el [Paso 3: Crear el fichero .h](#). Recuerda que la firma de la función generada para el método nativo displayHelloWorld() de la clase HelloWorld, se parece a esto:

```
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld(JNIEnv *, jobject);
```

Aquí tienes la implementación en lenguaje C para el método nativo Java_HelloWorld_displayHelloWorld(). Esta implementación se encuentra en el fichero llamado [HelloWorldImp.c](#).

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

La implementación para Java_HelloWorld_displayHelloWorld() es correcta: la función utiliza la función printf() para mostrar el string "Hello World!" y retorna.

Este fichero incluye tres ficheros de cabecera:

- jni.h - Este fichero de cabecera proporciona información que el código nativo necesita para interactuar con el sistema de ejecución Java. Cuando se escriban métodos nativos, siempre se debe incluir este fichero de cabecera en los ficheros fuente nativos.
- HelloWorld.h - El fichero .h que se generó en el [Paso 3: Crear el fichero .h](#).
- stdio.h - El código anterior también incluye el fichero stdio.h porque utiliza la función printf().

Paso 5: Crear una Librería Compartida

Recuerda que en el [Paso 1: Escribir el Código Java](#) se utilizó la siguiente llamada de método para cargar una librería compartida llamada hello dentro de nuestro programa en el momento de la ejecución:

```
System.loadLibrary("hello");
```

Ahora estamos listos para crear la librería compartida.

En el [paso anterior](#), creamos un fichero C en el que escribimos la implementación para el método nativo displayHelloWorld. Se grabó el método nativo en el fichero HelloWorldImp.c. Ahora, deberemos compilar este fichero en una librería, que debe llamarse hello para corresponder con el nombre utilizado en el método System.loadLibrary.

Se utilizan las herramientas disponibles para compilar el código nativo que se creó en los pasos anteriores en una librería compartida. En Solaris, se creará una librería compartida, mientras que en Windows 95/NT se creará una librería de enlace dinámico (DLL). Recuerda especificar el path o paths necesarios para todos los ficheros de cabecera necesarios.

En Solaris, el siguiente comando construye una librería llamada libhello.so:

```
cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris \
    HelloWorldImp.c -o libhello.so
```

En Win32, el siguiente comando construye una librería de enlace dinámico hello.dll utilizando Microsoft Visual C++ 4.0:

```
cl -Ic:\java\include -Ic:\java\include\win32 -LD HelloWorldImp.c -Fehello.dll
```

Por supuesto, se necesita especificar el path de include que corresponda con la configuración de nuestra máquina.

Paso 6: Ejecutar el Programa

Ahora ejecuta la aplicación Java (La clase Main) con el intérprete java. Deberías ver la siguiente salida:

```
Hello World!
```

Si ves una excepción como ésta:

```
java.lang.UnsatisfiedLinkError: no hello in shared library path
    at java.lang.Runtime.loadLibrary(Runtime.java)
    at java.lang.System.loadLibrary(System.java)
    at
    at java.lang.Thread.init(Thread.java)
```

es porque no tienes configurado correctamente el path de librerías. El path de librerías es una lista de directorios en la que el sistema de ejecución Java busca cuando se cargan librerías. [Configuración del path de librerías](#), y asegurate del nombre del directorio donde está la librería hello.

Interface de Programación Nativo de Java

El JDK 1.1 soporta el Interface Nativo Java (JNI). Por un lado, el JNI define un estándar de nombres y convenciones de llamadas para que la Máquina Virtual Java (VM) pueda encontrar nuestros métodos nativos. Por otro lado, el JNI ofrece un conjunto de funciones de interface estándar. Se puede llamar a las funciones JNI desde nuestros métodos nativos para hacer cosas como acceder y manipular objetos Java, liberar objetos Java, crear nuevos objetos, llamar a métodos Java, etc.

Esta sección muestra cómo seguir las convenciones de nombres y de llamadas del JNI, y como utilizar las funciones del JNI desde un método nativo. Cada ejemplo consiste en un programa escrito en Java que llama a varios métodos nativos implementados en C. Los métodos nativos, a su vez, podrían llamar a las funciones JNI para acceder a los objetos Java.

La sección también muestra cómo incluir la máquina virtual Java dentro de una aplicación nativa.

Declarar Métodos Nativos

En el lado del lenguaje Java, se declara un método nativo con la palabra clave `native` y un cuerpo de método vacío. En el lado del lenguaje nativo, se proporciona la implementación del método nativo. Debemos tener cuidado cuando escribamos métodos nativos que "corresponda" la implementación de la función nativa con la firma del método en el fichero de cabecera del lenguaje Java. La herramienta `javah`, que se explicó en el [Paso 3: Crear el fichero .h](#), nos ayuda a generar prototipos de funciones nativas que corresponden con la declaración del método nativo en lenguaje Java.

Mapear entre Tipos Java y Tipos Nativos

El JNI define un mapeado de los tipos del lenguaje Java y los tipos de lenguaje nativo (C/C++). Esta página presenta los tipos nativos correspondientes tanto a los tipos primitivos Java como `int` y `double`, y los objetos del lenguaje Java, incluyendo `strings` y `arrays`.

Acceder a Strings Java

Los `Strings` son un tipo de objeto Java particularmente útil. El JNI proporciona un conjunto de funciones para manipulación de `Strings` que hace sencilla la tarea manejar objetos `Strings` de Java en el código nativo. El programador puede traducir entre `Strings` Java y `Strings` nativos en los formatos `Unicode` y `UTF-8`.

[Acceder a Arrays Java](#)

Los Arrays son otro tipo de objetos Java muy utilizados. Se pueden utilizar las funciones de manipulación de arrays del JNI para crear y acceder a los elementos de un array.

[Llamar a Métodos Java](#)

El JNI soporta un completo conjunto de operaciones de llamada que permiten invocar métodos Java desde código nativo. Se localiza el método utilizando su nombre y su firma. Se pueden llamar tanto a métodos estáticos como a métodos de ejemplar (no estáticos). Se puede utilizar la herramienta javap para generar firmas de métodos al estilo JNI para los ficheros de clases.

[Acceder a Campos Java](#)

El JNI permite localizar un campo Java utilizando el nombre y la firma del campo. Se pueden localizar y acceder tanto a campos estáticos como campos de ejemplar (no estáticos). Se puede utilizar la herramienta javap para generar firmas de campos al estilo JNI para los ficheros de clases.

[Capturar y Lanzar Excepciones](#)

Esta sección enseña como tratar con excepciones dentro de una implementación de método nativo. Nuestros métodos nativos pueden capturar, lanzar y borrar excepciones.

[Referencias Locales y Globales](#)

El código nativo se puede referir a objetos Java utilizando referencias locales o globales. Las referencias locales son sólo válidas dentro de una llamada a método nativo. Son liberadas automáticamente después de que retorne el método nativo. Las referencias globales deben asignarse y liberarse explícitamente.

[Threads y Métodos Nativos](#)

Esta página describe la implicación de ejecutar métodos nativos en un entorno multi-thread. El JNI ofrece constructores de sincronización básicos para métodos nativos.

[Llamar a la Máquina Virtual Java](#)

Esta sección enseña cómo cargar la Máquina Virtual Java desde una

librería nativa en una aplicación nativa. Incluye instrucciones de cómo inicializar la Máquina Virtual Java e invocar métodos Java. El API "Invocation" también permite añadir threads nativos a una Máquina Virtual Java que ya está ejecutándose y convertirse ellos mismos en threads Java. Actualmente, el JDK sólo soporta la adición de threads nativos en Win32. El soporte para threads nativos en Solaris estará disponible en una futura versión.

Programación JNI en C++

En el lenguaje C++, el JNI presenta un interface mucho más claro y realiza chequeos de tipos estáticos adicionales.

Declarar Métodos Nativos

Esta página ilustra cómo declarar un método nativo en el lenguaje Java y cómo generar el correspondiente prototipo de función en C/C++.

El Lado Java

Nuestro primer ejemplo, [Prompt.java](#), contiene un método nativo que acepta e imprime un String Java. El programa llama al método nativo, que espera por una entrada del usuario y luego devuelve la línea que el usuario a tecelado.

La clase Java Prompt contiene un método main que es utilizado para llamar al programa. Además, existe un método nativo `getLine`:

```
private native String getLine(String prompt);
```

Observa que las declaraciones para métodos nativos son casi idénticas a las declaraciones de métodos normales de Java. Existen dos diferencias. Los métodos nativos deben tener la palabra clave `native`. Esta palabra clave informa al compilador Java que la implementación para este método se proporciona en otro lenguaje. También, la declaración del método nativo termina con un punto y coma, el símbolo de terminador de sentencia, porque no hay implementaciones para métodos nativos en los ficheros de clases del lenguaje Java.

El Lado del Lenguaje Nativo

Se deben declarar e implementar los métodos nativos en un lenguaje nativo, como C o C++. Antes de hacer esto, es útil generar un fichero de cabecera que contenga el prototipo de la función para la implementación del método nativo.

Compilamos el fichero `Prompt.java` y luego generamos el fichero `.h`. Primeros compilamos el fichero `Prompt.java` de esta forma:

```
javac Prompt.java
```

Una vez compilado satisfactoriamente y habiendo creado el fichero `Prompt.class`, podemos crear un fichero de cabecera al estilo JNI, especificando una opción `-jni` a `javah`:

```
javah -jni Prompt
```

Examinamos el fichero [Prompt.h](#). Observa el prototipo de función para el método nativo `getLine` que declaramos en [Prompt.java](#).

```
JNIEXPORT jstring JNICALL  
Java_Prompt_getLine(JNIEnv *, jobject, jstring);
```

La definición de la función del método nativo en el código de implementación debe

corresponder con el prototipo de función generado en el fichero de cabecera. Siempre se debe incluir JNIEXPORT y JNICALL en los prototipos de función para métodos nativos. JNIEXPORT y JNICALL se aseguran de que el código fuente se compilará en plataformas como Win32 que requieren palabras claves especiales para funciones exportadas desde librerías de enlace dinámico.

Los nombres de los métodos nativos son una concatenación de los siguientes componentes:

- El prefijo Java_
- El nombre de la clase totalmente cualificado
- Un subrayado "_" separador
- el nombre del método

(Observa que los nombres de los métodos nativos sobrecargados, además de los componentes anteriores, tienen un doble subrayado "__" añadido al nombre del método y seguido por la firma de argumentos).

Así, la implementación en código nativo para el método Prompt.getLine se convierte en Java_Prompt_getLine. (No existe nombre de paquete, porque la clase Prompt está en el paquete por defecto).

Cada método nativo tiene dos parámetros además de los que pongamos en la declaración en lenguaje Java. El primer parámetro, JNIEnv *, es el puntero al interface JNI. Este puntero a interface está organizado como una tabla de funciones, donde cada función JNI tiene un punto de entrada conocido en la tabla. Nuestros métodos nativos llaman a funciones específicas del JNI para acceder a objetos Java a través del puntero JNIEnv *. El parámetro jobject es una referencia al propio objeto (es como el puntero this en C++).

Y por último, observa que el JNI tiene un conjunto de tipos de nombres, como jobject y jstring, y cada tipo corresponde con un tipo del lenguaje Java. Todos esto se cubre en la [página siguiente](#).

Conversión entre Tipos Java y Tipos Nativos

En esta página aprenderemos como referenciar los tipos Java en nuestro método nativo. Se necesitará referenciar a los tipos Java cuando queramos:

- Acceder a los argumentos pasados a un método nativo desde una aplicación Java.
- Crear nuevos objetos Java en nuestro método nativo.
- Hacer que nuestro método nativo devuelva resultados a su llamador.

Tipos Primitivos de Java

Nuestro método nativo puede acceder directamente a los tipos primitivos de Java, como booleans, integers, floats, etc., que sean pasados desde programas escritos en lenguaje Java. Por ejemplo, el tipo boolean de Java se convierte en el tipo jboolean en el lenguaje nativo (representado como 8 bits sin signo), mientras que el tipo float de Java se convierte en el tipo jfloat en el lenguaje nativo (representado por 32 bits). La siguiente tabla describe las conversiones entre los tipos primitivos de Java y los tipos nativos.

Tipos Primitivos y Nativos Equivalentes

Tipo Java	Tipo Nativo	Tamaño en bits
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	n/a

Tipos de Objetos Java

Los Objetos Java se pasan por referencia. Todas las referencias a objetos Java tienen el tipo jobject. Por conveniencia y para reducir los errores de programación, el JNI implementa un conjunto de tipos que conceptualmente son todos "Subtipos" de jobject, de la siguiente forma:

- jobject representa todos los objetos Java.
 - jclass representa objetos del tipo (java.lang.Class).
 - jstring representa Strings Java (java.lang.String).
 - jarray representa arrays Java.

- `jobjectArray` representa arrays de objetos.
- `jbooleanArray` representa arrays booleanos.
- `jbyteArray` representa arrays de bytes.
- `jcharArray` representa arrays de char.
- `jshortArray` representa arrays de short.
- `jintArray` representa arrays de int.
- `jlongArray` representa arrays de long.
- `jfloatArray` representa arrays de float.
- `jdoubleArray` representa arrays de double.
- `jthrowable` representa excepciones Java (`java.lang.Throwable`).

En nuestro ejemplo `Prompt.java`, el método nativo `getLine`:

```
private native String getLine(String prompt);
```

toma un `String` Java como argumento y devuelve un `String` Java. Su implementación nativa correspondiente tiene el tipo `jstring` tanto para el argumento como para el valor de retorno:

```
JNIEXPORT jstring JNICALL  
Java_Prompt_getLine(JNIEnv *, jobject, jstring);
```

Como se mencionó anteriormente, `jstring` corresponde al tipo `String` de Java. Observa que el segundo argumento de `Java_Prompt_getLine`, que es una referencia al propio objeto, tiene el tipo `jobject`.

Strings Java

Cuando una aplicación Java pasa un string a un programa en lenguaje nativo, le pasa el string como del tipo `jstring`. Este tipo es muy diferente del tipo string normal del C (`Char *`). Si nuestro código intenta imprimir directamente un `jstring`, resultará en una caída de la VM (Máquina Virtual). Por ejemplo, el siguiente segmento de código intenta imprimir incorrectamente un `jstring` y podría resultar en una caída de la máquina virtual:

```
/* DO NOT USE jstring THIS WAY !!! */
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    printf("%s", prompt);
}
```

Nuestro método nativo debe utilizar las funciones JNI para convertir los strings Java en strings nativos. El JNI soporta la conversión desde y hacia strings nativos Unicode y UTF-8. En particular, los strings UTF-8 utilizan el bit más alto para señalar caracteres multi-bytes; por lo tanto son compatibles con el ASCII de 7-bits. En Java, los strings UTF-8 siempre terminan en 0.

Acceder a Strings Java

Nuestro método nativo necesita llamar a `GetStringUTFChars` para imprimir correctamente el string pasado desde la aplicación Java. `GetStringUTFChars` convierte la representación interna Unicode de un String Java en un string UTF-8. Una vez seguros de que el string sólo contiene caracteres ASCII de 7-bits, se puede pasar directamente a las funciones normales del C, como `printf`, como se vió en [Prompt.c](#).

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    char buf[128];
    const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
    printf("%s", str);
    (*env)->ReleaseStringUTFChars(env, prompt, str);
}
```

Observa que cuanto nuestro método nativo haya terminado de utilizar el string UTF-8, debe llamar a `ReleaseStringUTFChars`. `ReleaseStringUTFChars` informa a la VM que el método nativo ha terminado con el string, para que pueda liberar la memoria utilizada por el string. Si no llamamos a `ReleaseStringUTFChars` se nos llenará la memoria.

El método nativo también puede construir un nuevo string utilizando la función `NewStringUTF` del JNI. Las siguientes líneas de código de

Java_Prompt_getLine muestran esto:

```
scanf("%s", buf);  
return (*env)->NewStringUTF(env, buf);  
}
```

Utilizar el Puntero a Interface JNIEnv

Los métodos nativos deben acceder y manipular los objetos Java, como los strings, a través del puntero a interface env. En lenguaje C, esto requiere utilizar el puntero env para referenciar la función JNI. Observa cómo el método nativo utiliza el puntero env para referenciar las dos funciones, GetStringUTFChars y ReleaseStringUTFChars. Además, env se pasa como primer parámetro a esas funciones.

Otras Funciones JNI para Acceder a Strings Java

El JNI también proporcionar funciones para obtener la representación Unicode de strings Java. Esto es muy útil, por ejemplo, en aquellos sistemas operativos que soportan Unicode como formato nativo. También hay funciones útiles para obtener la longitud de un String Java tanto en UTF-8 como en Unicode.

- GetStringChars toma el string Java y devuelve un puntero a un array de caracteres Unicode que contiene el string.
- ReleaseStringChars libera el puntero al array de caracteres Unicode.
- NewString construye un nuevo objeto java.lang.String desde un array de caracteres Unicode.
- GetStringLength devuelve la longitud de un string que está contenido en un array de caracteres Unicode.
- GetStringUTFLength devuelve la longitud de un string si está representado en el formato UTF-8.

Acceder a Arrays Java

El JNI utiliza el tipo `jarray` para representar referencias a arrays java. Al igual que con `jstring`, no se puede acceder directamente a los tipos `jarray` desde el código nativo, se deben utilizar las funciones proporcionadas por el JNI que permiten obtener punteros a los elementos de arrays de enteros.

Nuestro segundo ejemplo, `IntArray.java`, contiene un método nativo que suma el total de un array de enteros pasado por la aplicación Java. No se puede implementar el método nativo direccionando directamente los elementos del array. El siguiente código intenta incorrectamente acceder directamente a los elementos del array:

```
/* This program is illegal! */
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
    int i, sum = 0;
    for (i=0; i<10; i++)
        sum += arr[i];
}
```

La forma correcta de implementar la función anterior, se puede ver en el método nativo [IntArray.c](#). En este ejemplo, se utiliza una función JNI para obtener la longitud del array. Luego se pueden recuperar los elementos. Y finalmente se utiliza una tercera función del JNI para liberar la memoria del array.

Acceder a un Array de Elementos Primitivos

Primero se obtiene la longitud del array llamando a la función `GetArrayLength` del JNI. Observa que al contrario que en los array del C, los array Java almacenan información sobre la longitud.

```
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
    int i, sum = 0;
    jsize len = (*env)->GetArrayLength(env, arr);
}
```

Luego se obtiene un puntero a los elementos del array. Nuestro ejemplo contiene un array de enteros, por eso utilizamos la función del JNI `GetIntArrayElements` para obtener este puntero. (El JNI proporciona un juego de funciones para obtener punteros a elementos del array; se utiliza la función que corresponde con el tipo primitivo del array). Una vez obtenido el puntero, se pueden utilizar las operaciones normales del C sobre el array de enteros resultante.

```
jint *body = (*env)->GetIntArrayElements(env, arr, 0);
```



```
for (i=0; i<len; i++)  
    sum += body[i];
```

En general, el recolector de basura podría eliminar los arrays Java. Sin embargo, la máquina virtual garantiza que el resultado de `GetIntArrayElements` apunta a un array de enteros inamovible. El JNI o bien baja el "pin" del array "marcándolo" o hace una copia del array en una memoria inamovible. A causa de esto, el código nativo debe llamar a `ReleaseIntArrayElements` cuando ha terminado de utilizar el array de esta forma:

```
(*env)->ReleaseIntArrayElements(env, arr, body, 0);  
return sum;  
}
```

`ReleaseIntArrayElements` permite al JNI copiar de vuelta y liberar la memoria referenciada por el parámetro `body` si es una copia del array original Java, "desmarca" el array java que fue marcado en la memoria. No olvides llamar a `ReleaseIntArrayElements`. Si se olvida hacer esta llamada el array permanece marcado por un largo periodo de Tiempo. Y la máquina virtual no podrá reclamar la memoria utilizada para almacenar la copia inamovible del array.

El JNI proporciona un conjunto de funciones para acceder a los elementos de arrays de los distintos tipos primitivos:

- `GetBooleanArrayElements` accede a los elementos de un array Java de `boolean`
- `GetByteArrayElements` accede a los elementos de un array Java de `bytes`
- `GetCharArrayElements` accede a los elementos de un array Java de `char`
- `GetShortArrayElements` accede a los elementos de un array Java de `short`
- `GetIntArrayElements` accede a los elementos de un array Java de `int`
- `GetLongArrayElements` accede a los elementos de un array Java de `long`
- `GetFloatArrayElements` accede a los elementos de un array Java de `float`
- `GetDoubleArrayElements` accede a los elementos de un array Java de `double`

Acceder a un Número Pequeño de Elementos

Observa que la función `Get< type> ArrayElements` potencialmente podría copiar el array entero. Podríamos querer copiar un número limitado de elementos, especialmente si el array es grande. Si sólo estamos interesados en un número pequeño de elementos (en un array potencialmente grande), deberíamos utilizar las funciones `Get/Set< type> ArrayRegion`. Estas funciones permiten acceder, mediante copia, a un conjunto pequeño de elementos de un array.

Acceder a Arrays de Objetos

El JNI proporciona un conjunto separado de funciones para acceder a elementos de

arrays de objetos. Se pueden utilizar estas funciones para obtener y seleccionar objetos individuales del array. Observa que no se puede obtener el array de objetos completo de una sola vez.

- `GetObjectArrayElement` devuelve el objeto del elemento en un índice dado.
 - `SetObjectArrayElement` actualiza el objeto del elemento en un índice dado.
-

Llamar a Métodos Java

Esta página ilustra cómo llamar a métodos Java desde métodos nativos. Nuestro programa de ejemplo, [Callbacks.java](#), llama a un método nativo. El método nativo hace una nueva llamada al método Java. Para hacer las cosas un poco más interesantes, el método Java llama de nuevo (recursivamente) al método nativo. Este proceso continúa hasta que la recursión alcanza cinco niveles de profundidad, en ese momento el método Java retorna sin hacer más llamadas al método nativo. Para ayudarnos a ver esto, los dos métodos imprimen una secuencia de trazado.

Llamar a un Método Java desde un Método Nativo

Para ver como el código nativo llama al método Java, enfoquémonos en la implementación de `Callbacks_nativeMethod`, que está implementada en [Callbacks.c](#). Este método nativo contiene una llamada al método java `Callbacks.callback`.

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "(I)V");
    if (mid == 0)
        return;
    printf("In C, depth = %d, about to enter Java\n", depth);
    (*env)->CallVoidMethod(env, obj, mid, depth);
    printf("In C, depth = %d, back from Java\n", depth);
}
```

Se puede llamar a un método de ejemplar (no estático) siguiendo estos tres pasos:

- El método nativo llama a la función JNI `GetObjectClass`. `GetObjectClass` devuelve el objeto class Java al que pertenece el objeto.
- Entonces el método nativo llama a la función JNI `GetMethodID`. `GetMethodID` realiza un búsqueda del método Java dentro de la clase dada. La búsqueda está basada tanto en el nombre del método como en su firma. Si el método no existe, `GetMethodID` devuelve cero (0). Un retorno inmediato desde el método nativo en ese punto causa el lanzamiento de una excepción `NoSuchMethodError` en el código Java.
- Finalmente, el método nativo llama a la función JNI `CallVoidMethod`. `CallVoidMethod` llama a un método de ejemplar que tiene el tipo de retorno como void. Se pasan el objeto, el ID del método y los argumentos reales a `CallVoidMethod`.

Formar el Nombre del Método y su Firma

El JNI realiza un búsqueda simbólica basándose en el nombre y la firma del método. Esto asegura que el mismo método nativo seguirá funcionando después de haber

añadido nuevos métodos a la clase Java correspondiente.

El nombre del método es el nombre del método en formato UTF-8. Especifica el nombre del método para un constructor de una clase encerrando la palabra init entre "< y >".

Observa que el JNI utiliza la firma del método para denotar el tipo de retorno del método Java. Esta firma (I)V, por ejemplo, denota un método Java que toma un argumento del tipo int y tiene un tipo de retorno void. La forma general para una firma de método es:

"(tipos-argumentos)tipo-retorno"

La siguiente tabla sumaria la codificación Java para las firmas:

Tipos de Firmas de la VM de Java

Firma	Tipo Java
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L clases-totalmente-cualificada ;	clase-totalmente-cualificada
[tipo	tipo[]
(tipo-argumento) tipo-retorno	tipo de método

Por ejemplo, el método Prompt.getLine tiene la firma:

(Ljava/lang/String;)Ljava/lang/String;

Prompt.getLine tiene un parámetro, un objeto String, y el tipo del método también es String.

El método Callbacks.main tiene la firma:

([Ljava/lang/String;)V

La firma indica que el método Callbacks.main toma un parámetro, un objeto String, y el tipo del método es void.

Los tipos de arrays son indicados con corchete abierto ([]) seguido por el tipo de los elementos del array.

Utilizar javap para Generar Firmas de Métodos

Para evitar los errores derivados de la firma de métodos manual, se puede utilizar la

herramienta javap para imprimir las firmas de métodos. Por ejemplo, ejecutando:

```
javap -s -p Prompt
```

Se puede obtener la siguiente salida:

```
Compiled from Prompt.java
class Prompt extends java.lang.Object
    /* ACC_SUPER bit set */
{
    private native getLine (Ljava/lang/String;)Ljava/lang/String;
    public static main ([Ljava/lang/String;)V
    <init> ()V
    static <clinit> ()V
}
```

La bandera "-s" informa a javap para que saque las firmas en vez de los tipos Java. La bandera "-p" instruye a javap para que incluya los miembros privados.

Llamar a Métodos Java utilizando los IDs

Cuando se llama a un método en el JNI, se le pasa un ID del método a la función real de llamada. Obtener un ID de un método es una operación que consume muchos recursos. Como se obtiene el ID del método de forma separada de la llamada al método, sólo se necesita realizar esta operación una vez. Así, es posible, obtener primero el ID del método y luego utilizarlo las veces necesarias para invocar al mismo método.

Es importante tener en mente que un ID de método sólo es válido mientras que no se descargue la clase de la se deriva. Una vez que la clase se ha descargado, el ID del método no es válido. Como resultado, si se quiere guardar el ID del método, debemos asegurarnos de mantener viva a una referencia a la clase Java de la que se deriva el ID del método. Mientras exista la referencia a la clase Java (el valor jclass), el código nativo mantendrá una referencia viva a la clase. La página [Referencias Locales y Globales](#) explica como mantener viva una referencia incluso después de que el método nativo retorne y el valor de jclass salga fuera de ámbito.

Pasar Argumentos al Método Java

El JNI proporciona varias formas de pasar argumentos al método Java. La más frecuente, se pasan los argumentos siguiendo al ID del método. También hay dos variaciones de llamadas a métodos que toman argumentos en un formato alternativo. Por ejemplo, la función CallVoidMethodV recibe los argumentos en un va_list y la función CallVoidMethodA espera los argumentos en un array de uniones jvalue. Los tipos del array de uniones jvalue son los siguientes:

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
```

```

    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat    f;
    jdouble   d;
    jobject   l;
} jvalue;

```

Además de la función `CallVoidMethod`, el JNI también soporta funciones de llamada a métodos con tipos de retorno, como `CallBooleanMethod`, `CallIntMethod`, etc. El tipo de retorno de la función de llamada a método debe corresponder con el tipo del método Java que se desea invocar.

Llamar a Métodos Estáticos

Se puede llamar a métodos estáticos Java de forma similar a como se llama a métodos de ejemplar. Se deben seguir estos pasos:

- Obtener el ID del método utilizando la función JNI `GetStaticMethodID` en vez la función `GetMethodID`.
- Pasar la clase, el ID del método y los argumentos a la familia de funciones de llamadas a métodos estáticos: `CallStaticVoidMethod`, `CallStaticBooleanMethod`, etc.

Si se comparan las funciones de llamada para métodos de ejemplar y estáticos, veremos que las funciones de llamada a métodos de ejemplar reciben el `object`, en vez de la clase, como el segundo argumento, seguido por el argumento `JNIEnv`. Por ejemplo, supongamos, que hemos añadido un método estático:

```
static int incDepth(int depth) {return depth + 1};
```

dentro de `Callback.java`. Podremos llamar a este método estático `incDepth` desde `Java_Callback_nativeMethod` utilizando las siguientes funciones JNI:

```

JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetStaticMethodID(env, cls, "incDepth", "(I)I");
    if (mid == 0)
        return;
    depth = (*env)->CallStaticIntMethod(env, cls, mid, depth);
}

```

Llamar a Métodos de Ejemplar de una Superclase

Se puede llamar a métodos definidos en una superclase que se han sobrescrito en la clase a la que pertenece el objeto. El JNI proporciona un conjunto de funciones `CallNonvirtual< type>Method` para este propósito. Para llamar a un método de ejemplar de una superclase, se debe hacer lo siguiente:

- Obtener el ID del método de la superclase utilizando `GetMethodID` en vez de `GetStaticMethodID`.
- Pasar el objeto, la superclase, el ID del método, y los argumentos a la familia de funciones de llamadas no virtuales: `CallNonvirtualVoidMethod`, `CallNonvirtualBooleanMethod`, etc.

Será raro que necesitemos invocar a métodos de ejemplar de una superclase. Esta facilidad es similar a llamar al método de una clase, digamos `f`, utilizando:

```
super.f();
```

en el lenguaje Java.

Ozito

Acceder a Campos Java

El JNI proporciona funciones que los métodos nativos utilizan para obtener y seleccionar campos Java. Se pueden obtener y seleccionar tanto campos de ejemplar como campos de clase (estáticos). Similarmente al acceso a los métodos, se utiliza un conjunto de funciones JNI para acceder a campos de ejemplar y otro conjunto para acceder a campos estáticos.

Nuestro programa de ejemplo, [FieldAccess.java](#), contiene una clase con un campo Integer estático `si` y un campo String de ejemplar `s`. El programa de ejemplo llama al método nativo `accessFields`, que imprime los valores de esos dos campos y modifica esos valores. Para verificar que se han modificado realmente los valores de los campos, imprimimos de nuevo sus valores desde la aplicación Java después de que haya retornado el método nativo.

Procedimiento para Acceder a Campos Java

Para obtener y seleccionar campos Java desde un método nativo, se debe hacer lo siguiente:

- Obtener el identificador para ese campo dentro de su clase, el nombre y el tipo de firma. Por ejemplo, en [FieldAccess.c](#), obtenemos el identificador para el campo estático entero `si` de esta forma:

```
fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
```

y obtenemos el identificador para el campo string de ejemplar `s`, de esta forma:

```
fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");
```

- Utilizar una de las varias funciones JNI para obtener o modificar el campo especificado por el identificador del campo. Para obtener el valor del campo, se pasa la clase a una de las funciones de acceso a campos apropiada. Por ejemplo, en `FieldAccess.c`, utilizamos `GetStaticIntField` para obtener el valor del campo estático Integer `si`, de esta forma:

```
si = (*env)->GetStaticIntField(env, cls, fid);
```

Utilizamos la función `GetObjectField` para obtener el valor del campo de ejemplar String `s`:

```
jstr = (*env)->GetObjectField(env, obj, fid);
```

Igual que hicimos cuando llamamos a un método Java, dividimos el costo de la búsqueda del campo utilizando un proceso de dos pasos. Primero obtenemos el ID del campo, luego utilizamos el ID del campo para acceder al propio campo. El ID de campo sólo identifica a un campo en una clase dada. Igualmente que los IDs de

método, un ID de campo permanece válido hasta que la clase de la que se ha derivado es descargada.

Firmas de Campos

Las firmas de campo se especifican siguiendo el mismo esquema de codificación que las firmas de métodos. La forma general de una firma de campo es:

```
"tipo_de_campo"
```

La firma del campo es un símbolo codificado para el tipo del campo, encerrado entre comillas (""). Los símbolos de campos son los mismos que los símbolos de los argumentos en la firma de métodos. Esto es, un campo Integer se representa con "I", un campo float se representa con "F", un campo double "D" y un campo booleano con "Z", etc.

La firma para un objeto Java, como un String, empieza con la letra L, seguida por la clase totalmente cualificada del objeto, y terminada con un punto y coma (;). Así, se formaría la firma un campo String, como c.s en FieldAccess.java, de esta forma:

```
"Ljava/lang/String;"
```

Los arrays se indican con un corchete abierto ([]) seguido por el tipo del array. Por ejemplo, se designaría un array de Integer de esta forma:

```
"[I"
```

Puedes referirte a la tabla de la [página anterior](#) que resume la codificación para las firmas Java y sus correspondientes tipos.

Se puede utilizar javap con la opción "-s" para generar firmas de campos de una clase. Por ejemplo, si se ejecuta:

```
javap -s -p FieldAccess
```

Este da una salida que contiene las siguientes firmas de campos:

```
...
static si I
s Ljava/lang/String;
...
```

Capturar y Lanzar Excepciones

Cuando se lanza una excepción en Java, la Máquina Virtual busca automáticamente el manejador de excepción más cercano y despliega la pila si es necesario. La ventaja de este estilo de manejo de excepciones es que libera al programador de tener cuidado y del manejo inusual de los errores situados en cada operación del programa. En su lugar, la Máquina Virtual Java propaga automáticamente las condiciones del error a la posición (la clausula catch de Java) que maneja el mismo tipo de condiciones de error de una forma centralizada.

Mientras que otros lenguajes como C++ soportan una noción similar de manejo de excepciones, no existe una manera uniformada de lanzar y capturar excepciones en los lenguajes nativos. Por lo tanto, el JNI requiere que se comprueben las posibles excepciones después de llamar a sus funciones. El JNI también proporciona funciones que permiten a los métodos nativos lanzar excepciones Java. Estas excepciones pueden ser manejadas por otras partes del código nativo o por la Máquina Virtual Java. Después de que el código nativo lance o capture una excepción, puede eliminar las excepciones pendientes para que el cálculo continúe, o puede lanzar otra excepción para un manejador exterior.

Muchas funciones JNI pueden lanzar excepciones. Por ejemplo, la función `GetFieldID` descrita en la [página anterior](#) lanza un `NoSuchFieldError` si el campo especificado no existe. Para simplificar el chequeo de errores, la mayoría de las funciones JNI utilizan una combinación de códigos de error y excepciones Java para informar de las condiciones de error. Por ejemplo, se podría chequear si el campo `jfieldID` devuelto por `GetFieldID` es cero (0) en vez de llamar a la función JNI `ExceptionOccurred`. Cuando el resultado de `GetFieldID` no es cero (0), se puede estar seguro de que no hay una excepción pendiente.

El resto de la página ilustra cómo capturar y lanzar excepciones en código nativo. El código de ejemplo está en [CatchThrow.java](#).

El método `CatchThrow.main` llama al método nativo. El método nativo, definido en [CatchThrow.c](#), primero llama al método `CatchThrow.callback` Java, de esta forma:

```
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "()V");
jthrowable exc;
if (mid == 0)
    return;
(*env)->CallVoidMethod(env, obj, mid);
```

Observa que como `CallVoidMethod` lanza una `NullPointerException`, el código nativo puede detectar esta excepción después de que `CallVoidMethod` retorne llamando al método `ExceptionOccurred`:

```
exc = (*env)->ExceptionOccurred(env);
if (exc) {
```

Como se puede ver es muy sencillo capturar y lanzar una excepción. En nuestro ejemplo, no hacemos mucho con la excepción en `CatchThrow.c` excepto utilizar el método `ExceptionDescribe` para sacar un mensaje de depurado. El método nativo lanza entonces una `IllegalArgumentException`. Es esta excepción `IllegalArgumentException` es la que verá el código Java que llamó al método nativo.

```
(*env)->ExceptionDescribe(env);
(*env)->ExceptionClear(env);

newExcCls = (*env)->FindClass(env, "java/lang/IllegalArgumentException");
if (newExcCls == 0) /* Unable to find the new exception class, give up. */
    return;
(*env)->ThrowNew(env, newExcCls, "thrown from C code");
```

La función `ThrowNew` construye un objeto exception desde una clase exception dada y un string de mensaje y postea la excepción en el thread actual.

Observa que es extremadamente importante comprobar, manejar y limpiar las excepciones pendientes antes de llamar a las siguientes funciones JNI. Llamar arbitrariamente a funciones JNI con una excepción pendiente podría generar resultados inesperados. Solo se pueden llamar de forma segura a unas pocas funciones JNI cuando existe una excepción pendiente. Estas funciones son: `ExceptionOccurred`, `ExceptionDescribe`, y `ExceptionClear`.

Ozito

Referencias Locales y Globales

Hasta ahora, hemos utilizado tipos de datos como jobject, jclass, y jstring para denotar referencias a objetos Java. Sin embargo, el JNI crea referencias para todos los argumentos pasados a los métodos nativos, así como de los objetos devueltos desde funciones JNI.

Las referencias sirven para evitar que los objetos Java sean recolectados por el recolector de basura. Por defecto, el JNI crea referencias locales porque éstas aseguran que la Máquina Virtual pueda liberar eventualmente los objetos Java. Las referencias locales se vuelven inválidas cuando la ejecución del programa retorna desde el método nativo en el que se creó. Por lo tanto, un método nativo no debería almacenar una referencia local y esperar utilizarla en llamadas subsecuentes.

Por ejemplo, el siguiente programa, que es una variación del método nativo de [FieldAccess.c](#), erróneamente captura el ID del campo Java para no tener que buscarlo repetidamente basándose en el nombre y la firma en cada llamada:

```
/* This code is illegal */
static jclass cls = 0;
static jfieldID fld;

JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
    ...
    if (cls == 0) {
        cls = (*env)->GetObjectClass(env, obj);
        if (cls == 0)
            ... /* error */
        fld = (*env)->GetStaticFieldID(env, cls, "si", "I");
    }
    ... /* access the field using cls and fld */
}
```

Este código es ilegal porque la referencia local devuelta desde GetObjectClass es sólo válida antes de que retorne el método nativo. Cuando una aplicación Java llama al método nativo Java_FieldAccess_accessField una segunda vez, el método nativo trata de utilizar una referencia no válida. Esto acabará en un resultado erróneo o un cuelgue de la Máquina Virtual.

Se puede solucionar este problema creando una referencia global. Una referencia global permanecerá válida hasta que se libere explícitamente. El siguiente código reescribe el programa anterior y utiliza correctamente la referencia global para capturar el ID del objeto:

```

/* This code is OK */
static jclass cls = 0;
static jfieldID fld;

JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
    ...
    if (cls == 0) {
        jclass cls1 = (*env)->GetObjectClass(env, obj);
        if (cls1 == 0)
            ... /* error */
        cls = (*env)->NewGlobalRef(env, cls1);
        if (cls == 0)
            ... /* error */
        fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
    }
    ... /* access the field using cls and fid */
}

```

Una referencia global evita que la Máquina Virtual descargue la clase Java, y por lo tanto también asegura que el ID del campo permanezca válido, como se explicó en [Acceder a Campos Java](#). Sin Embargo, el código nativo debe llamar a `DeleteGlobalRefs` cuando no necesite acceder más a la referencia global. De otro modo, la Máquina Virtual nunca descargará el objeto correspondiente.

En la mayoría de los casos, los programadores nativos relegan en la VM la liberación de las referencias locales después de retornar del método nativo. Sin embargo, en ciertas situaciones, el código nativo podría necesitar llamar a la función `DeleteLocalRef` para borrar explícitamente una referencia local. Estas situaciones son:

- Podríamos saber que estamos manteniendo la única refencia a un objeto Java de gran tamaño y no queremos esperar hasta que el método nativo retorne antes de que el recolector de basura reclame ese objeto. Por ejemplo, en el siguiente fragmento de código, el colector de basura podría liberar el objeto Java referenciado por `lref` cuando está dentro de `lengthyComputation`:

```

lref = ... /* a large Java object */

... /* last use of lref */
(*env)->DeleteLocalRef(env, lref);

lengthyComputation(); /* may take some time */

return; /* all local refs will now be freed */

```

```
}
```

- Podríamos necesitar crear un gran número de referencias locales en una sola llamada a un método nativo. Esto podría resultar en una sobrecarga en la tabla de referencias locales del JNI. Es una buena idea borrar aquellas referencias locales que no se van a necesitar. Por ejemplo, en el siguiente fragmento de código, el código nativo itera a través de un array potencialmente grande arr que contiene strings Java. Después de cada iteración, el programa puede liberar la referencia local del elemento string:

```
for(i = 0; i < len; i++) {  
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);  
    ... /* processes jstr */  
    (*env)->DeleteLocalRef(env, jstr); /* no longer needs jstr */  
}
```

Threads y Métodos Nativos

Java es un sistema multi-thread, por lo tanto los métodos nativos deben ser seguros con los threads, A menos que tengamos información extra (por ejemplo, si el método nativo está sincronizado), debemos asumir que pueden existir varios threads de control ejecutándose sobre un mismo método nativo en un momento dado. Por lo tanto, los métodos nativos no deberían modificar variables globales sensibles de una forma desprotegida. Esto es, deben compartir y coordinar su acceso a variables de ciertas secciones críticas de código.

Antes de leer esta sección, deberías estar familiarizado con los conceptos de threads de control y programas multi-threads. [Threads de Control](#) cubre la programación de threads. En particular, la página [Programas Multi-Thread](#) cubre elementos relacionados con la escritura de programas que contienen varios threads, incluyendo cómo sincronizarlos.

Los Threads y el JNI

El puntero al interface JNI (JNIEnv *) sólo es válido en el thread actual. Se debe pasar el puntero al intercace de un thread a otro, o guardar el puntero al interface y utilizarlo en varios threads. La máquina virtual Java pasa el mismo puntero al interface en llamadas sucesivas a un método nativo desde el mismo thread, pero diferentes threads pasan un diferente puntero al interface a los métodos nativos.

No se deben pasar referencias locales de un thread a otro. En particular, una referencia local se podría volver no válida antes de que otro thread tenga la posibilidad de utilizarla. Se deben convertir siempre en referencias globales cuando haya alguna duda sobre que una referencia a un objeto pueda ser utilizada por threads diferentes.

Chequea el uso de variables globales cuidadosamente. Varios threads podrían acceder a las variables globales al mismo tiempo. Asegurate de que pones los bloqueos necesarios para asegurar la seguridad.

Sincronización de Threads en Métodos Nativos.

El JNI proporciona dos funciones de sincronización que permiten implementar bloques sincronizados. En Java, son implementados utilizando la sentencia synchronized. Por ejemplo:

```
synchronized (obj) {  
    ...                               /* synchronized block */  
}
```

La Máquina Virtual Java garantiza que un thread deba adquirir el monitor asociado con el objeto Java obj antes de poder ejecutar las sentencias del bloque. Por lo

tanto, en un momento dado, sólo puede haber un thread ejecutándose dentro del bloque sincronizado.

El código nativo puede realizar una sincronización equivalente de objetos utilizando las funciones del JNI `MonitorEnter` y `MonitorExit`. Por ejemplo:

```
(*env)->MonitorEnter(obj);  
... /* synchronized block */  
(*env)->MonitorExit(obj);
```

Un thread debe introducir el monitor asociado con `obj` antes de poder continuar con la ejecución. El monitor contiene un contador que señala cuantas veces ha sido introducido por un thread dado. `MonitorEnter` incrementa el contador cuando el thread entra un monitor que ya ha sido introducido. `MonitorExit` decrementa el contador. Si el contador alcanza el valor 0, otros threads pueden introducir el monitor.

Wait y Notify

Otro mecanismo de sincronización de threads es `Object.wait`, `Object.notify`, y `Object.notifyAll`. El JNI no soporta directamente estas funciones. Sin embargo, un método nativo puede seguir el [mecanismo de llamada a métodos](#) para invocar a estos métodos.

Invocar a la Máquina Virtual Java

En el JDK 1.1, la Máquina Virtual Java está almacenada como una librería compartida (o librería de enlace dinámico en Win32). Podemos introducir la VM en nuestra aplicación nativa, enlazando la aplicación con la librería. El JNI soporta un API de "Invocation" invocación que permite cargar, inicializar, y llamar a la Máquina Virtual Java. De hecho, la forma normal de arrancar el intérprete Java, java, no es más que un sencillo programa C que analiza los argumentos de la línea de comandos y llama a la Máquina Virtual Java a través del API Invocation.

Llamar a la Máquina Virtual Java

Para ilustrar como llamar a la Máquina Virtual Java, escribiremos un programa C que la invoque y llame al método Prog.main definido en Prog.java:

```
public class Prog {  
    public static void main(String[] args) {  
        System.out.println("Hello World" + args[0]);  
    }  
}
```

El código C de [invoke.c](#) empieza con una llamada a JNI_GetDefaultJavaVMInitArgs para obtener los valores para la inicialización por defecto (tamaño de la pila, etc) Luego llama a JNI_CreateJavaVM para cargar e inicializar la Máquina Virtual. JNI_CreateJavaVM rellena dos valores de retorno:

- jvm se refiere la Máquina Virtual creada. Se puede utilizar para destruir la Máquina Virtual, por ejemplo.
- env es un puntero a interface JNI que el thread actual puede utilizar para acceder a las características Java, como llamar a un método Java.

Observa que después de que JNI_CreateJavaVM haya retornado satisfactoriamente, el thread nativo, se introduce a sí mismo en la Máquina Virtual Java y por lo tanto se está ejecutando como si fuera un método nativo. La única diferencia es que no existe el concepto de retorno de la Máquina Virtual Java. Por lo tanto, las [referencias locales](#) creadas después no serán liberadas hasta que se llame a DestroyJavaVM.

Una vez que se ha creado la Máquina Virtual Java, se pueden realizar llamadas normales del JNI, por ejemplo a Prog.main. DestroyJavaVM intenta descargar la MV. (En el JDK 1.1 la Máquina Virtual Java no puede ser descargada; por lo tanto DestroyJavaVM siempre devuelve un código de error.)

Se necesita compilar y enlazar invoke.c con las librerías Java distribuidas con el JDK 1.1. En Solaris, se puede utilizar el siguiente comando para compilar y enlazar invoke.c:

```
cc -I<where jni.h is> -L<where libjava.so is> -ljava invoke.c
```

En Win32 con Microsft Visual C++ 4.0, la línea de comandos es:

```
cl -I<where jni.h is> -MT invoke.c -link <where javai.lib is>\javai.lib
```

Aquellos que trabajen en el entorno MacOS deberán referirse al API JManager, que forma parte del [MacOS Runtime for Java \(MRJ\)](#). Se utiliza este API para incluir aplicaciones Java en aplicaciones MAC.

Al ejecutar el programa resultante desde la línea de comandos, es posible que obtengamos el siguiente mensaje de error:

```
Unable to initialize threads: cannot find class java/lang/Thread
Can't create Java VM
```

Este mensaje de error indica que se ha seleccionado erróneamente el valor para la variable `vm_args.classpath`. Por otro lado, si el error indica que no puede encontrar `libjava.so` (en Solaris) o `javai.dll` (en Win32), añadiremos `libjava.so` a nuestro `LD_LIBRARY_PATH` en Solaris, o `javai.dll` en nuestro path de ejecutables en Win32. Si el programa muestra un error diciendo que no puede encontrar la clase `Prog`, debemos asegurarnos de que el directorio que contiene `Prog.class` también está en la variable `vm_args.classpath`.

Añadir Threads Nativos

El API Invocation también permite añadir threads nativos a una VM Java que se esté ejecutando y convertir los propios threads en threads Java. Esto requiere que la Máquina Virtual Java utilice internamente threads nativos. En el JDK 1.1, esta característica sólo funciona en Win32. La versión Solaris de la Máquina Virtual Java utiliza soporte de threads de nivel de usuario y por lo tanto es incapaz de añadir threads nativos. Una futura versión del JDK para solaris soportará threads nativos.

Por lo tanto, nuestro programa de ejemplo, [attach.c](#), sólo funcionará en Win32. Es una variación de `invoke.c`. En lugar de llamar a `Prog.main` en el thread principal, el código nativo expande cinco threads y espera a que finalicen antes de destruir la Máquina Virtual Java. Cada thread se añade a sí mismo a la Máquina Virtual Java, llama al método `Prog.main`, y finalmente se borra de la Máquina Virtual antes de terminar. Observa que el tercer argumento de `AttachCurrentThread` está reservado y siempre debe ser `NULL`.

Cuando se llama a `DetachCurrentThread`, todas las referencias locales pertenecientes al thread actual serán liberadas.

Limitaciones del API Invocation en el JDK 1.1

Como se mencionó anteriormente, existe un número de limitaciones en la implementación del API Invocation en el JDK 1.1.

- La implementación de threads a nivel de usuario en Solaris requiere que la

Máquina Virtual Java redireccione ciertas llamadas al sistema. El juego de llamadas redireccionadas realmente incluye: read, readv, write, writev, getmsg, putmsg, poll, open, close, pipe, fcntl, dup, create, accept, recv, send, etc. Esto podría causar efectos indeseados en aplicaciones nativas, que también dependan de llamadas al sistema.

- No se pueden añadir threads nativos a la Máquina Virtual Java bajo Solaris. AttachCurrentThread simplemente falla en Solaris (a menos que se llame desde el thread principal que creó la Máquina Virtual).
- No se puede destruir la Máquina Virtual Java sin terminar el proceso. La llamada a DestroyJavaVM simplemente devuelve un código de error.

Estos problemas serán corregidos en futuras versiones del JDK.

Ozito

Programación JNI en C++

El JNI proporciona un interface sencillo para programadores C++. El fichero jni.h contiene un conjunto de funciones C++ para que el programador del método nativo pueda escribir solo:

```
jclass cls = env->FindClass("java/lang/String");
```

en lugar de :

```
jclass cls = (*env)->FindClass(env, "java/lang/String");
```

El nivel extra de indirección de env y el argumento env a FindClass están ocultos para el programador. El compilador C++ expandirá las llamadas a las funciones miembros de C++, y por lo tanto el código resultante es exactamente el mismo.

El fichero jni.h también define un conjunto de clases inútiles de C++ para forzar la relación de tipado entre las diferentes variaciones del tipo jobject:

```
class _jobject {};  
class _jclass : public _jobject {};  
class _jthrowable : public _jobject {};  
class _jstring : public _jobject {};  
... /* more on jarray */
```

```
typedef _jobject *jobject;  
typedef _jclass *jclass;  
typedef _jthrowable *jthrowable;  
typedef _jstring *jstring;  
... /* more on jarray */
```

Por lo tanto, el compilador C++ puede detectar si se le pasa, un jobject a GetMethodID, por ejemplo:

```
jmethodID GetMethodID(jclass clazz, const char *name,  
                      const char *sig);
```

ya que GetMethodID espera un jclass. En C, jclass es simplemente lo mismo que jobject:

```
typedef jobject jclass;
```

Por lo tanto, un compilador C no puede detectar que se le ha pasado erróneamente un jobject en vez de un jclass.

La adición del tipos seguros en C++ tiene un pequeño inconveniente. Recordemos de [Acceder a Arrays Java](#) que en C, podemos crear un String Java desde un array de Strings y directamente asignar el resultado a un jstring:

```
jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
```

Sin embargo, en C++, necesitamos insertar una conversión explícita:

```
jstring jstr = (jstring)env->GetObjectArrayElement(arr, i);
```

porque jstring es un subtipo de jobject, el tipo de retorno de GetObjectArrayElement.

Ozito

JDBC - Acceso a Bases de Datos

JDBCtm fue diseñado para mantener sencillas las cosas sencillas. Esto significa que el API JDBC hace muy sencillas las tareas diarias de una base de datos, como una simple sentencia SELECT. Esta sección nos llevará a través de ejemplos que utilizan el JDBC para ejecutar sentencias SQL comunes, para que podamos ver lo sencilla que es la utilización del API JDBC básico.

Esta sección está dividida a su vez en dos secciones:

[JDBC Basico](#) que cubre el API JDBC 1.0, que está incluido en el JDKtm 1.1. La segunda parte cubre el API JDBC 2.0 API, que forma parte de la versión 1.2 del JDK. También describe brevemente las extensiones del API JDBC, que, al igual que otras extensiones estándar, serán liberadas independientemente.

Al final de esta primera sección, sabremos como utilizar el API básico del JDBC para crear tablas, insertar valores en ellas, pedir tablas, recuperar los resultados de las peticiones y actualizar las tablas. En este proceso, aprenderemos como utilizar las sentencias sencillas y sentencias preparadas, y veremos un ejemplo de un procedimiento almacenado. También aprenderemos como realizar transacciones y como capturar excepciones y avisos. En la última parte de este sección veremos como crear un Applet.

[Nuevas Características en el API JDBC 2.0](#) nos enseña como mover el cursor por una hoja de resultados, cómo actualizar la hoja de resultados utilizando el API JDBC 2.0, y como hacer actualizaciones batch. También conoceremos los nuevos tipos de datos de SQL3 y como utilizarlos en aplicaciones escritas en Javatm. La parte final de esta lección entrega una visión de la extensión del API JDBC, con caracterísitas que se aprovechan de la tecnologías JavaBeanstm y Enterprise JavaBeanstm.

Esta sección no cubre cómo utilizar el API metadata, que se utiliza en programas más sofisticados, como aplicaciones que deban descubrir y presentar dinámicamente la estructura de una base de datos fuente.

Para empezar...

Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos:

1. Instalar Java y el JDBC en nuestra máquina.

Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el JDBC.. El código de ejemplo de demostración del API del JDBC 1.0 fue escrito para el JDK 1.1 y se ejecutará en cualquier versión de la plataforma Java compatible con el JDK 1.1, incluyendo el JDK1.2. Teniendo en cuenta que los ejemplos del API del JDBC 2.0 requieren el JDK 1.2 y no se podrán ejecutar sobre el JDK 1.1.

Podrás encontrar la última versión del JDK en la siguiente dirección::

<http://java.sun.com/products/JDK/CurrentRelease>

2. Instalar un driver en nuestra máquina.

Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuestra máquina; no se necesita ninguna configuración especial.

El driver "puente JDBC-ODBC" no es tan sencillo de configurar. Si descargamos las versiones Solaris o Windows de JDK 1.1, automáticamente obtendremos una versión del driver Bridge JDBC-ODBC, que tampoco requiere una configuración especial. Si embargo, ODBC, si lo necesita. Si no tenemos ODBC en nuestra máquina, necesitaremos preguntarle al vendedor del driver ODBC sobre su instalación y configuración.

3. Instalar nuestro Controlador de Base de Datos si es necesario.

Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

Seleccionar una Base de Datos

A lo largo de la sección asumiremos que la base de datos COFFEEBREAK ya existe. (crear una base de datos no es nada difícil, pero requiere permisos especiales y normalmente lo hace un administrador de bases de datos). Cuando creamos las tablas utilizadas como ejemplos en este tutorial, serán la base de datos por defecto. Hemos mantenido un número pequeño de tablas para mantener las cosas manejables.

Supongamos que nuestra base de datos está siendo utilizada por el propietario de un pequeño café llamado "The Coffee Break", donde los granos de café se venden por kilos y el café líquido se vende por tazas. Para mantener las cosas sencillas, también supondremos que el propietario sólo necesita dos tablas, una para los tipos de café y otra para los suministradores.

Primero veremos como abrir una conexión con nuestro controlador de base de datos, y luego, ya que JDBC puede enviar código SQL a nuestro controlador, demostraremos algún código SQL. Después, veremos lo sencillo que es utilizar JDBC para pasar esas sentencias SQL a nuestro controlador de bases de datos y procesar los resultados devueltos.

Este código ha sido probado en la mayoría de los controladores de base de datos. Sin embargo, podríamos encontrar algunos problemas de compatibilidad si utilizamos antiguos drivers ODB con el puente JDBC.ODBC.

Establecer una Conexión

Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos: (1) cargar el driver y (2) hacer la conexión.

Cargar los Drivers

Cargar el driver o drivers que queremos utilizar es muy sencillo y sólo implica una línea de código. Si, por ejemplo, queremos utilizar el puente JDBC-ODBC, se cargaría la siguiente línea de código:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es jdbc.DriverXYZ, cargaríamos el driver con esta línea de código:

```
Class.forName("jdbc.DriverXYZ");
```

No necesitamos crear un ejemplar de un driver y registrarlo con el DriverManager porque la llamada a Class.forName lo hace automáticamente. Si hubiéramos creado nuestro propio ejemplar, crearíamos un duplicado innecesario, pero no pasaría nada.

Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

Hacer la Conexión

El segundo paso para establecer una conexión es tener el driver apropiado conectado al controlador de base de datos. La siguiente línea de código ilustra la idea general:

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

Este paso también es sencillo, lo más duro es saber qué suministrar para url. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con jdbc:odbc:. el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada "Fred," por ejemplo, nuestro URL podría ser jdbc:odbc:Fred. En lugar de "myLogin" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de "myPassword" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre "Fernando" y la password of "J8," estas dos líneas de código establecerán una conexión:

```
String url = "jdbc:odbc:Fred";  
Connection con = DriverManager.getConnection(url, "Fernando", "J8");
```

Si estamos utilizando un puente JDBC desarrollado por una tercera parte, la documentación nos dirá el subprotocolo a utilizar, es decir, qué poner después de jdbc: en la URL. Por ejemplo, si el desarrollador ha registrado el nombre "acme" como el subprotocolo, la primera y segunda parte de la URL de JDBC serán

`jdbc:acme:`. La documentación del driver también nos dará las guías para el resto de la URL del JDBC. Esta última parte de la URL suministra información para la identificación de los datos fuente.

Si uno de los drivers que hemos cargado reconoce la URL suministrada por el método `DriverManager.getConnection`, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase `DriverManager`, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface `Driver`, y el único método de `DriverManager` que realmente necesitaremos conocer es `DriverManager.getConnection`.

La conexión devuelta por el método `DriverManager.getConnection` es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, `con` es una conexión abierta, y se utilizará en los ejemplos posteriores.

Seleccionar Tablas

Crear una Tabla

Primero, crearemos una de las tablas de nuestro ejemplo. Esta tabla, COFFEES, contiene la información esencial sobre los cafés vendidos en "The Coffee Break", incluyendo los nombres de los cafés, sus precios, el número de libras vendidas la semana actual, y el número de libras vendidas hasta la fecha. Aquí puedes ver la tabla COFFEES, que describiremos más adelante:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

La columna que almacena el nombre del café es COF_NAME, y contiene valores con el tipo VARCHAR de SQL y una longitud máxima de 32 caracteres. Como utilizamos nombres diferentes para cada tipo de café vendido, el nombre será un único identificador para un café particular y por lo tanto puede servir como clave primaria. La segunda columna, llamada SUP_ID, contiene un número que identifica al suministrador del café; este número será un tipo INTEGER de SQL. La tercera columna, llamada PRICE, almacena valores del tipo FLOAT de SQL porque necesita contener valores decimales. (Observa que el dinero normalmente se almacena en un tipo DECIMAL o NUMERIC de SQL, pero debido a las diferencias entre controladores de bases de datos y para evitar la incompatibilidad con viejas versiones de JDBC, utilizamos el tipo más estándar FLOAT.) La columna llamada SALES almacena valores del tipo INTEGER de SQL e indica el número de libras vendidas durante la semana actual. La columna final, TOTAL, contiene otro valor INTEGER de SQL que contiene el número total de libras vendidas hasta la fecha.

SUPPLIERS, la segunda tabla de nuestra base de datos, tiene información sobre cada uno de los suministradores:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Las tablas COFFEES y SUPPLIERS contienen la columna SUP_ID, lo que significa que estas dos tablas pueden utilizarse en sentencias SELECT para obtener datos basados en la información de ambas tablas. La columna SUP_ID es la clave primaria de la tabla SUPPLIERS, y por lo tanto, es un identificador único para cada uno de los suministradores de café. En la tabla COFFEES, SUP_ID es llamada clave extranjera. (Se puede pensar en una clave extranjera en el sentido en que es

importada desde otra tabla). Observa que cada número SUP_ID aparece sólo una vez en la tabla SUPPLIERS; esto es necesario para ser una clave primaria. Sin embargo, en la tabla COFFEES, donde es una clave extranjera, es perfectamente correcto que haya números duplicados de SUP_ID porque un suministrador puede vender varios tipos de café. Más adelante en este capítulo podremos ver cómo utilizar claves primarias y extranjeras en una sentencia SELECT.

La siguiente sentencia SQL crea la tabla COFFEES. Las entradas dentro de los paréntesis exteriores consisten en el nombre de una columna seguido por un espacio y el tipo SQL que se va a almacenar en esa columna. Una coma separa la entrada de una columna (que consiste en el nombre de la columna y el tipo SQL) de otra. El tipo VARCHAR se crea con una longitud máxima, por eso toma un parámetro que indica la longitud máxima. El parámetro debe estar entre paréntesis siguiendo al tipo. La sentencia SQL mostrada aquí, por ejemplo, especifica que los nombres de la columna COF-NAME pueden tener hasta 32 caracteres de longitud:

```
CREATE TABLE COFFEES
    (COF_NAME VARCHAR(32),
    SUP_ID INTEGER,
    PRICE FLOAT,
    SALES INTEGER,
    TOTAL INTEGER)
```

Este código no termina con un terminador de sentencia de un controlador de base de datos, que puede variar de un controlador a otro. Por ejemplo, Oracle utiliza un punto y coma (;) para finalizar una sentencia, y Sybase utiliza la palabra go. El driver que estamos utilizando proporcionará automáticamente el terminador de sentencia apropiado, y no necesitaremos introducirlo en nuestro código JDBC.

Otra cosa que debíamos apuntar sobre las sentencias SQL es su forma. En la sentencia CREATE TABLE, las palabras clave se han imprimido en letras mayúsculas, y cada ítem en una línea separada. SQL no requiere nada de esto, estas convenciones son sólo para una fácil lectura. El estándar SQL dice que las palabras claves no son sensibles a las mayúsculas, así, por ejemplo, la anterior sentencia SELECT puede escribirse de varias formas. Y como ejemplo, estas dos versiones son equivalentes en lo que concierne a SQL:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE "Washington"
select First_Name, Last_Name from Employees where
Last_Name like "Washington"
```

Sin embargo, el material entre comillas si es sensible a las mayúsculas: en el nombre "Washington", "W" debe estar en mayúscula y el resto de las letras en minúscula.

Los requerimientos pueden variar de un controlador de base de datos a otro cuando se trata de nombres de identificadores. Por ejemplo, algunos controladores,

requieren que los nombres de columna y de tabla seán exactamente los mismos que se crearon en las sentencias CREATE y TABLE, mientras que otros controladores no lo necesitan. Para asegurarnos, utilizaremos mayúsculas para identificadores como COFFEES y SUPPLIERS porque así es como los definimos.

Hasta ahora hemos escrito la sentencia SQL que crea la tabla COFFEES. Ahora le pondremos comillas (crearemos un string) y asignaremos el string a la variable createTableCoffees para poder utilizarla en nuestro código JDBC más adelante. Como hemos visto, al controlador de base de datos no le importa si las líneas están divididas, pero en el lenguaje Java, un objeto String que se extienda más allá de una línea no será compilado. Consecuentemente, cuando estamos entregando cadenas, necesitamos encerrar cada línea entre comillas y utilizar el signo más (+) para concatenarlas:

```
String createTableCoffees = "CREATE TABLE COFFEES " +  
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +  
    "SALES INTEGER, TOTAL INTEGER)";
```

Los tipos de datos que hemos utilizado en nuestras sentencias CREATE y TABLE son tipos genéricos SQL (también llamados tipos JDBC) que están definidos en la clase java.sql.Types. Los controladores de bases de datos generalmente utilizan estos tipos estándares, por eso cuando llegue el momento de probar alguna aplicación, sólo podremos utilizar la aplicación CreateCoffees.java, que utiliza las sentencias CREATE y TABLE. Si tu controlador utiliza sus propios nombres de tipos, te suministraremos más adelante una aplicación que hace eso.

Sin embargo, antes de ejecutar alguna aplicación, veremos lo más básico sobre el JDBC.

Crear sentencias JDBC

Un objeto Statement es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto Statement y lo ejecutamos, suministrando el método SQL apropiado con la sentencia SQL que queremos enviar. Para una sentencia SELECT, el método a ejecutar es executeQuery. Para sentencias que crean o modifican tablas, el método a utilizar es executeUpdate.

Se toma un ejemplar de una conexión activa para crear un objeto Statement. En el siguiente ejemplo, utilizamos nuestro objeto Connection: con para crear el objeto Statement: stmt:

```
Statement stmt = con.createStatement();
```

En este momento stmt existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el método que utilizaremos para ejecutar stmt. Por ejemplo, en el siguiente fragmento de código, suministramos executeUpdate con la sentencia SQL del ejemplo anterior:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +
```

```
"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +  
"SALES INTEGER, TOTAL INTEGER)");
```

Como ya habíamos creado un String con la sentencia SQL y lo habíamos llamado `createTableCoffees`, podríamos haber escrito el código de esta forma alternativa:

```
stmt.executeUpdate(createTableCoffees);
```

Ejecutar Sentencias

Utilizamos el método `executeUpdate` porque la sentencia SQL contenida en `createTableCoffees` es una sentencia DDL (data definition language). Las sentencias que crean, modifican o eliminan tablas son todas ejemplos de sentencias DDL y se ejecutan con el método `executeUpdate`. Como se podría esperar de su nombre, el método `executeUpdate` también se utiliza para ejecutar sentencias SQL que actualizan una tabla. En la práctica `executeUpdate` se utiliza más frecuentemente para actualizar tablas que para crearlas porque una tabla se crea sólo una vez, pero se puede actualizar muchas veces.

El método más utilizado para ejecutar sentencias SQL es `executeQuery`. Este método se utiliza para ejecutar sentencias `SELECT`, que comprenden la amplia mayoría de las sentencias SQL. Pronto veremos como utilizar este método.

Introducir Datos en una Tabla

Hemos visto como crear la tabla `COFFEES` especificando los nombres de columnas y los tipos de datos almacenados en esas columnas, pero esto sólo configura la estructura de la tabla. La tabla no contiene datos todavía. Introduciremos datos en nuestra tabla una fila cada vez, suministrando la información a almacenar en cada columna de la fila. Observa que los valores insertados en las columnas se listan en el mismo orden en que se declararon las columnas cuando se creó la tabla, que es el orden por defecto.

El siguiente código inserta una fila de datos con `Colombian` en la columna `COF_NAME`, `101` en `SUP_ID`, `7.99` en `PRICE`, `0` en `SALES`, y `0` en `TOTAL`. (Como acabamos de inaugurar "The Coffee Break", la cantidad vendida durante la semana y la cantidad total son cero para todos los cafés). Al igual que hicimos con el código que creaba la tabla `COFFEES`, crearemos un objeto `Statement` y lo ejecutaremos utilizando el método `executeUpdate`.

Como la sentencia SQL es demasiado larga como para entrar en una sola línea, la hemos dividido en dos strings concatenándolas mediante un signo más (+) para que puedan compilarse. Presta especial atención a la necesidad de un espacio entre `COFFEES` y `VALUES`. Este espacio debe estar dentro de las comillas y debe estar después de `COFFEES` y antes de `VALUES`; sin un espacio, la sentencia SQL sería leída erróneamente como `"INSERT INTO COFFEESVALUES . . ."` y el controlador de la base de datos buscaría la tabla `COFFEESVALUES`. Observa también que utilizamos comilla simple alrededor del nombre del café porque está anidado dentro de las comillas dobles. Para la mayoría de controladores de bases de datos, la regla

general es alternar comillas dobles y simples para indicar anidación.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

El siguiente código inserta una segunda línea dentro de la tabla COFFEES. Observa que hemos reutilizado el objeto Statement: stmt en vez de tener que crear uno nuevo para cada ejecución.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Los valores de las siguientes filas se pueden insertar de esta forma:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Obtener Datos desde una Tabla

Ahora que la tabla COFFEES tiene valores, podemos escribir una sentencia SELECT para acceder a dichos valores. El asterisco (*) en la siguiente sentencia SQL indica que la columna debería ser seleccionada. Como no hay cláusula WHERE que limite las columnas a seleccionar, la siguiente sentencia SQL selecciona la tabla completa:

```
SELECT * FROM COFFEES
```

El resultado, que es la tabla completa, se parecería a esto:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

El resultado anterior es lo que veríamos en nuestro terminal si introdujeramos la petición SQL directamente en el sistema de la base de datos. Cuando accedemos a una base de datos a través de una aplicación Java, como veremos pronto, necesitamos recuperar los resultados para poder utilizarlos. Veremos como hacer esto en la siguiente página.

Aquí tenemos otro ejemplo de una sentencia SELECT, ésta obtiene una lista de cáfes y sus respectivos precios por libra:


```
SELECT COF_NAME, PRICE FROM COFFEES
```

El resultado de esta consulta se parecería a esto:

COF_NAME	-----	PRICE
Colombian		7.99
French_Roast		8.99
Espresso		9.99
Colombian_Decaf		8.99
French_Roast_Decaf		9.99

La sentencia SELECT genera los nombres y precios de todos los cáfes de la tabla. La siguiente sentencia SQL limita los cafés seleccionados a aquellos que cuesten menos de \$9.00 por libra:

```
SELECT COF_NAME, PRICE  
FROM COFFEES  
WHERE PRICE < 9.00
```

El resultado se parecería es esto:

COF_NAME	-----	PRICE
Colombian		7.99
French_Roast		8.99
Colombian Decaf		8.99

Recuperar Valores de una Hoja de Resultados

Ahora veremos como enviar la sentencia SELECT de la página anterior desde un programa escrito en Java y como obtener los resultados que hemos mostrado.

JDBC devuelve los resultados en un objeto ResultSet, por eso necesitamos declarar un ejemplar de la clase ResultSet para contener los resultados. El siguiente código presenta el objeto ResultSet: rs y le asigna el resultado de una consulta anterior:

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

Utilizar el Método next

La variable rs, que es un ejemplar de ResultSet, contiene las filas de cafés y sus precios mostrados en el juego de resultados de la página anterior. Para acceder a los nombres y los precios, iremos a la fila y recuperaremos los valores de acuerdo con sus tipos. El método next mueve algo llamado cursor a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto ResultSet, primero debemos llamar al método next para mover el cursor a la primera fila y convertirla en la fila actual. Sucesivas invocaciones del método next moverán el cursor de línea de arriba a abajo. Observa que con el JDBC 2.0, cubierto en la siguiente sección, se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia adelante.

Utilizar los métodos getXXX

Los métodos getXXX del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de rs es COF_NAME, que almacena un valor del tipo VARCHAR de SQL. El método para recuperar un valor VARCHAR es getString. La segunda columna de cada fila almacena un valor del tipo FLOAT de SQL, y el método para recuperar valores de ese tipo es getFloat. El siguiente código accede a los valores almacenados en la fila actual de rs e imprime una línea con el nombre seguido por tres espacios y el precio. Cada vez que se llama al método next, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en rs:

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String s = rs.getString("COF_NAME");
        Float n = rs.getFloat("PRICE");
        System.out.println(s + "    " + n);
    }
```

La salida se parecerá a esto:

```
Colombian      7.99
French_Roast   8.99
Espresso       9.99
Colombian_Decaf  8.99
French_Roast_Decaf  9.99
```

Veamos cómo funcionan los métodos getXXX examinando las dos sentencias getXXX de este código. Primero examinaremos getString.

```
String s = rs.getString("COF_NAME");
```

El método getString es invocado sobre el objeto ResultSet: rs, por eso getString recuperará (obtendrá) el valor almacenado en la columna COF_NAME de la fila actual de rs. El valor recuperado por getString se ha convertido desde un VARCHAR de SQL a un String de Java y se ha asignado al objeto String s. Observa que utilizamos la variable s en la expresión println mostrada arriba, de esta forma: println(s + " " + n)

La situación es similar con el método getFloat excepto en que recupera el valor almacenado en la columna PRICE, que es un FLOAT de SQL, y lo convierte a un float de Java antes de asignarlo a la variable n.

JDBC ofrece dos formas para identificar la columna de la que un método getXXX obtiene un valor. Una forma es dar el nombre de la columna, como se ha hecho arriba. La segunda forma es dar el índice de la columna (el número de columna), con un 1 significando la primera columna, un 2 para la segunda, etc. Si utilizáramos el número de columna en vez del nombre de columna el código anterior se podría parecer a esto:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

La primera línea de código obtiene el valor de la primera columna de la fila actual de rs (columna COF_NAME), convirtiéndolo a un objeto String de Java y asignándolo a s. La segunda línea de código obtiene el valor de la segunda columna de la fila actual de rs, lo convierte a un float de Java y lo asigna a n. Recuerda que el número de columna se refiere al número de columna en la hoja de resultados no en la tabla original.

En suma, JDBC permite utilizar tanto el nombre cómo el número de la columna como argumento a un método getXXX. Utilizar el número de columna es un poco más eficiente, y hay algunos casos donde es necesario utilizarlo.

JDBC permite muchas lateralidades para utilizar los métodos getXXX para obtener diferentes tipos de datos SQL. Por ejemplo, el método getInt puede ser utilizado para recuperar cualquier tipo numérico de caracteres. Los datos recuperados serán convertidos a un int; esto es, si el tipo SQL es VARCHAR, JDBC intentará convertirlo en un entero. Se recomienda utilizar el método getInt sólo para recuperar INTEGER de SQL, sin embargo, no puede utilizarse con los tipos BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, o TIMESTAMP de SQL.

[Métodos para Recuperar Tipos SQL](#) muestra qué métodos pueden utilizarse legalmente para recuperar tipos SQL, y más importante, qué métodos están recomendados para recuperar los distintos tipos SQL. Observa que esta tabla utiliza el término "JDBC type" en lugar de "SQL type." Ambos términos se refieren a los tipos genéricos de SQL definidos en java.sql.Types, y ambos son intercambiables.

Utilizar el método getString

Aunque el metodo getString está recomendado para recuperar tipos CHAR y VARCHAR de SQL, es posible recuperar cualquier tipo básico SQL con él. (Sin embargo, no se pueden recuperar los nuevos tipos de datos del SQL3. Explicaremos el SQL3 más adelante).

Obtener un valor con getString puede ser muy útil, pero tiene sus limitaciones. Por ejemplo, si se está utilizando para recuperar un tipo numérico, getString lo convertirá en un String de Java, y el valor tendrá que ser convertido de nuevo a número antes de poder operar con él.

Utilizar los métodos de ResultSet.getString para Recuperar tipos JDBC

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Una "x" indica que el método getXXX se puede utilizar legalmente para recuperar el tipo JDBC dado.

Una "X" indica que el método getXXX está recomendado para recuperar el tipo JDBC dado.

Actualizar Tablas

Supongamos que después de una primera semana exitosa, el propietario de "The Coffee Break" quiere actualizar la columna SALES de la tabla COFFEES introduciendo el número de libras vendidas de cada tipo de café. La sentencia SQL para actualizar una columna se podría parecer a esto:

```
String updateString = "UPDATE COFFEES " +  
    "SET SALES = 75 " +  
    "WHERE COF_NAME LIKE 'Colombian'";
```

Utilizando el objeto stmt, este código JDBC ejecuta la sentencia SQL contenida en updateString:

```
stmt.executeUpdate(updateString);
```

La tabla COFFEES ahora se parecerá a esto:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Observa que todavía no hemos actualizado la columna TOTAL, y por eso tiene valor 0.

Ahora seleccionaremos la fila que hemos actualizado, recuperando los valores de las columnas COF_NAME y SALES, e imprimiendo esos valores:

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +  
    "WHERE COF_NAME LIKE 'Colombian'";  
ResultSet rs = stmt.executeQuery(query);  
while (rs.next()) {  
    String s = rs.getString("COF_NAME");  
    int n = rs.getInt("SALES");  
    System.out.println(n + " pounds of " + s +  
        " sold this week.")  
}
```

Esto imprimirá lo siguiente:

```
75 pounds of Colombian sold this week.
```

Cómo la cláusula WHERE limita la selección a una sola línea, sólo hay una línea en la ResultSet: rs y una línea en la salida. Por lo tanto, sería posible escribir el código sin un bucle while:

```
rs.next();  
String s = rs.getString(1);
```

```
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + " sold this week.")
```

Aunque hay una sólo línea en la hoja de resultados, necesitamos utilizar el método `next` para acceder a ella. Un objeto `ResultSet` se crea con un cursor apuntando por encima de la primera fila. La primera llamada al método `next` posiciona el cursor en la primera fila (y en este caso, la única) de `rs`. En este código, sólo se llama una vez a `next`, si sucediera que existiera una línea, nunca se accedería a ella.

Ahora actualizaremos la columna `TOTAL` añadiendo la cantidad vendida durante la semana a la cantidad total existente, y luego imprimiremos el número de libras vendidas hasta la fecha:

```
String updateString = "UPDATE COFFEES " +
                      "SET TOTAL = TOTAL + 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +
               "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString(1);
    int n = rs.getInt(2);
    System.out.println(n + " pounds of " + s + " sold to date.")
}
}
```

Observa que en este ejemplo, utilizamos el índice de columna en vez del nombre de columna, suministrando el índice 1 a `getString` (la primera columna de la hoja de resultados es `COF_NAME`), y el índice 2 a `getInt` (la segunda columna de la hoja de resultados es `TOTAL`). Es importante distinguir entre un índice de columna en la tabla de la base de datos como opuesto al índice en la tabla de la hoja de resultados. Por ejemplo, `TOTAL` es la quinta columna en la tabla `COFFEES` pero es la segunda columna en la hoja de resultados generada por la petición del ejemplo anterior.

Utilizar Sentencias Prepared

Algunas veces es más conveniente o eficiente utilizar objetos PreparedStatement para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, Statement, que ya conocemos.

Cuándo utilizar un Objeto PreparedStatement

Si queremos ejecutar muchas veces un objeto Statement, reduciremos el tiempo de ejecución si utilizamos un objeto PreparedStatement, en su lugar.

La característica principal de un objeto PreparedStatement es que, al contrario que un objeto Statement, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilado. Como resultado, el objeto PreparedStatement no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto significa que cuando se ejecuta la PreparedStatement, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.

Aunque los objetos PreparedStatement se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventaja de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos. Veremos un ejemplo de esto en las páginas siguientes.

Crear un Objeto PreparedStatement

Al igual que los objetos Statement, creamos un objeto PreparedStatement con un objeto Connection. Utilizando nuestra conexión con abierta en ejemplos anteriores, podríamos escribir lo siguiente para crear un objeto PreparedStatement que tome dos parámetros de entrada:

```
PreparedStatement updateSales = con.prepareStatement(  
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

La variable updateSales contiene la sentencia SQL, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?", que también ha sido, en la mayoría de los casos, enviada al controlador de la base de datos, y ha sido precompilado.

Suministrar Valores para los Parámetros de un PreparedStatement

Necesitamos suministrar los valores que se utilizarán en los lugares donde están las marcas de interrogación, si hay alguno, antes de ejecutar un objeto PreparedStatement. Podemos hacer esto llamado a uno de los métodos setXXX definidos en la clase PreparedStatement. Si el valor que queremos sustituir por una marca de interrogación es un int de Java, podemos llamar al método setInt. Si el valor que queremos sustituir es un String de Java, podemos llamar al método setString, etc. En general, hay un método setXXX para cada tipo Java.

Utilizando el objeto updateSales del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un int de Java, con un valor de 75:

```
updateSales.setInt(1, 75);
```

Cómo podríamos asumir a partir de este ejemplo, el primer argumento de un método setXXX indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el string "Colombian":

```
updateSales.setString(2, "Colombian");
```

Después de que estos valores hayan sido asignados para sus dos parámetros, la sentencia SQL de updateSales será equivalente a la sentencia SQL que hay en string updateString que utilizando en el ejemplo anterior. Por lo tanto, los dos fragmentos de código siguientes consiguen la misma cosa:

Código 1:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +  
                      "WHERE COF_NAME LIKE 'Colombian'";  
stmt.executeUpdate(updateString);
```

Código 2:

```
PreparedStatement updateSales = con.prepareStatement(  
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");  
updateSales.setInt(1, 75);  
updateSales.setString(2, "Colombian");  
updateSales.executeUpdate();
```

Utilizamos el método executeUpdate para ejecutar ambas sentencias stmt updateSales. Observa, sin embargo, que no se suministran argumentos a executeUpdate cuando se utiliza para ejecutar updateSales. Esto es cierto porque updateSales ya contiene la sentencia SQL a ejecutar.

Mirando estos ejemplos podríamos preguntarnos por qué utilizar un objeto PreparedStatement con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si actualizáramos la columna SALES sólo una o dos veces, no sería necesario utilizar una sentencia SQL con parámetros. Si por otro lado, tuviéramos que actualizarla frecuentemente, podría ser más fácil utilizar un objeto PreparedStatement, especialmente en situaciones cuando la utilizamos con un bucle while para seleccionar un parámetro a una sucesión de valores. Veremos este ejemplo más adelante en esta sección.

Una vez que a un parámetro se ha asignado un valor, el valor permanece hasta que lo resetee otro valor o se llame al método clearParameters. Utilizando el objeto PreparedStatement: updateSales, el siguiente fragmento de código reutiliza una sentencia prepared después de resetar el valor de uno de sus parámetros, dejando el otro igual:

```
updateSales.setInt(1, 100);  
updateSales.setString(2, "French_Roast");  
updateSales.executeUpdate();  
// changes SALES column of French Roast row to 100  
updateSales.setString(2, "Espresso");  
updateSales.executeUpdate();  
// changes SALES column of Espresso row to 100 (the first  
// parameter stayed 100, and the second parameter was reset  
// to "Espresso")
```

Utilizar una Bucle para asignar Valores

Normalmente se codifica más sencillo utilizando un bucle for o while para asignar valores de los parámetros de entrada.

El siguiente fragmento de código demuestra la utilización de un bucle for para asignar los parámetros en un objeto PreparedStatement: updateSales. El array salesForWeek contiene las cantidades vendidas semanalmente. Estas cantidades corresponden con los nombres de los cafés listados en el array coffees, por eso la primera cantidad de salesForWeek (175) se aplica al primer nombre de café de coffees ("Colombian"), la segunda cantidad de salesForWeek (150) se aplica al segundo nombre de café en coffees ("French_Roast"), etc. Este fragmento de código

demuestra la actualización de la columna SALES para todos los cafés de la tabla COFFEES

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
    "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);int [] salesForWeek = {175, 150, 60,
155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
    "Colombian_Decaf", "French_Roast_Decaf"};

int len = coffees.length;
for(int i = 0; i < len; i++) {
    updateSales.setInt(1, salesForWeek[i]);
    updateSales.setString(2, coffees[i]);
    updateSales.executeUpdate();
}
```

Cuando el propietario quiera actualizar las ventas de la semana siguiente, puede utilizar el mismo código como una plantilla. Todo lo que tiene que hacer es introducir las nuevas cantidades en el orden apropiado en el array salesForWeek. Los nombres de cafés del array coffees permanecen constantes, por eso no necesitan cambiarse. (En una aplicación real, los valores probablemente serían introducidos por el usuario en vez de desde un array inicializado).

Valores de retorno del método executeUpdate

Siempre que executeQuery devuelve un objeto ResultSet que contiene los resultados de una petición al controlador de la base de datos, el valor devuelto por executeUpdate es un int que indica cuántas líneas de la tabla fueron actualizadas. Por ejemplo, el siguiente código muestra el valor de retorno de executeUpdate asignado a la variable n:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

La tabla COFFEES se ha actualizado poniendo el valor 50 en la columna SALES de la fila correspondiente a Espresso. La actualización afecta sólo a una línea de la tabla, por eso n es igual a 1.

Cuando el método executeUpdate es utilizado para ejecutar una sentencia DDL, como la creación de una tabla, devuelve el int: 0. Consecuentemente, en el siguiente fragmento de código, que ejecuta la sentencia DDL utilizada para crear la tabla COFFEES, n tendrá el valor 0:

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Observa que cuando el valor devuelto por executeUpdate sea 0, puede significar dos cosas: (1) la sentencia ejecutada no ha actualizado ninguna fila, o (2) la sentencia ejecutada fue una sentencia DDL.

Utilizar Joins

Algunas veces necesitamos utilizar una o más tablas para obtener los datos que queremos. Por ejemplo, supongamos que el propietario del "The Coffee Break" quiere una lista de los cafés que le compra a Acme, Inc. Esto implica información de la tabla COFFEES y también de la que vamos a crear SUPPLIERS. Este es el caso en que se necesitan los "joins" (unión). Una unión es una operación de base de datos que relaciona dos o más tablas por medio de los valores que comparten. En nuestro ejemplo, las tablas COFFEES y SUPPLIERS tienen la columna SUP_ID, que puede ser utilizada para unir las.

Antes de ir más allá, necesitamos crear la tabla SUPPLIERS y rellenarla con valores.

El siguiente código crea la tabla SUPPLIERS:

```
String createSUPPLIERS = "create table SUPPLIERS " +
    "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
    "STREET VARCHAR(40), CITY VARCHAR(20), " +
    "STATE CHAR(2), ZIP CHAR(5))";

stmt.executeUpdate(createSUPPLIERS);
```

El siguiente código inserta filas para tres suministradores dentro de SUPPLIERS:

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " +
    "'Acme, Inc.', '99 Market Street', 'Groundsville', " + "'CA',
    '95199')");
stmt.executeUpdate("Insert into SUPPLIERS values (49," +
    "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA', " +
    "'95460')");
stmt.executeUpdate("Insert into SUPPLIERS values (150, " +
    "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " +
    "'93966')");
```

El siguiente código selecciona la tabla y nos permite verla:

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

El resultado sería algo similar a esto:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Ahora que tenemos las tablas COFFEES y SUPPLIERS, podremos proceder con el escenario en que el propietario quería una lista de los cafés comprados a un suministrador particular. Los nombres de los suministradores están en la tabla SUPPLIERS, y los nombres de los cafés en la tabla COFFEES. Como ambas tablas tienen la columna SUP_ID, podemos utilizar esta columna en una unión. Lo siguiente que necesitamos es la forma de distinguir la columna SUP_ID a la que nos referimos. Esto se hace precediendo el nombre de la columna con el nombre de la tabla, "COFFEES.SUP_ID" para indicar que queremos referirnos a la columna SUP_ID de la tabla COFFEES. En el siguiente código, donde stmt es un objeto Statement, seleccionamos los cafés comprados a Acme, Inc.:

```
String query = "
SELECT COFFEES.COF_NAME " +
    "FROM COFFEES, SUPPLIERS " +
    "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.'" +
```



```
"and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";
```

```
ResultSet rs = stmt.executeQuery(query);  
System.out.println("Coffees bought from Acme, Inc.: ");  
while (rs.next()) {  
    String coffeeName = getString("COF_NAME");  
    System.out.println("    " + coffeeName);  
}
```

Esto producirá la siguiente salida:

```
Coffees bought from Acme, Inc.:  
    Colombian  
    Colombian_Decaf
```

Ozito

Utilizar Transacciones

Hay veces que no queremos que una sentencia tenga efecto a menos que otra también suceda. Por ejemplo, cuando el propietario del "The Coffee Break" actualiza la cantidad de café vendida semanalmente, también querrá actualizar la cantidad total vendida hasta la fecha. Sin embargo, el no querrá actualizar una sin actualizar la otra; de otro modo, los datos serían inconsistentes. La forma para asegurarnos que ocurren las dos acciones o que no ocurre ninguna es utilizar una transacción. Una transacción es un conjunto de una o más sentencias que se ejecutan como una unidad, por eso o se ejecutan todas o no se ejecuta ninguna.

Desactivar el modo Auto-entrega

Cuando se crea una conexión, está en modo auto-entrega. Esto significa que cada sentencia SQL individual es tratada como una transacción y será automáticamente entregada justo después de ser ejecutada. (Para ser más preciso, por defecto, una sentencia SQL será entregada cuando está completa, no cuando se ejecuta. Una sentencia está completa cuando todas sus hojas de resultados y cuentas de actualización han sido recuperadas. Sin embargo, en la mayoría de los casos, una sentencia está completa, y por lo tanto, entregada, justo después de ser ejecutada).

La forma de permitir que dos o más sentencias sean agrupadas en una transacción es desactivar el modo auto-entrega. Esto se demuestra en el siguiente código, donde con es una conexión activa:

```
con.setAutoCommit(false);
```

Entregar una Transacción

Una vez que se ha desactivado la auto-entrega, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método commit. Todas las sentencias ejecutadas después de la anterior llamada al método commit serán incluidas en la transacción actual y serán entregadas juntas como una unidad. El siguiente código, en el que con es una conexión activa, ilustra una transacción:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

En este ejemplo, el modo auto-entrega se desactiva para la conexión con, lo que significa que las dos sentencias prepared updateSales y updateTotal serán entregadas juntas

cuando se llame al método commit. Siempre que se llame al método commit (bien automáticamente, cuando está activado el modo auto-commit o explícitamente cuando está desactivado), todos los cambios resultantes de las sentencias de la transacción serán permanentes. En este caso, significa que las columnas SALES y TOTAL para el café Colombian han sido cambiadas a 50 (si TOTAL ha sido 0 anteriormente) y mantendrá este valor hasta que se cambie con otra sentencia de actualización.

La línea final del ejemplo anterior activa el modo auto-commit, lo que significa que cada sentencia será de nuevo entregada automáticamente cuando esté completa. Volvemos por lo tanto al estado por defecto, en el que no tenemos que llamar al método commit. Es bueno desactivar el modo auto-commit sólo mientras queramos estar en modo transacción. De esta forma, evitamos bloquear la base de datos durante varias sentencias, lo que incrementa los conflictos con otros usuarios.

Utilizar Transacciones para Preservar la Integridad de los Datos

Además de agrupar las sentencias para ejecutarlas como una unidad, las transacciones pueden ayudarnos a preservar la integridad de los datos de una tabla. Por ejemplo, supongamos que un empleado se ha propuesto introducir los nuevos precios de los cafés en la tabla COFFEES pero lo retrasa unos días. Mientras tanto, los precios han subido, y hoy el propietario está introduciendo los nuevos precios. Finalmente el empleado empieza a introducir los precios ahora desfasados al mismo tiempo que el propietario intenta actualizar la tabla. Después de insertar los precios desfasados, el empleado se da cuenta de que ya no son válidos y llama el método rollback de la Connection para deshacer sus efectos. (El método rollback aborta la transacción y restaura los valores que había antes de intentar la actualización. Al mismo tiempo, el propietario está ejecutando una sentencia SELECT e imprime los nuevos precios. En esta situación, es posible que el propietario imprima los precios que más tarde serían devueltos a sus valores anteriores, haciendo que los precios impresos sean incorrectos.

Esta clase de situaciones puede evitarse utilizando Transacciones. Si un controlador de base de datos soporta transacciones, y casi todos lo hacen, proporcionará algún nivel de protección contra conflictos que pueden surgir cuando dos usuarios acceden a los datos a la misma vez.

Para evitar conflictos durante una transacción, un controlador de base de datos utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que están siendo accedidos por una transacción. (Observa que en el modo auto-commit, donde cada sentencia es una transacción, el bloqueo sólo se mantiene durante una sentencia). Una vez activado, el bloqueo permanece hasta que la transacción sea entregada o anulada. Por ejemplo, un controlador de base de datos podría bloquear una fila de una tabla hasta que la actualización se haya entregado. El efecto de este bloqueo es evitar que un usuario obtenga una lectura sucia, esto es, que lea un valor antes de que sea permanente. (Acceder a un valor actualizado que no haya sido entregado se considera una lectura sucia porque es posible que el valor sea devuelto a su valor anterior. Si leemos un valor que luego es devuelto a su valor antiguo, habremos leído un valor nulo).

La forma en que se configuran los bloqueos está determinado por lo que se llama nivel de aislamiento de transacción, que puede variar desde no soportar transacciones en absoluto a soportar todas las transacciones que fuerzan una reglas de acceso muy estrictas.

Un ejemplo de nivel de aislamiento de transacción es TRANSACTION_READ_COMMITTED, que no permite que se acceda a un valor hasta que haya sido entregado. En otras palabras,

si nivel de aislamiento de transacción se selecciona a TRANSACTION_READ_COMMITTED, el controlador de la base de datos no permitirá que ocurran lecturas sucias. El interface Connection incluye cinco valores que representan los niveles de aislamiento de transacción que se pueden utilizar en JDBC.

Normalmente, no se necesita cambiar el nivel de aislamiento de transacción; podemos utilizar el valor por defecto de nuestro controlador. JDBC permite averiguar el nivel de aislamiento de transacción de nuestro controlador de la base de datos (utilizando el método `getTransactionIsolation` de `Connection`) y permite configurarlo a otro nivel (utilizando el método `setTransactionIsolation` de `Connection`). Sin embargo, ten en cuenta, que aunque JDBC permite seleccionar un nivel de aislamiento, hacer esto no tendrá ningún efecto a no ser que el driver del controlador de la base de datos lo soporte.

Cuándo llamar al método `rollback`

Como se mencionó anteriormente, llamar al método `rollback` aborta la transacción y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transacción y obtenemos una `SQLException`, deberíamos llamar al método `rollback` para abortar la transacción y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y cuál no ha sido entregada. Capturar una `SQLException` nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada. Como no podemos contar con el hecho de que nada fue entregado, llamar al método `rollback` es la única forma de asegurarnos.

Procedimientos Almacenados

Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Por ejemplo, las operaciones sobre una base de datos de empleados (salarios, despidos, promociones, bloqueos) podrían ser codificados como procedimientos almacenados ejecutados por el código de la aplicación. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros y resultados, y podrían tener cualquier combinación de parámetros de entrada/salida.

Los procedimientos almacenados están soportados por la mayoría de los controladores de bases de datos, pero existe una gran cantidad de variaciones en su sintaxis y capacidades. Por esta razón, sólo mostraremos un ejemplo sencillo de lo que podría ser un procedimiento almacenado y cómo llamarlos desde JDBC, pero este ejemplo no está diseñado para ejecutarse.

Sentencias SQL para crear un Procedimiento Almacenado

Esta página muestra un procedimiento almacenado muy sencillo que no tiene parámetros. Aunque la mayoría de los procedimientos almacenados hacen cosas más complejas que este ejemplo, sirve para ilustrar algunos puntos básicos sobre ellos. Como paso previo, la sintaxis para definir un procedimiento almacenado es diferente de un controlador de base de datos a otro. Por ejemplo, algunos utilizan `begin . . . end` u otras palabras clave para indicar el principio y final de la definición de procedimiento. En algunos controladores, la siguiente sentencia SQL crea un procedimiento almacenado:

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

El siguiente código pone la sentencia SQL dentro de un string y lo asigna a la variable `createProcedure`, que utilizaremos más adelante:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
                        "as " +
                        "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
                        "from SUPPLIERS, COFFEES " +
                        "where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
                        "order by SUP_NAME";
```

El siguiente fragmento de código utiliza el objeto `Connection`, con para crear un objeto `Statement`, que es utilizado para enviar la sentencia SQL que crea el procedimiento almacenado en la base de datos:

```
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

El procedimiento `SHOW_SUPPLIERS` será compilado y almacenado en la base de datos como un objeto de la propia base y puede ser llamado, como se llamaría a cualquier otro método.

Llamar a un Procedimiento Almacenado desde JDBC

JDBC permite llamar a un procedimiento almacenado en la base de datos desde una aplicación escrita en Java. El primer paso es crear un objeto `CallableStatement`. Al igual que con los objetos `Statement` y `PreparedStatement`, esto se hace con una conexión abierta, `Connection`. Un objeto `CallableStatement` contiene una llamada a un procedimiento almacenado; no contiene el propio procedimiento. La primera línea del código siguiente crea una llamada al procedimiento almacenado `SHOW_SUPPLIERS` utilizando la conexión `con`. La parte que está encerrada entre corchetes es la sintaxis de escape para los procedimientos almacenados. Cuando un controlador encuentra `"{call SHOW_SUPPLIERS}"`, traducirá esta sintaxis de escape al SQL nativo utilizado en la base de datos para llamar al procedimiento almacenado llamado `SHOW_SUPPLIERS`:

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

La hoja de resultados de `rs` será similar a esto:

SUP_NAME	COF_NAME
-----	-----
Acme, Inc.	Colombian
Acme, Inc.	Colombian_Decaf
Superior Coffee	French_Roast
Superior Coffee	French_Roast_Decaf
The High Ground	Espresso

Observa que el método utilizado para ejecutar cs es `executeQuery` porque cs llama a un procedimiento almacenado que contiene una petición y esto produce una hoja de resultados. Si el procedimiento hubiera contenido una sentencia de actualización o una sentencia DDL, se hubiera utilizado el método `executeUpdate`. Sin embargo, en algunos casos, cuando el procedimiento almacenado contiene más de una sentencia SQL producirá más de una hoja de resultados, o cuando contiene más de una cuenta de actualización o alguna combinación de hojas de resultados y actualizaciones. en estos casos, donde existen múltiples resultados, se debería utilizar el método `execute` para ejecutar `CallableStatement`.

La clase `CallableStatement` es una subclase de `PreparedStatement`, por eso un objeto `CallableStatement` puede tomar parámetros de entrada como lo haría un objeto `PreparedStatement`. Además, un objeto `CallableStatement` puede tomar parámetros de salida, o parámetros que son tanto de entrada como de salida. Los parámetros INOUT y el método `execute` se utilizan raramente.

Crear Aplicaciones JDBC Completas

Hasta ahora sólo hemos visto fragmentos de código. Más adelante veremos programas de ejemplo que son aplicaciones completas que podremos ejecutar.

El primer código de ejemplo crea la tabla COFFEES; el segundo inserta valores en la tabla e imprime los resultados de una petición. La tercera aplicación crea la tabla SUPPLIERS, y el cuarto la rellena con valores. Después de haber ejecutado este código, podemos intentar una petición que una las tablas COFFEES y SUPPLIERS, como en el quinto código de ejemplo. El sexto ejemplo de código es una aplicación que demuestra una transacción y también muestra como configurar las posiciones de los parámetros en un objeto PreparedStatement utilizando un bucle for.

Como son aplicaciones completas, incluyen algunos elementos del lenguaje Java que no hemos visto en los fragmentos anteriores. Aquí explicaremos estos elementos brevemente.

Poner Código en una Definición de Clase

En el lenguaje Java, cualquier código que querramos ejecutar debe estar dentro de una definición de clase. Tecleamos la definición de clase en un fichero y a éste le damos el nombre de la clase con la extensión .java. Por eso si tenemos una clase llamada MySQLStatement, su definición debería estar en un fichero llamado MySQLStatement.java

Importar Clases para Hacerlas Visibles

Lo primero es importar los paquetes o clases que se van a utilizar en la nueva clase. Todas las clases de nuestros ejemplos utilizan el paquete java.sql (el API JDBC), que se hace visible cuando la siguiente línea de código precede a la definición de clase:

```
import java.sql.*;
```

El asterisco (*) indica que todas las clases del paquete java.sql serán importadas. Importar una clase la hace visible y significa que no tendremos que escribir su nombre totalmente cualificado cuando utilicemos un método o un campo de esa clase. Si no incluimos "import java.sql.*;" en nuestro código, tendríamos que escribir "java.sql." más el nombre de la clase delante de todos los campos o métodos JDBC que utilicemos cada vez que los utilicemos. Observa que también podemos importar clases individuales selectivamente en vez de importar un paquete completo. Java no requiere que importemos clases o paquetes, pero al hacerlo el código se hace mucho más conveniente.

Cualquier línea que importe clases aparece en la parte superior de los ejemplos de código, que es donde deben estar para hacer visibles las clases importadas a la clase que está siendo definida. La definición real de la clase sigue a cualquier línea que importe clases.

Utilizar el Método main()

Si una clase se va a ejecutar, debe contener un método static public main. Este método viene justo después de la línea que declara la clase y llama a los otros métodos de la clase. La palabra clave static indica que este método opera a nivel de clase en vez sobre ejemplares individuales de la clase. La palabra clave public significa que los miembros de cualquier clase pueden acceder a este método. Como no estamos definiendo clases sólo para ser ejecutadas por otras clases sino que queremos ejecutarlas, las aplicaciones de ejemplo de este capítulo incluyen un método main.

Utilizar bloques try y catch

Algo que también incluyen todas las aplicaciones de ejemplo son los bloques try y catch. Este es un mecanismo del lenguaje Java para manejar excepciones. Java requiere que cuando un método lanza una excepción exista un mecanismo que la maneje. Generalmente un bloque catch capturará la excepción y especificará lo que sucederá (que podría ser no hacer nada). En el código de ejemplo, utilizamos dos bloques try y dos bloques catch. El primer bloque try

contiene el método `Class.forName`, del paquete `java.lang`. Este método lanza una `ClassNotFoundException`, por eso el bloque `catch` que le sigue maneja esa excepción. El segundo bloque `try` contiene métodos JDBC, todos ellos lanzan `SQLException`, por eso el bloque `catch` del final de la aplicación puede manejar el resto de las excepciones que podrían lanzarse ya que todas serían objetos `SQLException`.

Recuperar Excepciones

JDBC permite ver los avisos y excepciones generados por nuestro controlador de base de datos y por el compilador Java. Para ver las excepciones, podemos tener un bloque `catch` que las imprima. Por ejemplo, los dos bloques `catch` del siguiente código de ejemplo imprimen un mensaje explicando la excepción:

```
try {
    // Aquí va el código que podría generar la excepción.
    // Si se genera una excepción, el bloque catch imprimirá
    // información sobre ella.
} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}

try {
    Class.forName("myDriverClassName");
} catch(java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
}
```

Si ejecutáramos `CreateCOFFEES.java` dos veces, obtendríamos un mensaje de error similar a éste:

```
SQLException: There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
```

Este ejemplo ilustra la impresión del componente mensaje de un objeto `SQLException`, lo que es suficiente para la mayoría de las situaciones.

Sin embargo, realmente existen tres componentes, y para ser completos, podemos imprimirlos todos. El siguiente fragmento de código muestra un bloque `catch` que se ha completado de dos formas. Primero, imprime las tres partes de un objeto `SQLException`: el mensaje (un string que describe el error), el `SQLState` (un string que identifica el error de acuerdo a los convenciones X/Open de `SQLState`), y un código de error del vendedor (un número que es el código de error del vendedor del driver). El objeto `SQLException`, `ex` es capturado y se accede a sus tres componentes con los métodos `getMessage`, `getSQLState`, y `getErrorCode`.

La segunda forma del siguiente bloque `catch` completo obtiene todas las excepciones que podrían haber sido lanzadas. Si hay una segunda excepción, sería encadenada a `ex`, por eso se llama a `ex.getNextException` para ver si hay más excepciones. Si las hay, el bucle `while` continúa e imprime el mensaje de la siguiente excepción, el `SQLState`, y el código de error del vendedor. Esto continúa hasta que no haya más excepciones.

```
try {
    // Aquí va el código que podría generar la excepción.
    // Si se genera una excepción, el bloque catch imprimirá
    // información sobre ella.
} catch(SQLException ex) {
    System.out.println("\n--- SQLException caught ---\n");
    while (ex != null) {
        System.out.println("Message:    " + ex.getMessage ());
```

```

        System.out.println("SQLState:  " + ex.getSQLState ());
        System.out.println("ErrorCode: " + ex.getErrorCode ());
        ex = ex.getNextException();
        System.out.println("");
    }
}

```

Si hubieramos sustituido el bloque catch anterior en el Código de ejemplo 1 (CreateCoffees) y lo hubieramos ejecutado después de que la tabla COFFEES ya se hubiera creado, obtendríamos la siguiente información:

```

--- SQLException caught ---
Message:  There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
SQLState: 42501
ErrorCode: 2714

```

SQLState es un código definido en X/Open y ANSI-92 que identifica la excepción. Aquí podemos ver dos ejemplos de códigos SQLState:

```

08001 -- No suitable driver
HY011 -- Operation invalid at this time

```

El código de error del vendedor es específico de cada driver, por lo que debemos revisar la documentación del driver buscando una lista con el significado de estos códigos de error.

Recuperar Avisos

Los objetos SQLWarning son una subclase de SQLException que trata los avisos de accesos a bases de datos. Los Avisos no detienen la ejecución de una aplicación, como las excepciones; simplemente alertan al usuario de que algo no ha salido como se esperaba. Por ejemplo, un aviso podría hacernos saber que un privilegio que queríamos revocar no ha fue revocado. O un aviso podría decirnos que ha ocurrido algún error durante una petición de desconexión.

Un aviso puede reportarse sobre un objeto Connection, un objeto Statement (incluyendo objetos PreparedStatement y CallableStatement), o un objeto ResultSet. Cada una de esas clases tiene un método getWarnings, al que debemos llamar para ver el primer aviso reportado en la llamada al objeto. Si getWarnings devuelve un aviso, podemos llamar al método getNextWarning de SQLWarning para obtener avisos adicionales. Al ejecutar una sentencia se borran automáticamente los avisos de la sentencia anterior, por eso no se apilan. Sin embargo, esto significa que si queremos recuperar los avisos reportados por una sentencia, debemos hacerlo antes de ejecutar otra sentencia.

El siguiente fragmento de código ilustra como obtener información completa sobre los avisos reportados por el objeto Statement, stmt y también por el objeto ResultSet, rs:

```

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select COF_NAME from COFFEES");
while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    System.out.println("Coffees available at the Coffee Break:  ");
    System.out.println("      " + coffeeName);
    SQLWarning warning = stmt.getWarnings();
    if (warning != null) {
        System.out.println("\n---Warning---\n");
        while (warning != null) {
            System.out.println("Message: " + warning.getMessage());
            System.out.println("SQLState: " + warning.getSQLState());
            System.out.print("Vendor error code: ");
            System.out.println(warning.getErrorCode());
            System.out.println("");
        }
    }
}

```

```

        warning = warning.getNextWarning();
    }
}
SQLWarning warn = rs.getWarnings();
if (warn != null) {
    System.out.println("\n---Warning---\n");
    while (warn != null) {
        System.out.println("Message: " + warn.getMessage());
        System.out.println("SQLState: " + warn.getSQLState());
        System.out.print("Vendor error code: ");
        System.out.println(warn.getErrorCode());
        System.out.println("");
        warn = warn.getNextWarning();
    }
}
}

```

Los avisos no son muy comunes, De aquellos que son reportados, el aviso más común es un DataTruncation, una subclase de SQLWarning. Todos los objetos DataTruncation tienen un SQLState 01004, indicando que ha habido un problema al leer o escribir datos. Los métodos de DataTruncation permiten encontrar en que columna o parámetro se truncaron los datos, si la ruptura se produjo en una operación de lectura o de escritura, cuántos bytes deberían haber sido transmitidos, y cuántos bytes se transmitieron realmente.

Ejecutar las Aplicaciones de Ejemplo

Ahora estamos listos para probar algún código de ejemplo. El directorio `book.html`, contiene una aplicación completa, ejecutable, que ilustra los conceptos presentados en este capítulo y el siguiente. Puedes descargar este código de ejemplo del site de JDBC situado en :

<http://www.javasoft.com/products/jdbc/book.html>

Antes de poder ejecutar una de esas aplicaciones, necesitamos editar el fichero sustituyendo la información apropiada para las siguientes variables:

`url`

La URL JDBC, las partes uno y dos son suministradas por el driver, y la tercera parte especifica la fuente de datos.

`myLogin`

Tu nombre de usuario o login.

`myPassword`

Tu password para el controlador de base de datos.

`myDriver.ClassName`

El nombre de clase suministrado con tu driver

La primera aplicación de ejemplo es la clase `CreateCoffees`, que está en el fichero llamado [CreateCoffees.java](#). Abajo tienes las instrucciones para ejecutar `CreateCoffees.java` en las dos plataformas principales.

La primera línea compila el código del fichero `CreateCoffees.java`. Si la compilación tiene éxito, se producirá un fichero llamado `CreateCoffees.class`, que contendrá los bytecodes traducidos desde el fichero `CreateCoffees.java`. Estos bytecodes serán interpretados por la máquina virtual Java, que es la que hace posible que el código Java se pueda ejecutar en cualquier máquina que la tenga instalada.

La segunda línea de código ejecuta el código. Observa que se utiliza el nombre de la clase, `CreateCoffees`, no el nombre del fichero `CreateCoffees.class`.

UNIX

```
javac CreateCoffees.java
java CreateCoffees
```

Windows 95/NT

```
javac CreateCoffees.java
java CreateCoffees
```

Crear un Applet desde una Aplicación

Supongamos que el propietario de "The Coffee Break" quiere mostrar los precios actuales de los cafés en un applet en su página Web. Puede asegurarse de que está mostrando los precios actuales haciendo que applet obtenga los precios directamente desde su base de datos.

Para hacer esto necesita crear dos ficheros de código, uno con el código del applet y otro con el código HTML. El código del applet contiene el código JDBC que podría aparecer en una aplicación normal más el código adicional para ejecutar el applet y mostrar el resultado de la petición a la base de datos. En nuestro ejemplo el código del applet está en el fichero `OutputApplet.java`. Para mostrar el applet en una página HTML, el fichero `OutputApplet.html` le dice al navegador qué mostrar y dónde mostrarlo.

El resto de esta página explicará algunos elementos que se encuentran en el código del applet y que no están presentes en el código de las aplicaciones. Algunos de estos ejemplos involucran aspectos avanzados del lenguaje Java. Daremos alguna explicación básica y racional, ya que una explicación completa va más allá de este tutorial. El propósito de este applet es dar una idea general, para poder utilizarlo como plantilla, sustituyendo nuestras consultas por las del applet.

Escribir el Código del Applet

Para empezar, los applets importan clases que no son utilizadas por las aplicaciones. Nuestro applet importa dos clases que son especiales para los applets: la clase `Applet`, que forma parte del paquete `java.applet`, y la clase `Graphics`, que forma parte del paquete `java.awt`. Este applet también importa la clase de propósito general `java.util.Vector` que se utiliza para acceder a un contenedor tipo array cuyo tamaño puede ser modificado. Este código utiliza objetos `Vector` para almacenar los resultados de las peticiones para poder mostrarlas después.

Todos los applets descienden de la clase `Applet`; es decir, son subclases de `Applet`. Por lo tanto, toda definición de applet debe contener las palabras `extends Applet`; como se vé aquí:

```
public class MyAppletName extends Applet {  
    . . .  
}
```

En nuestro ejemplo, esta línea también incluye las palabras `implements Runnable`, por lo que se parece a esto:

```
public class OutputApplet extends Applet implements Runnable {  
    . . .
```

}

Runnable es un interface que hace posible ejecutar más de un thread a la vez. Un thread es un flujo secuencial de control, y un programa puede tener muchos threads haciendo cosas diferentes concurrentemente. La clase OutputApplet implementa el interface Runnable definiendo el método run; el único método de Runnable. En nuestro ejemplo el método run contiene el código JDBC para abrir la conexión, ejecutar una petición, y obtener los resultados desde la hoja de resultados. Como las conexiones a las bases de datos pueden ser lentas, y algunas veces pueden tardar varios segundos, es una buena idea estructurar un applet para que pueda manejar el trabajo con bases de datos en un thread separado.

Al igual que las aplicaciones deben tener un método main, un applet debe implementar al menos uno de estos métodos init, start, o paint. Nuestro ejemplo define un método start y un método paint. Cada vez que se llama a start, crea un nuevo thread (worker) para re-evaluar la petición a la base de datos. Cada vez que se llama a paint, se muestra o bien el resultado de la petición o un string que describe el estado actual del applet.

Como se mencionó anteriormente, el método run definido en OutputApplet contiene el código JDBC. Cuando el thread worker llama al método start, se llama automáticamente al método run, y éste ejecuta el código JDBC en el thread worker. El código de run es muy similar al código que hemos visto en otros ejemplos con tres excepciones. Primero, utiliza la clase Vector para almacenar los resultados de la petición. Segundo, no imprime los resultados, sino que los añade al Vector, results para mostrarlos más tarde. Tercero, tampoco muestra ninguna excepción, en su lugar almacena los mensajes de error para mostrarlos más tarde.

Los applets tienen varias formas de dibujar, o mostrar, su contenido. Este applet, es uno muy simple que sólo utiliza texto, utiliza el método drawString (una parte de la clase Graphics) para mostrar texto. El método drawString tiene tres argumentos: (1) el string a mostrar, (2) la coordenada x, indicando la posición horizontal de inicio del string, y (3) la coordenada y, indicando la posición vertical de inicio del string (que está en la parte inferior del texto).

El método paint es el que realmente dibuja las cosas en la pantalla, y en OutputApplet.java, está definido para contener llamadas al método drawString. La cosa principal que muestra drawString es el contenido del Vector, results (los resultados almacenados). Cuando no hay resultados que mostrar, drawString muestra el estado actual contenido en el String, message. Este string será "Initializing" para empezar. Será "Connecting to database" cuando se llame al método start, y el

método `setError` pondrá en él un mensaje de error cuando capture una excepción. Así, si la conexión a la base de datos tarda mucho tiempo, la persona que está viendo el applet verá el mensaje "Connecting to database" porque ese será el contenido de message en ese momento. (El método `paint` es llamado por el AWT cuando quiere que el applet muestre su estado actual en la pantalla).

Al menos dos métodos definidos en la clase `OutputApplet`, `setError` y `setResults` son privados, lo que significa que sólo pueden ser utilizados por `OutputApplet`. Estos métodos llaman el método `repaint`, que borra la pantalla y llama a `paint`. Por eso si `setResults` llama a `repaint`, se mostrarán los resultados de la petición, y si `setError` llama a `repaint`, se mostrará un mensaje de error.

Otro punto es hacer que todos los métodos definidos en `OutputApplet` excepto `run` son `synchronized`. La palabra clave `synchronized` indica que mientras que un método esté accediendo a un objeto, otros métodos `synchronized` están bloqueados para acceder a ese objeto. El método `run` no se declara `synchronized` para que el applet pueda dibujarse en la pantalla mientras se produce la conexión a la base de datos. Si los métodos de acceso a la base de datos fueran `synchronized`, evitarían que el applet se redibujara mientras se están ejecutando, lo que podría resultar en retrasos sin acompañamiento de mensaje de estado.

Para sumarizar, en un applet, es una buena práctica de programación es hacer algunas cosas que no necesitaríamos hacer en una aplicación.

1. Poner nuestro código JDBC en un thread separado.
2. Mostrar mensajes de estado durante los retardos, como cuando la conexión a la base de datos tarda mucho tiempo.
3. Mostrar los mensajes de error en la pantalla en lugar de imprimirlos en `System.out` o `System.err`.

Ejecutar un Applet

Antes de ejecutar nuestro applet, necesitamos compilar el fichero `OutputApplet.java`. Esto crea el fichero `OutputApplet.class`, que es referenciado por el fichero `OutputApplet.html`.

La forma más fácil de ejecutar un applet es utilizar el `appletviewer`, que se incluye en el JDK. Sólo debemos seguir las instrucciones para nuestra plataforma:

UNIX

```
javac OutputApplet.java
appletviewer OutputApplet.html
```

Windows 95/NT


```
javac OutputApplet.java  
appletviewer OutputApplet.html
```

Los applets descargados a través de la red están sujetos a distintas restricciones de seguridad. Aunque esto puede parecer molesto, es absolutamente necesario para la seguridad de la red, y la seguridad es una de las mayores ventajas de utilizar Java. Un applet no puede hacer conexiones en la red excepto con el host del que se descargó a menos que el navegador se lo permita. Si uno puede tratar con applets instalados localmente como applets "trusted" (firmados) también dependen de restricciones de seguridad impuestas por el navegador. Un applet normalmente no puede leer o escribir ficheros en el host en el que se está ejecutando, y no puede cargar librerías ni definir métodos nativos.

Los applets pueden hacer conexiones con el host del que vinieron, por eso pueden trabajar muy bien en intranets.

El driver puente JDBC-ODBC es de alguna forma un caso especial. Puede utilizarse satisfactoriamente para acceder a intranet, pero requiere que ODBC, el puente, la librería nativa, y el JDBC esten instalados en cada cliente. Con esta configuración, los accesos a intranet funcionan con aplicaciones Java y con applets firmados. Sin embargo, como los puentes requieren configuraciones especiales del cliente, no es práctico para ejecutar applets en Internet con el puente JDBC-ODBC. Observa que esta limitaciones son para el puente JDBC-ODBC, no para JDBC. Con un driver JDBC puro Java, no se necesita ninguna configuración especial para ejecutar applets en Internet.

Nuevas Características en el API JDBC 2.0

El paquete `java.sql` que está incluido en la versión JDB 1.2 (conocido como el API JDBC 2.0) incluye muchas nuevas características no incluidas en el paquete `java.sql` que forma parte de la versión JDK 1.1 (referenciado como el API JDBC 1.0).

Con el API JDBC 2.0, podremos hacer las siguientes cosas:

- Ir hacia adelante o hacia atrás en una hoja de resultados o movernos a un fila específica.
- Hacer actualizaciones de las tablas de la base datos utilizando métodos Java en lugar de utilizar comandos SQL.
- Enviar múltiples secuencias SQL a la base de datos como una unidad, o batch.
- Utilizar los nuevos tipos de datos SQL3 como valores de columnas.

[Configuración del API JDBC 2.0](#)

[Mover el Cursor en una Hoja de Resultados](#)

[Hacer Actualizaciones en una Hoja de Resultados Actualizable](#)

[Actualizar una Hoja de Resultados Programáticamente](#)

[Insertar y Borrar Filas Programáticamente](#)

[Código de ejemplo para Insertar una Fila](#)

[Borrar un Fila](#)

[Hacer Actualizaciones Batch](#)

[Utilizar Tipos de Datos SQL3](#)

[Características de Extensión Estándar](#)

Configuración del API JDBC 2.0

Si queremos ejecutar código que emplee alguna de las características del JDBC 2.0, necesitamos hacer lo siguiente:

1. Descargar el JDK 1.2 siguiendo las instrucciones de descarga
2. Instalar un Driver JDBC que implemente las características del JDBC 2.0 utilizadas en el código.
3. Acceder a un controlador de base de datos que implemente las características del JDBC utilizadas en el código.

En el momento de escribir esto, no se había implementado ningún driver que soportara las nuevas características, pero hay muchos en desarrollo. Como consecuencia, no es posible probar el código de demostración de las características del JDBC 2.0. Podemos aprender de los ejemplos, pero no se asegura que éstos funcionen.

Mover el Cursor en una Hoja de Resultados

Una de las nuevas características del API JDBC 2.0 es la habilidad de mover el cursor en una hoja de resultados tanto hacia atrás como hacia adelante. También hay métodos que nos permiten mover el cursor a una fila particular y comprobar la posición del cursor. La hoja de resultados Scrollable hace posible crear una herramienta GUI (Interface Gráfico de Usuario) para navegar a través de ella, lo que probablemente será uno de los principales usos de esta característica. Otro uso será movernos a una fila para actualizarla.

Antes de poder aprovechar estas ventajas, necesitamos crear un objeto ResultSet Scrollable:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                     ResultSet.CONCUR_READ_ONLY);  
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

Este código es similar al utilizado anteriormente, excepto en que añade dos argumentos al método createStatement. El primer argumento es una de las tres constantes añadidas al API ResultSet para indicar el tipo de un objeto ResultSet: TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, y TYPE_SCROLL_SENSITIVE. El segundo argumento es una de las dos constantes de ResultSet para especificar si la hoja de resultados es de sólo lectura o actualizable: CONCUR_READ_ONLY y CONCUR_UPDATABLE. Lo que debemos recordar aquí es que si especificamos un tipo, también debemos especificar si es de sólo lectura o actualizable. También, debemos especificar primero el tipo, y como ambos parámetros son int, el compilador no comprobará si los hemos intercambiado.

Especificando la constante TYPE_FORWARD_ONLY se crea una hoja de resultados no desplazable, es decir, una hoja en la que el cursor sólo se mueve hacia adelante. Si no se especifican constantes para el tipo y actualización de un objeto ResultSet, obtendremos automáticamente una TYPE_FORWARD_ONLY y CONCUR_READ_ONLY (exactamente igual que en el API del JDBC 1.0).

Obtendremos un objeto ResultSet desplazable si utilizamos una de estas constantes: TYPE_SCROLL_INSENSITIVE o TYPE_SCROLL_SENSITIVE. La diferencia entre estas dos es si la hoja de resultados refleja los cambios que se han hecho mientras estaba abierta y si se puede llamar a ciertos métodos para detectar estos cambios. Generalmente hablando, una hoja de resultados TYPE_SCROLL_INSENSITIVE no refleja los cambios hechos mientras estaba abierta y en una hoja TYPE_SCROLL_SENSITIVE si se reflejan. Los tres tipos de hojas de resultados harán visibles los resultados si se cierran y se vuelve a abrir. En este momento, no necesitamos preocuparnos de los puntos delicados de las capacidades de un objeto ResultSet, entraremos en más detalle más adelante. Aunque deberíamos tener en mente el hecho de que no importa el tipo de hoja de resultados que especifiquemos, siempre estaremos limitados por nuestro controlador de base de datos y el driver utilizados.

Una vez que tengamos un objeto ResultSet desplazable, srs en el ejemplo anterior, podemos utilizarlo para mover el cursor sobre la hoja de resultados. Recuerda que cuando creabamos un objeto ResultSet anteriormente, tenía el cursor posicionado antes de la primera fila. Incluso aunque una hoja de resultados se seleccione desplazable, el cursor también se posiciona inicialmente delante de la primera fila. En el API JDBC 1.0, la única forma de mover el cursor era llamar al método next. Este método todavía es apropiado si queremos acceder a las filas una a una, yendo de la primera fila a la última, pero ahora tenemos muchas más formas para mover el cursor.

La contrapartida del método next, que mueve el cursor una fila hacia delante (hacia el final de la hoja de resultados), es el nuevo método previous, que mueve el cursor una fila hacia atrás (hacia el inicio de la hoja de resultados). Ambos métodos devuelven false cuando el cursor se sale de la hoja de resultados (posición antes de la primera o después de la última fila), lo que hace posible utilizarlos en un bucle while. Ya hemos utilizado un método next en un bucle while, pero para refrescar la memoria, aquí tenemos un ejemplo que mueve el cursor a la primera fila y luego a la

siguiente cada vez que pasa por el bucle while. El bucle termina cuando alcanza la última fila, haciendo que el método next devuelva false. El siguiente fragmento de código imprime los valores de cada fila de srs, con cinco espacios en blanco entre el nombre y el precio:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
while (srs.next()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "      " + price);
}
```

La salida se podría parecer a esto:

```
Colombian      7.99
French_Roast    8.99
Espresso       9.99
Colombian_Decaf 8.99
French_Roast_Decaf 9.99
```

Al igual que en el fragmento anterior, podemos procesar todas las filas de srs hacia atrás, pero para hacer esto, el cursor debe estar detrás de la última fila. Se puede mover el cursor explícitamente a esa posición con el método afterLast. Luego el método previous mueve el cursor desde la posición detrás de la última fila a la última fila, y luego a la fila anterior en cada iteración del bucle while. El bucle termina cuando el cursor alcanza la posición anterior a la primera fila, cuando el método previous devuelve false.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "      " + price);
}
```

La salida se podría parecer a esto:

```
French_Roast_Decaf 9.99
Colombian_Decaf    8.99
Espresso           9.99
French_Roast       8.99
Colombian          7.99
```

Como se puede ver, las dos salidas tienen los mismos valores, pero las filas están en orden inverso.

Se puede mover el cursor a una fila particular en un objeto ResultSet. Los métodos first, last, beforeFirst, y afterLast mueven el cursor a la fila indicada en sus nombres. El método absolute moverá el cursor al número de fila indicado en su argumento. Si el número es positivo, el cursor se mueve al número dado desde el principio, por eso llamar a absolute(1) pone el cursor en la primera fila. Si el número es negativo, mueve el cursor al número dado desde el final, por eso llamar a absolute(-1) pone el cursor en la última fila. La siguiente línea de código mueve el cursor a la cuarta fila de srs:

```
srs.absolute(4);
```

Si srs tuviera 500 filas, la siguiente línea de código movería el cursor a la fila 497:

```
srs.absolute(-4);
```

Tres métodos mueven el cursor a una posición relativa a su posición actual. Como hemos podido ver, el método `next` mueve el cursor a la fila siguiente, y el método `previous` lo mueve a la fila anterior. Con el método `relative`, se puede especificar cuántas filas se moverá desde la fila actual y también la dirección en la que se moverá. Un número positivo mueve el cursor hacia adelante el número de filas dado; un número negativo mueve el cursor hacia atrás el número de filas dado. Por ejemplo, en el siguiente fragmento de código, el cursor se mueve a la cuarta fila, luego a la primera y por último a la tercera:

```
srs.absolute(4); // cursor está en la cuarta fila
...
srs.relative(-3); // cursor está en la primera fila
...
srs.relative(2); // cursor está en la tercera fila
```

El método `getRow` permite comprobar el número de fila donde está el cursor. Por ejemplo, se puede utilizar `getRow` para verificar la posición actual del cursor en el ejemplo anterior:

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum debería ser 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum debería ser 1
srs.relative(2);
int rowNum = srs.getRow(); // rowNum debería ser 3
```

Existen cuatro métodos adicionales que permiten verificar si el cursor se encuentra en una posición particular. La posición se indica en sus nombres: `isFirst`, `isLast`, `isBeforeFirst`, `isAfterLast`. Todos estos métodos devuelven un boolean y por lo tanto pueden ser utilizados en una sentencia condicional. Por ejemplo, el siguiente fragmento de código comprueba si el cursor está después de la última fila antes de llamar al método `previous` en un bucle `while`. Si el método `isAfterLast` devuelve `false`, el cursor no estará después de la última fila, por eso se llama al método `afterLast`. Esto garantiza que el cursor estará después de la última fila antes de utilizar el método `previous` en el bucle `while` para cubrir todas las filas de `srs`.

```
if (srs.isAfterLast() == false) {
    srs.afterLast();
}
while (srs.previous()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "      " + price);
}
```

En la siguiente página, veremos cómo utilizar otros dos métodos de `ResultSet` para mover el cursor, `moveToInsertRow` y `moveToCurrentRow`. También veremos ejemplos que ilustran por qué podríamos querer mover el cursor a ciertas posiciones.

Hacer Actualizaciones en una Hoja de Resultados Actualizable

Otra nueva característica del API JDBC 2.0 es la habilidad de actualizar filas en una hoja de resultados utilizando métodos Java en vez de tener que enviar comandos SQL. Pero antes de poder aprovechar esta capacidad, necesitamos crear un objeto `ResultSet` actualizable. Para hacer esto, suministramos la constante `CONCUR_UPDATABLE` de `ResultSet` al método `createStatement`, como se ha visto en ejemplos anteriores. El objeto `Statement` creado producirá un objeto `ResultSet` actualizable cada vez que se ejecute una petición. El siguiente fragmento de código ilustra la creación de un objeto `ResultSet` actualizable, `uprs`. Observa que el código también lo hace desplazable. Un objeto `ResultSet` actualizable no tiene porque ser desplazable, pero cuando se hacen cambios en una hoja de resultados, generalmente queremos poder movernos por ella. Con una hoja de resultados desplazable, podemos movernos a las filas que queremos cambiar, y si el tipo es `TYPE_SCROLL_SENSITIVE`, podemos obtener el nuevo valor de una fila después de haberlo cambiado.

```
Connection con = DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

El objeto `ResultSet`, `uprs` resultante se podría parecer a esto:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

Podemos utilizar los nuevos métodos del JDBC 2.0 en el interface `ResultSet` para insertar una nueva fila en `uprs`, borrar una fila de `uprs`, o modificar un valor de una columna de `uprs`.

Actualizar una Hoja de Resultados Programáticamente

Una actualización es la modificación del valor de una columna de la fila actual. Supongamos que queremos aumentar el precio del café "French Roast Decaf" a 10.99. utilizando el API JDBC 1.0, la actualización podría ser algo como esto:

```
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" +  
                  "WHERE COF_NAME = FRENCH_ROAST_DECAF");
```

El siguiente fragmento de código muestra otra forma de realizar la actualización, esta vez utilizando el API JDBC 2.0:

```
uprs.last();  
uprs.updateFloat("PRICE", 10.99);
```

Las operaciones de actualización en el API JDBC 2.0 afectan a los valores de columna de la fila en la que se encuentra el cursor, por eso en la primera línea se llama al método `last` para mover el cursor a la última fila (la fila donde la columna `COF_NAME` tiene el valor `FRENCH_ROAST_DECAF`). Una vez situado el cursor, todos los métodos de actualización que llamemos operarán sobre esa fila hasta que movamos el cursor a otra fila. La segunda línea de código cambia el valor de la columna `PRICE` a 10.99 llamando al método `updateFloat`. Se utiliza este método porque el valor de la columna que queremos actualizar es un float Java.

Los métodos `updateXXX` de `ResultSet` toman dos parámetros: la columna a actualizar y el nuevo valor a colocar en ella. Al igual que en los métodos `getXXX` de `ResultSet`, el parámetro que designa la columna podría ser el nombre de la columna o el número de la columna. Existe un método `updateXXX` diferente para cada tipo (`updateString`, `updateBigDecimal`, `updateInt`, etc.)

En este punto, el precio en `uprs` para "French Roast Decaf" será 10.99, pero el precio en la tabla `COFFEES` de la base de datos será todavía 9.99. Para que la actualización tenga efecto en la base de datos y no sólo en la hoja de resultados, debemos llamar al método `updateRow` de `ResultSet`. Aquí está el código para actualizar tanto `uprs` como `COFFEES`:

```
uprs.last();  
uprs.updateFloat("PRICE", 10.99);  
uprs.updateRow();
```

Si hubiéramos movido el cursor a una fila diferente antes de llamar al método `updateRow`, la actualización se habría perdido. Si, por el contrario, nos damos cuenta de que el precio debería haber sido 10.79 en vez de 10.99 podríamos haber cancelado la actualización llamando al método `cancelRowUpdates`. Tenemos que llamar al método `cancelRowUpdates` antes de llamar al método `updateRow`;

una vez que se llama a `updateRow`, llamar a `cancelRowUpdates` no hará nada. Observa que `cancelRowUpdates` cancela todas las actualizaciones en una fila, por eso, si había muchas llamadas a método `updateXXX` en la misma fila, no podemos cancelar sólo una de ellas. El siguiente fragmento de código primero cancela el precio 10.99 y luego lo actualiza a 10.79:

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
```

En este ejemplo, sólo se había actualizado una columna, pero podemos llamar a un método `updateXXX` apropiado para cada una de las columnas de la fila. El concepto a recordar es que las actualizaciones y las operaciones relacionadas se aplican sobre la fila en la que se encuentra el cursor. Incluso si hay muchas llamadas a métodos `updateXXX`, sólo se hace una llamada al método `updateRow` para actualizar la base de datos con todos los cambios realizados en la fila actual.

Si también queremos actualizar el precio de `COLOMBIAN_DECAF`, tenemos que mover el cursor a la fila que contiene ese café. Como la fila de `COLOMBIAN_DECAF` precede inmediatamente a la fila de `FRENCH_ROAST_DECAF`, podemos llamar al método `previous` para posicionar el cursor en la fila de `COLOMBIAN_DECAF`. El siguiente fragmento de código cambia el precio de esa fila a 9.79 tanto en la hoja de resultados como en la base de datos:

```
uprs.previous();
uprs.updateFloat("PRICE", 9.79);
uprs.updateRow();
```

Todos los movimientos de cursor se refieren a filas del objeto `ResultSet`, no a filas de la tabla de la base de datos. Si una petición selecciona cinco filas de la tabla de la base de datos, habrá cinco filas en la hoja de resultados, con la primera fila siendo la fila 1, la segunda siendo la fila 2, etc. La fila 1 puede ser identificada como la primera, y, en una hoja de resultados con cinco filas, la fila 5 será la última.

El orden de las filas en la hoja de resultados no tiene nada que ver con el orden de las filas en la tablas de la base de datos. De hecho, el orden de las filas en la tabla de la base de datos es indeterminado. El controlador de la base de datos sigue la pista de las filas seleccionadas, y hace las actualizaciones en la fila apropiada, pero podrían estar localizadas en cualquier lugar de la tabla. Cuando se inserta una fila, por ejemplo, no hay forma de saber donde será insertada dentro de la tabla.

Insertar y Borrar Filas Programáticamente

En la sección anterior hemos visto cómo modificar el valor de una columna utilizando métodos del API JDBC 2.0 en vez de utilizar comandos SQL. Con el API JDBC 2.0 también podemos insertar una fila en una tabla o borrar una fila existente programáticamente.

Supongamos que nuestro propietario del café ha obtenido una nueva variedad de café de uno de sus suministradores. El "High Ground", y quiere añadirlo a su base de datos. Utilizando el API JDBC 1,0 podría escribir el código que pasa una sentencia insert de SQL al controlador de la base de datos. El siguiente fragmento de código, en el que stmt es un objeto Statement, muestra esta aproximación:

```
stmt.executeUpdate("INSERT INTO COFFEES " +  
                  "VALUES ('Kona', 150, 10.99, 0, 0)");
```

Se puede hacer esto mismo sin comandos SQL utilizando los métodos de ResultSet del API JDBC 2.0. Básicamente, después de tener un objeto ResultSet con los resultados de la tabla COFFEES, podemos construir una nueva fila insertándola tanto en la hoja de resultados como en la tabla COFFEES en un sólo paso. Se construye una nueva fila en una llamada "fila de inserción", una fila especial asociada con cada objeto ResultSet. Esta fila realmente no forma parte de la hoja de resultados; podemos pensar en ella como un buffer separado en el que componer una nueva fila.

El primer paso será mover el cursor a la fila de inserción, lo que podemos hacer llamando al método moveToInsertRow. El siguiente paso será seleccionar un valor para cada columna de la fila. Hacemos esto llamando a los métodos updateXXX apropiados para cada valor. Observa que estos son los mismos métodos updateXXX utilizados en la página anterior para cambiar el valor de una columna. Finalmente, podemos llamar al método insertRow para insertar la fila que hemos rellenado en la hoja de resultados. Este único método inserta la fila simultáneamente tanto en el objeto ResultSet como en la tabla de la base de datos de la que la hoja de datos fue seleccionada.

El siguiente fragmento de código crea un objeto ResultSet actualizable y desplazable, uprs, que contiene todas las filas y columnas de la tabla COFFEES:

```
Connection con = DriverManager.getConnection("jdbc:mySubprotocol:mySubName");  
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                     ResultSet.CONCUR_UPDATABLE);  
ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
```

El siguiente fragmento de código utiliza el objeto ResultSet, uprs para insertar la fila para "kona", mostrada en el ejemplo SQL. Mueve el cursor a la fila de inserción, selecciona los cinco valores de columna e inserta la fila dentro de uprs y COFFEES:

```
uprs.moveToInsertRow();  
uprs.updateString("COF_NAME", "Kona");  
uprs.updateInt("SUP_ID", 150);  
uprs.updateFloat("PRICE", 10.99);  
uprs.updateInt("SALES", 0);  
uprs.updateInt("TOTAL", 0);  
uprs.insertRow();
```

Como podemos utilizar el nombre o el número de la columna para indicar la columna seleccionada, nuestro código para seleccionar los valores de columna se podría parecer a esto:

```
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
```

Podríamos habernos preguntado por qué los métodos updateXXX parecen tener un comportamiento distinto a como lo hacían en los ejemplos de actualización. En aquellos ejemplos, el valor seleccionado con un método updateXXX reemplazaba inmediatamente el valor de la columna en la hoja de resultados. Esto era porque el cursor estaba sobre una fila de la hoja de resultados. Cuando el cursor está sobre la fila de inserción, el valor seleccionado con un método updateXXX también es automáticamente seleccionado, pero lo es en la fila de inserción y no en la propia hoja de resultados. Tanto en actualizaciones como en inserciones, llamar a los métodos updateXXX no afectan a la tabla de la base de datos. Se debe llamar al método updateRow para hacer que las actualizaciones ocurran en la base de datos. Para las inserciones, el método insertRow inserta la nueva fila en la hoja de resultados y en la base de datos al mismo tiempo.

Podríamos preguntarnos que sucedería si insertáramos una fila pero sin suministrar los valores para cada columna. Si no suministramos valores para una columna que estaba definida para aceptar valores NULL de SQL, el valor asignado a esa columna es NULL. Si la columna no acepta valores null, obtendremos una SQLException. Esto también es cierto si falta una columna de la tabla en nuestro objeto ResultSet. En el ejemplo anterior, la petición era SELECT * FROM COFFEES, lo que producía una hoja de resultados con todas las columnas y todas las filas. Cuando queremos insertar una o más filas, nuestra petición no tiene porque seleccionar todas las filas, pero sí todas las columnas. Especialmente si nuestra tabla tiene cientos o miles de filas, queremos utilizar una cláusula WHERE para limitar el número de filas devueltas por la sentencia SELECT.

Después de haber llamado al método insertRow, podemos construir otra fila, o podemos mover el cursor de nuevo a la hoja de resultados. Por ejemplo, podemos llamar a cualquier método que ponga el cursor en una fila específica, como first, last, beforeFirst, afterLast, y absolute. También podemos utilizar los métodos previous, relative, y moveToCurrentRow. Observa que sólo podemos llamar a moveToCurrentRow cuando el cursor está en la fila de inserción.

Cuando llamamos al método moveToInsertRow, la hoja de resultados graba la fila en la que se encontraba el cursor, que por definición es la fila actual. Como consecuencia, el método moveToCurrentRow puede mover el cursor desde la fila de inserción a la fila en la que se encontraba anteriormente. Esto también explica porque podemos utilizar los métodos previous y relative, que requieren movimientos relativos a la fila actual.

Código de Ejemplo para Insertar una Fila

El siguiente ejemplo de código es un programa completo que debería funcionar si tenemos un driver JDBC 2.0 que implemente una hoja de resultados desplazable.

Hay algunas cosas que podríamos observar sobre el código:

1. El objeto `ResultSet`, `uprs` es actualizable, desplazable y sensible a los cambios hechos por ella y por otros. Aunque es `TYPE_SCROLL_SENSITIVE`, es posible que los métodos `getXXX` llamados después de las inserciones no recuperen los valores de la nuevas filas. Hay métodos en el interface `DatabaseMetaData` que nos dirán qué es visible y qué será detectado en los diferentes tipos de hojas de resultados para nuestro driver y nuestro controlador de base de datos.
2. Después de haber introducido los valores de una fila con los métodos `updateXXX`, el código inserta la fila en la hoja de resultados y en la base de datos con el método `insertRow`. Luego, estando todavía en la "fila de inserción", selecciona valores para otra nueva fila.

```
import java.sql.*;

public class InsertRows {
    public static void main(String args[]) {
        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        Statement stmt;

        try {
            Class.forName("myDriver.ClassName");
        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(url, "myLogin", "myPassword");
            stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);

            ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
            uprs.moveToInsertRow();
            uprs.updateString("COF_NAME", "Kona");
            uprs.updateInt("SUP_ID", 150);
            uprs.updateFloat("PRICE", 10.99f);
            uprs.updateInt("SALES", 0);
            uprs.updateInt("TOTAL", 0);
            uprs.insertRow();
            uprs.updateString("COF_NAME", "Kona_Decaf");
            uprs.updateInt("SUP_ID", 150);
            uprs.updateFloat("PRICE", 11.99f);
            uprs.updateInt("SALES", 0);
            uprs.updateInt("TOTAL", 0);
            uprs.insertRow();
            uprs.beforeFirst();
            System.out.println("Table COFFEES after insertion:");
            while (uprs.next()) {
                String name = uprs.getString("COF_NAME");
                int id = uprs.getInt("SUP_ID");
                float price = uprs.getFloat("PRICE");
                int sales = uprs.getInt("SALES");
            }
        }
    }
}
```

```
        int total = uprs.getInt("TOTAL");
        System.out.print(name + "    " + id + "    " + price);
        System.out.println("    " + sales + "    " + total);
    }

    uprs.close();
    stmt.close();
    con.close();

} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}
}
```

Borrar una Fila

Hasta ahora, hemos visto cómo actualizar un valor y cómo insertar una nueva fila. Borrar una fila es la tercera forma de modificar un objeto ResultSet, y es la más simple. Todo lo que tenemos que hacer es mover el cursor a la fila que queremos borrar y luego llamar al método `deleteRow`. Por ejemplo, si queremos borrar la cuarta fila de la hoja de resultados `uprs`, nuestro código se parecería a esto:

```
uprs.absolute(4);  
uprs.deleteRow();
```

La cuarta fila ha sido eliminada de `uprs` y de la base de datos.

El único problema con las eliminaciones es lo que ResultSet realmente hace cuando se borra una fila. Con algunos driver JDBC, una línea borrada es eliminada y ya no es visible en una hoja de resultados. Algunos drivers JDBC utilizan una fila en blanco en su lugar (un "hole") donde la fila borrada fuera utilizada. Si existe una fila en blanco en lugar de la fila borrada, se puede utilizar el método `absolute` con la posición original de la fila para mover el cursor, porque el número de filas en la hoja de resultados no ha cambiado.

En cualquier caso, deberíamos recordar que los drivers JDBC manejan las eliminaciones de forma diferente. Por ejemplo, si escribimos una aplicación para ejecutarse con diferentes bases de datos, no deberíamos escribir código que dependiera de si hay una fila vacía en la hoja de resultados.

Ver los cambios en una Hoja de Resultados

Si modificamos los datos en un objeto ResultSet, los cambios se harán visibles si lo cerramos y lo abrimos de nuevo. En otras palabras, si re-ejecutamos la misma petición, producirá una nueva hoja de resultados, basada en los datos actuales de la tabla. Esta hoja de resultados reflejara naturalmente los cambios que hayamos hecho anteriormente.

La cuestión es si podemos ver los cambios que hayamos realizado mientras el objeto ResultSet esté todavía abierto. (Generalmente, estaremos más interesados en los cambios hechos por otros). La respuesta depende del controlador de la base de datos, del driver, y del tipo del objeto ResultSet utilizado.

Con un objeto ResultSet que sea `TYPE_SCROLL_SENSITIVE`, siempre podremos ver las actualizaciones que alguien haga en los valores de las columnas. Normalmente veremos inserciones y eliminaciones, pero la única forma de estar seguros es utilizar los métodos `DatabaseMetaData` que devuelven esta información.

Podemos regular la extensión de que los cambios sean visibles aumentando o bajando el nivel de aislamiento de la transacción con la base de datos. Por ejemplo, la siguiente línea de código, donde `con` es un objeto Connection activo, selecciona el nivel de aislamiento de la conexión a `TRANSACTION_READ_COMMITTED`:


```
con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);
```

Con este nivel de aislamiento, nuestro objeto `ResultSet` no mostrará ningún cambio antes de ser enviado, pero puede mostrar los cambios que podrían tener problemas de consistencia. Para permitir menores niveles de inconsistencia, podríamos subir el nivel de aislamiento a `TRANSACTION_REPEATABLE_READ`. El problema es que a niveles más altos de aislamiento, el rendimiento se empobrece. Y siempre estamos limitados por lo que proporcionan las bases de datos y los drivers.

En un objeto `ResultSet` que sea `TYPE_SCROLL_INSENSITIVE`, generalmente no podremos ver los cambios hechos mientras esté abierta. Algunos programadores utilizan sólo este tipo de objeto `ResultSet` porque quieren una vista consistente de los datos y no quieren ver los cambios hechos por otros.

Se puede utilizar el método `refreshRow` para obtener los últimos valores de una fila en la base de datos. Este método puede utilizar muchos recursos, especialmente si el controlador de la base de datos devuelve múltiples filas cada vez que se llama a `refreshRow`. De todas formas, puede utilizarse cuando es crítico tener los últimos datos. Incluso aunque una hoja de resultados sea sensible y los cambios sean visibles, una aplicación no podría ver siempre los últimos cambios si el driver recupera varias filas a la vez y las almacena. Por eso, utilizar el método `refreshRow` es el único método para asegurarnos que estamos viendo los últimos datos.

El siguiente código ilustra cómo una aplicación podría utilizar el método `refreshRow` cuando es absolutamente crítico ver los últimos valores. Observa que la hoja de resultados debería ser sensible; si queremos utilizar el método `refreshRow` con un objeto `ResultSet` que sea `TYPE_SCROLL_INSENSITIVE`, no hará nada. (La urgencia de obtener los últimos datos es bastante improbable en la tabla `COFFEES`, pero la fortuna de un inversor, depende de conocer los últimos precios en la amplia fluctuación del mercado del café. O, por ejemplo, querriamos asegurarnos de que el nuestro asiento reservado en el avión de regreso todavía está disponible).

```
Statement stmt = con.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(SELECT COF_NAME, PRICE FROM COFFEES);
uprs.absolute(4);
Float price1 = uprs.getFloat("PRICE");
// do something. . .
uprs.absolute(4);
uprs.refreshRow();
Float price2 = uprs.getFloat("PRICE");
if (price2 > price1) {
    // do something. . .
}
```

Hacer Actualizaciones por Lotes

Una actualización por lotes es un conjunto de varias sentencias de actualización que son enviadas a la base de datos para ser procesadas como un lote. Enviar múltiples sentencias de actualización juntas a la base de datos puede, en algunas situaciones, ser mucho más eficiente que enviar cada sentencia separadamente. Esta posibilidad de enviar actualizaciones como una unidad, referida como facilidad de actualización por lotes, es una de las características proporcionadas por el API JDBC 2.0.

Utilizar Objetos Statement para Actualizaciones por Lotes

En el API JDBC 1.0, los objetos Statement enviaban actualizaciones a la base de datos individualmente con el método `executeUpdate`. Se pueden enviar varias sentencias `executeUpdate` en la misma transacción, pero aunque son enviadas como una unidad, son procesadas individualmente. Los interfaces derivados de Statement, `PreparedStatement` y `CallableStatement`, tienen las mismas capacidades, utilizando sus propias versiones de `executeUpdate`.

Con el API JDBC 2.0, los objetos Statement, `PreparedStatement` y `CallableStatement` tienen la habilidad de mantener una lista de comandos que pueden ser enviados juntos como un lote. Ellos son creados con una lista asociada, que inicialmente está vacía. Se pueden añadir comandos SQL a esta lista con el método `addBatch`, y podemos vaciar la lista con el método `clearBatch`. Todos los comandos de la lista se envían a la base de datos con el método `executeBatch`. Ahora veamos como funcionan estos métodos.

Supongamos que nuestro propietario del café quiere traer nuevos cafés. Ha determinado que su mejor fuente es uno de sus actuales suministradores, Superior Coffee, y quiere añadir cuatro nuevos cafés a la tabla COFFEES. Como sólo va a insertar cuatro nuevas filas, la actualización por lotes podría no aumentar el rendimiento significativamente, pero es una buena oportunidad para demostrar la actualización por lotes. Recordamos que la tabla COFFEES tiene cinco columnas: COF_NAME del tipo VARCHAR(32), SUP_ID del tipo INTEGER, PRICE del tipo FLOAT, SALES del tipo INTEGER, y TOTAL del tipo INTEGER. Cada fila insertada tendrá valores para las cinco columnas en orden. El código para insertar una nueva fila como un lote se podría parecer a esto:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO COFFEES" +
              "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES" +
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES" +
              "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
```

```
stmt.addBatch("INSERT INTO COFFEES" +  
              "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");  
int [] updateCounts = stmt.executeBatch();
```

Ahora examinemos el código línea por línea.

```
con.setAutoCommit(false);
```

Esta línea desactiva el modo auto-commit para el objeto Connection, con para que la transacción no sea enviada o anulada automáticamente cuando se llame al método executeBatch. (Si no recuerdas qué era una transacción, deberías revisar la página [Transacciones](#)). Para permitir un manejo de errores correcto, también deberíamos actualizar el modo auto-commit antes de empezar una actualización por lotes.

```
Statement stmt = con.createStatement();
```

Esta línea de código crea el objeto Statement, stmt. Al igual que todos los nuevos objetos Statement recién creados, stmt tiene una lista de comandos asociados y ésta lista está vacía.

```
stmt.addBatch("INSERT INTO COFFEES" +  
              "VALUES('Amaretto', 49, 9.99, 0, 0)");  
stmt.addBatch("INSERT INTO COFFEES" +  
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");  
stmt.addBatch("INSERT INTO COFFEES" +  
              "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");  
stmt.addBatch("INSERT INTO COFFEES" +  
              "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");
```

Cada una de estas líneas de código añade un comando a la lista de comandos asociados con stmt. Estos comandos son todas sentencias INSERT INTO, cada una añade una nueva fila que consiste en cinco valores de columna. Los valores para las columnas COF_NAME y PRICE se explican a sí mismos. el segundo valor de cada fila es 49 porque es el número de identificación del suministrador, Superior Coffee. Los últimos dos valores, las entradas para las columnas SALES y TOTAL, todas empiezan siendo cero porque todavía no se ha vendido nada. (SALES es el número de libras del café de esa columna vendidas la semana actual; y TOTAL es el número total de libras vendidas de este café).

```
int [] updateCounts = stmt.executeBatch();
```

En esta línea, stmt envía a la base de datos los cuatro comandos SQL que fueron añadidos a su lista de comandos para que sean ejecutados como un lote. Observa que stmt utiliza el método executeBatch para el lote de inserciones, no el método executeUpdate, que envía sólo un comando y devuelve una sólo cuenta de actualización. El controlador de la base de datos ejecuta los comandos en el orden en que fueron añadidos a la lista de comandos, por eso primero añadirá la

fila de valores apra Amaretto, luego añade la fila de Hazelnut, luego Amaretto decaf, y finalmente Hazelnut decaf. Si los cuatro comandos se ejecutan satisfactoriamente, el controlador de la base de datos devolverá una cuenta de actualización para cada comando en el orden en que fue ejecutado. Las cuentas de actualización, indicarán cuántas líneas se vieron afectadas por cada comando, y se almacenan en el array de enteros updateCounts.

En este punto, updateCounts debería contener cuatro elementos del tipo int. En este caso, cada uno de ellos será 1 porque una inserción afecta a un fila. La lista de comandos asociados con stmt ahora estará vacía porque los cuatro comandos añadidos anteriormente fueron enviados a la base de datos cuando stmt llamó al método executeBatch. En cualquier momento podemos vaciar la lista de comandos con el método clearBatch.

Excepciones en las Actualizaciones por Lotes

Existen dos excepciones que pueden ser lanzadas durante una actualización por lotes: SQLException y BatchUpdateException.

Todos los métodos del API JDBC lanzarán un objeto SQLException si existe algún problema de acceso a la base de datos. Además, el método executeBatch lanzará una SQLException si hemos utilizado el método addBatch para añadir un comando y devuelve una hoja de resultados al lote de comandos que está siendo ejecutado. Típicamente una petición (una sentencia SELECT) devolverá una hoja de resultados, pero algunos métodos, como algunos de DatabaseMetaData pueden devolver una hoja de datos.

Sólo utilizar el método addBatch para añadir un comando que produce una hoja de resultados no hace que se lance una excepción. No hay problema mientras el comando está siendo situado en la lista de comandos de un objeto Statement. Pero habrá problemas cuando el método executeBatch envíe el lote al controlador de la base de datos para ejecutarlo. Cuando se ejecuta cada comando, debe devolver una cuenta de actualización que pueda ser añadida al array de cuentas de actualización devuelto por el método executeBatch. Intentar poner una hoja de resultados en un array de cuentas de actualización causará un error y hará que executeBatch lance una SQLException. En otras palabras, sólo los comandos que devuelven cuentas de actualización (comandos como INSERT INTO, UPDATE, DELETE, CREATE TABLE, DROP TABLE, ALTER TABLE, etc) pueden ser ejecutados como un lote con el método executeBatch.

Si no se lanzó una SQLException, sabremos que no hubo problemas de acceso y que todos los comandos produjeron cuentas de actualización. Si uno de los comandos no puede ser ejecutado por alguna razón, el método executeBatch lanzará una BatchUpdateException. Además de la información que tienen todas las excepciones, este excepción contiene un array de cuentas de actualización para los comandos que se ejecutaron satisfactoriamente antes de que se lanzara la excepción. Cómo las cuentas de actualización están en el mismo orden que los

comandos que las produjeron, podremos decir cuántos comandos fueron ejecutados y cuáles fueron.

BatchUpdateException descende de SQLException. Esto significa que utiliza todos los métodos disponibles en un objeto SQLException. El siguiente fragmento de código imprime la información de SQLException y las cuentas de actualización contenidas en un objeto BatchUpdateException. Como getUpdateCounts devuelve un array de int, utiliza un bucle for para imprimir todas las cuentas de actualización.

```
try {
    // make some updates
} catch (BatchUpdateException b) {
    System.err.println("SQLException: " + b.getMessage());
    System.err.println("SQLState:   " + b.getSQLState());
    System.err.println("Message:   " + b.getMessage());
    System.err.println("Vendor:    " + b.getErrorCode());
    System.err.print("Update counts:  ");
    int [] updateCounts = b.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + "    ");
    }
}
```

Para ver un programa completo de actualización por lotes, puedes ver [BatchUpdate.java](#). El código pone juntos los fragmentos de código de las páginas anteriores y crea un programa completo. Podrías observar que hay dos bloques catch al final del código. Si hay un objeto BatchUpdateException el primer bloque lo capturará. El segundo bloque capturará un objeto SQLException que no sea un objeto BatchUpdateException.

Utilizar Tipos de Datos SQL3

Los tipos de datos comunmente referidos como tipos SQL3 son los nuevos tipos de datos que están siendo adoptados en la nueva versión del estándar ANSI/ISO de SQL. El JDBC 2.0 proporciona interfaces que representan un mapeado de estos tipos de datos SQL3 dentro del lenguaje Java. Con estos nuevos interfaces, podremos trabajar con tipos SQL3 igual que con otros tipos.

Los nuevos tipos SQL3 le dan a una base de datos relacional más flexibilidad en lo que pueden utiizar como tipos para una columna de una tabla. Por ejemplo, una columna podría ahora almacenar el nuevo tipo de dato BLOB (Binary Large Object), que puede almacenar grandes cantidades de datos como una fila de bytes. Una columna también puede ser del tipo CLOB (Character Large Object), que es capaz de almacenar grandes cantidades de datos en formato caracter. El nuevo tipo ARRAY hace posible el uso de un array como un valor de columna. Incluso las nuevas estructuras de tipos-definidos-por-el-usuario (UDTs) de SQL pueden almacenarse como valores de columna.

La siguiente lista tiene los interfaces del JDBC 2.0 que mapean los tipos SQL3. Los explicaremos en más detalle más adelante.

- Un ejemplar Blob mapea un BLOB de SQL.
- Un ejemplar Clob mapea un CLOB de SQL.
- Un ejempar Array mapea un ARRAY de SQL.
- Un ejemplar Struct mapea un tipo Estructurado de SQL.
- Un ejemplar Ref mapea un REF de SQL.

Uitlizar tipos de datos SQL3

Se recuperan, almacenan, y actualizan tipos de datos SQL3 de la misma forma que los otros tipos. Se utilizan los métodos `ResultSet.getXXX` o `CallableStatement.getXXX` para recuperarlos, los métodos `PreparedStatement.setXXX` para almacenarlos y `updateXXX` para actualizarlos. Probablemente el 90% de las operacuines realizadas con tipos SQL3 implican el uso de los métodos `getXXX`, `setXXX`, y `updateXXX`. La siguiente tabla muestra qué métodos utilizar:

Tipo SQL3	Método getXXX	Método setXXX	Método updateXXX
BLOB	<code>getBlob</code>	<code>setBlob</code>	<code>updateBlob</code>
CLOB	<code>getClob</code>	<code>setClob</code>	<code>updateClob</code>
ARRAY	<code>getArray</code>	<code>setArray</code>	<code>updateArray</code>
Tipo Structured	<code>getObject</code>	<code>setObject</code>	<code>updateObject</code>
REF(Tipo Structured)	<code>getRef</code>	<code>setRef</code>	<code>updateRef</code>

Por ejemplo, el siguiente fragmento de código recupera un valor ARRAY de SQL. Para este ejemplo, la columna SCORES de la tabla STUDENTS contiene valores del tipo ARRAY. La variable `stmt` es un objeto `Statement`.

```
ResultSet rs = stmt.executeQuery("SELECT SCORES FROM STUDENTS WHERE ID = 2238");
rs.next();
Array scores = rs.getArray("SCORES");
```

La variable `scores` es un puntero lógico al objeto ARRAY de SQL almacenado en la tabla STUDENTS en la fila del estudiante 2238.

Si queremos almacenar un valor en la base de datos, utilizamos el método `setXXX` apropiado. Por ejemplo, el siguiente fragmento de código, en el que `rs` es un objeto `ResultSet`, almacena un objeto Clob:

```
Clob notes = rs.getClob("NOTES");
```

```

PreparedStatement pstmt = con.prepareStatement("UPDATE MARKETS
                                                SET COMMENTS = ? WHERE SALES <
1000000",
                                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                ResultSet.CONCUR_UPDATABLE);

pstmt.setClob(1, notes);

```

Este código configura notes como el primer parámetro de la sentencia de actualización que está siendo enviada a la base de datos. El valor CLOB designado por notes será almacenado en la tabla MARKETS en la columna COMMENTS en cada columna en que el valor de la columna SALES sea menor de un millón.

Objetos Blob, Clob, y Array

Una característica importante sobre los objetos Blob, Clob, y Array es que se pueden manipular sin tener que traer todos los datos desde el servidor de la base de datos a nuestra máquina cliente. Un ejemplar de cualquiera de esos tipos es realmente un puntero lógico al objeto en la base de datos que representa el ejemplar. Como los objetos SQL BLOB, CLOB, o ARRAY pueden ser muy grandes, esta característica puede aumentar drásticamente el rendimiento.

Se pueden utilizar los comandos SQL y los API JDBC 1.0 y 2.0 con objetos Blob, Clob, y Array como si estuviéramos operando realmente con el objeto de la base de datos. Sin embargo, si queremos trabajar con cualquiera de ellos como un objeto Java, necesitamos traer todos los datos al cliente, con lo que la referencia se materializa en el objeto. Por ejemplo, si queremos utilizar un ARRAY de SQL en una aplicación como si fuera un array Java, necesitamos materializar el objeto ARRAY en el cliente para convertirlo en un array de Java. Entonces podremos utilizar los métodos de arrays de Java para operar con los elementos del array. Todos los interfaces Blob, Clob, y Array tienen métodos para materializar los objetos que representan.

Tipos Struct y Distinct

Los tipos estructurados y distinct de SQL son dos tipos de datos que el usuario puede definir. Normalmente nos referimos a ellos como UDTs (user-defined types), y se crean con un sentencia CREATE TYPE de SQL.

Un tipo estructurado de SQL se parece a los tipos estructurados de Java en que tienen miembros, llamados atributos, que puede ser cualquier tipo de datos. De echo, un atributo podría ser a su vez un tipo estructurado. Aquí tienes un ejemplp de una simple definición de un nuevo tipo de dato SQL:

```

CREATE TYPE PLANE_POINT
(
    X FLOAT,
    Y FLOAT
)

```

Al contrario que los objetos Blob, Clob, y Array, un objeto Struct contiene valores para cada uno de los atributos del tipo estructurado de SQL y no es sólo un puntero lógico al objeto en la base de datos. Por ejemplo, supongamos que un objeto PLANE_POINT está almacenado en la columna POINTS de la tabla PRICES.

```

ResultSet rs = stmt.executeQuery("SELECT POINTS FROM PRICES WHERE PRICE > 3000.00");
while (rs.next()) {
    Struct point = (Struct)rs.getObject("POINTS");
    // do something with point
}

```

Si el objeto PLANE_POINT recuperado tiene un valor X de 3 y un valor Y de -5, el objeto Struct,

point contendrá los valores 3 y -5.

Podríamos haber observado que Struct es el único tipo que no tiene métodos `getXXX` y `setXXX` con su nombre como XXX. Debemos utilizar `getObject` y `setObject` con ejemplares Struct. Esto significa que cuando recuperemos un valor utilizando el método `getObject`, obtendremos un Object Java que podremos forzar a Struct, como se ha hecho en el ejemplo de código anterior.

El segundo tipo SQL que un usuario puede definir con una sentencia `CREATE TYPE` de SQL es un tipo `Distinct`. Este tipo se parece al `typedef` de C o C++ en que es un tipo basado en un tipo existente. Aquí tenemos un ejemplo de creación de un tipo `distinct`:

```
CREATE TYPE MONEY AS NUMERIC(10, 2)
```

Esta definición crea un nuevo tipo llamado `MONEY`, que es un número del tipo `NUMERIC` que siempre está en base 10 con dos dígitos después de la coma decimal. `MONEY` es ahora un tipo de datos en el esquema en el que fue definido, y podemos almacenar ejemplares de `MONEY` en una tabla que tenga una columna del tipo `MONEY`.

Un tipo `distinct SQL` se mapea en Java al tipo en el que se mapearía el tipo original. Por ejemplo, `NUMERIC` mapea a `java.math.BigDecimal`, porque el tipo `MONEY` mapea a `java.math.BigDecimal`. Para recuperar un objeto `MONEY`, utilizamos `ResultSet.getBigDecimal` o `CallableStatement.getBigDecimal`; para almacenarlo utilizamos `PreparedStatement.setBigDecimal`.

Características Avanzadas de SQL3

Algunos aspectos del trabajo con los tipos SQL3 pueden parecer bastante complejos. Mencionamos alguna de las características más avanzadas para que las conozcas, pero una explicación profunda no es apropiada para un tutorial básico.

El interface `Struct` es el mapeo estándar para un tipo estructurado de SQL. Si queremos trabajar fácilmente con un tipo estructurado de Java, podemos mapearlo a una clase Java. El tipo estructurado se convierte en una clase, y sus atributos en campos. No tenemos que utilizar un mapeado personalizado, pero normalmente es más conveniente.

Algunas veces podríamos querer trabajar con un puntero lógico a un tipo estructurado de SQL en vez de con todos los valores contenidos en el propio tipo. Esto podría suceder, por ejemplo, si el tipo estructurado tiene muchos atributos o si los atributos son muy grandes. Para referenciar un tipo estructurado, podemos declarar un tipo `REF` de SQL que represente un tipo estructurado particular. Un objeto `REF` de SQL se mapea en un objeto `Ref` de Java, y podemos operar con ella como si lo hicieramos con el objeto del tipo estructurado al que representa.

Características de Extensión Estándard

El paquete javax.sql es una extensión estándar al lenguaje Java. La especificación final no está terminada pero podemos ver algunas de las funcionalidades básicas que proporcionará. Estas son las características de la extensión estándar del JDBC 2.0:

Rowsets

Este objeto encapsula un conjunto de filas de una hoja de resultados y podría mantener abierta la conexión con la base de datos o desconectarse de la fuente. Un rowset es un componente JavaBean; puede ser creado en el momento del diseño y puede utilizarse con otros JavaBeans en una herramienta visual.

JNDI tm para Nombrar Bases de Datos

El interface JNDI (Nombrado y Direccionado) de Java hace posible conectar a una base de datos utilizando un nombre lógico en lugar de codificar un nombre de base de datos y de driver.

Connection Pooling

Un Connection Pool es un caché para conexiones frecuentes que pueden ser utilizadas y reutilizadas, esto recorta la sobrecarga de crear y destruir conexiones a bases de datos.

Soporte de Transacción Distribuida

Este soporte permite al driver JDBC soportar el protocolo estándar de dos-fases utilizados en el API Java Transaction (JTA). Esta característica facilita el uso de las funcionalidades JDBC en componentes JavaBeans de Enterprise.

Utilizar el RMI (Invocación Remota de Métodos)

El sistema de Invocación Remota de Métodos (RMI) de Java permite a un objeto que se está ejecutando en una Máquina Virtual Java (VM) llamar a métodos de otro objeto que está en otra VM diferente.

Nota: RMI proporciona comunicación remota entre programas escritos en Java. Si uno de nuestros programas está escrito en otro lenguaje, deberemos considerar la utilización de [IDL](#) en su lugar.

Esta sección ofrece una breve descripción del sistema RMI que pasea a través de un ejemplo completo cliente/servidor que utiliza las capacidades únicas de RMI para cargar y ejecutar tareas definidas por el usuario en tiempo de ejecución. El servidor del ejemplo implementa un motor de cálculo general. El cliente utiliza el motor de cálculo para calcular el valor del número pi.

[Introducción a las Aplicaciones RMI](#)

Describe el sistema RMI y lista sus ventajas. Además, esta lección proporciona una descripción de una aplicación típica de RMI, compuesta por un servidor y un cliente, y presenta los términos importantes.

[Escribir un Servidor RMI](#)

Muestra el código del servidor del motor de cálculo. A través de este ejemplo, aprenderemos cómo diseñar e implementar un servidor RMI.

[Crear un Programa Cliente](#)

Echa un vistazo a un posible cliente del motor de cálculo y lo utiliza para ilustrar las características importantes de un cliente RMI.

[Compilar y Ejecutar el Ejemplo](#)

Muestra cómo compilar y ejecutar tanto el servidor del motor de cálculo como su cliente.

Introducción a las Aplicaciones RMI

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente. Una aplicación servidor típica crea un montón de objetos remotos, hace accesibles unas referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Una aplicación cliente típica obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos. RMI proporciona el mecanismo por el que se comunican y se pasan información del cliente al servidor y viceversa. Cuando es una aplicación algunas veces nos referimos a ella como Aplicación de Objetos Distribuidos.

Las aplicaciones de objetos distribuidos necesitan:

Localizar Objetos Remotos

Las aplicaciones pueden utilizar uno de los dos mecanismos para obtener referencias a objetos remotos. Puede registrar sus objetos remotos con la facilidad de nombrado de RMI `rmiregistry`. O puede pasar y devolver referencias de objetos remotos como parte de su operación normal.

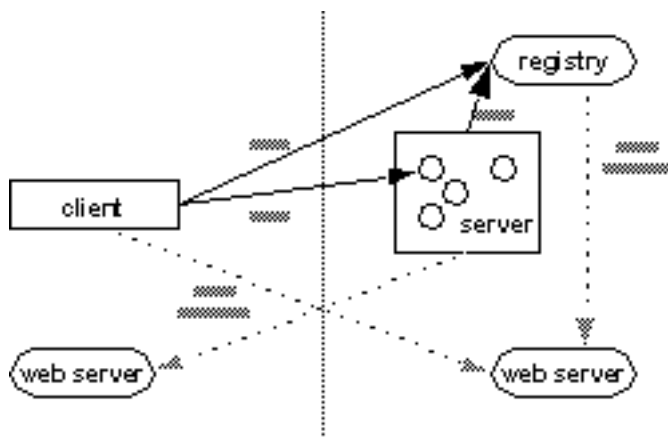
Comunicar con Objetos Remotos

Los detalles de la comunicación entre objetos remotos son manejados por el RMI; para el programador, la comunicación remota se parecerá a una llamada estándar a un método Java.

Cargar Bytecodes para objetos que son enviados.

Como RMI permite al llamador pasar objetos Java a objetos remotos, RMI proporciona el mecanismo necesario para cargar el código del objeto, así como la transmisión de sus datos.

La siguiente ilustración muestra una aplicación RMI distribuida que utiliza el registro para obtener referencias a objetos remotos. El servidor llama al registro para asociar un nombre con un objeto remoto. El cliente busca el objeto remoto por su nombre en el registro del servidor y luego llama a un método. Esta ilustración también muestra que el sistema RMI utiliza un servidor Web existente para cargar los bytecodes de la clase Java, desde el servidor al cliente y desde el cliente al servidor, para los objetos que necesita.



El sistema RMI utiliza un servidor Web para cargar los bytecodes de la clase Java,

desde el servidor al cliente y desde el cliente al servidor.

Ventajas de la Carga Dinámica de Código

Una de las principales y únicas características de RMI es la habilidad de descargar los bytecodes (o simplemente, código) de una clase de un objeto si la clase no está definida en la máquina virtual del receptor. Los tipos y comportamientos de un objeto, anteriormente sólo disponibles en una sola máquina virtual, ahora pueden ser transmitidos a otra máquina virtual, posiblemente remota. RMI pasa los objetos por su tipo verdadero, por eso el comportamiento de dichos objetos no cambia cuando son enviados a otra máquina virtual. Esto permite que los nuevos tipos sean introducidos en máquinas virtuales remotas, y así extender el comportamiento de una aplicación dinámicamente. El ejemplo del motor de cálculo de este capítulo utiliza las capacidades de RMI para introducir un nuevo comportamiento en un programa distribuido.

Interfaces, Objetos y Métodos Remotos

Una aplicación distribuida construida utilizando RMI de Java, al igual que otras aplicaciones Java, está compuesta por interfaces y clases. Los interfaces definen métodos, mientras que las clases implementan los métodos definidos en los interfaces y, quizás, también definen algunos métodos adicionales. En una aplicación distribuida, se asume que algunas implementaciones residen en diferentes máquinas virtuales. Los objetos que tienen métodos que pueden llamarse por distintas máquinas virtuales son los objetos remotos.

Un objeto se convierte en remoto implementando un interface remoto, que tenga estas características:

- Un interface remoto descende del interface `java.rmi.Remote`.
- Cada método del interface declara que lanza una `java.rmi.RemoteException` además de cualquier excepción específica de la aplicación.

El RMI trata a un objeto remoto de forma diferente a como lo hace con los objetos no-remotos cuando el objeto es pasado desde una máquina virtual a otra. En vez de hacer una copia de la implementación del objeto en la máquina virtual que lo recibe, RMI pasa un stub para un objeto remoto. El stub actúa como la representación local o proxy del objeto remoto y básicamente, para el llamador, es la referencia remota. El llamador invoca un método en el stub local que es responsable de llevar a cabo la llamada al objeto remoto.

Un stub para un objeto remoto implementa el mismo conjunto de interfaces remotos que el objeto remoto. Esto permite que el stub sea tipado a cualquiera de los interfaces que el objeto remoto implementa. Sin embargo, esto también significa que sólo aquellos métodos definidos en un interface remoto están disponibles para ser llamados en la máquina virtual que lo recibe.

Crear Aplicaciones Distribuidas utilizando RMI

Cuando se utiliza RMI para desarrollar una aplicación distribuida, debemos seguir estos pasos generales:

1. [Diseñar e implementar los componentes de nuestra aplicación distribuida.](#)
2. [Compilar los Fuentes y generar stubs.](#)
3. [Hacer las clases Accesibles a la Red.](#)
4. [Arrancar la Aplicación.](#)

Diseñar e implementar los componentes de nuestra aplicación distribuida.

Primero, decidimos la arquitectura de nuestra aplicación y determinamos qué componentes son objetos locales y cuales deberían ser accesibles remotamente. Este paso incluye:

- Definir los Interfaces Remotos. Un interface remoto especifica los métodos que pueden ser llamados remotamente por un cliente. Los clientes programan los interfaces remotos, no la implementación de las clases de dichos interfaces. Parte del diseño de dichos interfaces es la determinación de cualquier objeto local que sea utilizado como parámetro y los valores de retorno de esos métodos; si alguno de esos interfaces o clases no existen aún también tenemos que definirlos.
- Implementar los Objetos Remotos. Los objetos remotos deben implementar uno o varios interfaces remotos. La clase del objeto remoto podría incluir implementaciones de otros interfaces (locales o remotos) y otros métodos (que sólo estarán disponibles localmente). Si alguna clase local va a ser utilizada como parámetro o como valor de retorno de alguno de esos métodos, también debe ser implementada.
- Implementar los Clientes. Los clientes que utilizan objetos remotos pueden ser implementados después de haber definido los interfaces remotos, incluso después de que los objetos remotos hayan sido desplegados.

Compilar los Fuentes y Generar stubs.

Este es un proceso de dos pasos. En el primer paso, se utiliza el compilador `javac` para compilar los ficheros fuentes de Java, los cuales contienen las implementaciones de los interfaces remotos, las clases del servidor, y del cliente. En el segundo paso es utilizar el compilador `rmic` para crear los stubs de los objetos remotos. RMI utiliza una clase stub del objeto remoto como un proxy en el cliente para que los clientes puedan comunicarse con un objeto remoto particular.

Hacer accesibles las Clases en la Red.

En este paso, tenemos que hacer que todo - los ficheros de clases Java asociados con los interfaces remotos, los stubs, y otras clases que necesitemos descargar en

los clientes - sean accesibles a través de un servidor Web.

Arrancar la Aplicación.

Arrancar la aplicación incluye ejecutar el registro de objetos remotos de RMI, el servidor y el cliente.

El resto de este capítulo muestra cómo seguir estos pasos para crear un motor de cálculo.

Construir un Motor de Cálculo Genérico

Esta sección se enfoca a una sencilla pero potente aplicación distribuida llamada motor de cálculo. Este motor de cálculo es un objeto remoto en el servidor que toma tareas de clientes, las ejecuta, y devuelve los resultados. Las tareas se ejecutan en la máquina en la que se está ejecutando el servidor. Este tipo de aplicación distribuida podría permitir que un número de máquinas clientes utilizaran una máquina potente, o una que tuviera hardware especializado.

El aspecto novedoso del motor de cálculo es que las tareas que ejecuta no necesitan estar definidas cuando se escribe el motor de cálculo. Se pueden crear nuevas clases de tareas en cualquier momento y luego entregarlas al motor de cálculo para ejecutarlas. Todo lo que una tarea requiere es que su clase implemente un interface particular. Por eso una tarea puede ser enviada al motor de cálculo y ejecutada, incluso si la clase que define la tarea fue escrita mucho después de que el motor de cálculo fuera escrito y arrancado. El código necesita conseguir que una tarea sea descargada por el sistema RMI al motor de cálculo, y que éste ejecute la tarea utilizando los recursos de la máquina en la que está ejecutando el motor de cálculo.

La habilidad para realizar tareas arbitrarias esta permitida por la naturaleza dinámica de la plataforma Java, que se extiende a través de la red mediante RMI. El RMI carga dinámicamente el código de las tareas en la máquina virtual del motor de cálculo y ejecuta la tarea si tener un conocimiento anterior de la clase que implementa la tarea. Una aplicación como ésta que tiene la habilidad de descargar código dinámicamente recibe el nombre de "aplicación basada en comportamiento". Dichas aplicaciones normalmente requieren infraestructuras que permitan agentes. Con RMI, dichas aplicaciones son parte del mecanismo básico de programación distribuida de Java.

Escribir un Servidor RMI

El servidor del motor de cálculo acepta tareas de los clientes, las ejecuta, y devuelve los resultados. El servidor está compuesto por un interface y una clase. El interface proporciona la definición de los métodos que pueden ser llamados desde el cliente. Esencialmente, el interface define lo que el cliente ve del objeto remoto. La clase proporciona la implementación.

[Diseñar un Interface Remoto](#)

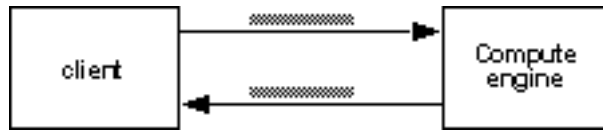
Esta página muestra cómo el interface Compute es el pegamento que conecta el cliente y el servidor. También aprenderemos sobre el API de RMI que soporta esta comunicación.

[Implementar un Interface Remoto](#)

En esta página exploraremos la clase que implementa el interface Compute, que implementa un objeto remoto. Esta clase también proporciona el resto del código que configura el programa servidor: un método main que crea un ejemplar del objeto remoto, lo registra con la facilidad de nombrado, y configura un controlador de seguridad.

Diseñar un Interface Remoto

En el corazón del motor de cálculo hay un protocolo que permite que se le puedan enviar trabajos, el motor de cálculo ejecuta esos trabajos, y los resultados son devueltos al cliente. Este protocolo está expresado en interfaces soportados por el motor de cálculo y por los objetos que le son enviados.



El protocolo del motor de cálculo en acción.

Cada uno de los interfaces contiene un sólo método. El interface del motor de cálculo `Compute`, permite que los trabajos sean enviados al motor, mientras que el interface `Task` define cómo el motor de cálculo ejecuta una tarea enviada.

El interface [compute.Compute](#) define la parte accesible remotamente - el propio motor de cálculo. Aquí está el interface remoto con su único método:

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    Object executeTask(Task t) throws RemoteException;
}
```

Al extender el interface `java.rmi.Remote`, este interface se marca a sí mismo como uno de aquellos métodos que pueden ser llamados desde cualquier máquina virtual. Cualquier objeto que implemente este interface se convierte en un objeto remoto.

Como miembro de un interface remoto, el método `executeTask` es un método remoto. Por lo tanto, el método debe ser definido como capaz de lanzar una `java.rmi.RemoteException`. Esta excepción es lanzada por el sistema RMI durante una llamada a un método remoto para indicar que ha fallado la comunicación o que ha ocurrido un error de protocolo. Una `RemoteException` es una excepción chequeada, por eso cualquier método Java que haga una llamada a un método remoto necesita manejar esta excepción, capturándola o declarándola en sus cláusula `throws`.

El segundo interface necesitado por el motor de cálculo define el tipo `Task`. Este tipo es utilizado como un argumento del método `executeTask` del interface `Compute`. El interface [compute.Task](#) define el interface entre el motor de cálculo y el trabajo que necesita hacer, proporcionando la forma de iniciar el trabajo:

```
package compute;

import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```

El interface Task define un sólo método, execute. Este método devuelve un Object, y no tiene parámetros ni lanza excepciones. Como este interface no extiende Remote, el método no necesita listar java.rmi.RemoteException en su clausula throws.

El valor de retorno de los métodos executeTask de Compute y execute de Task es declarado como del tipo Object. Esto significa que cualquiera tarea que quiera devolver un valor de uno de los tipos primitivos de Java (como un int o un float) necesita crear un ejemplar de la clase envolvente equivalente para ese tipo (como un Integer o un Float) y devolver ese objeto en su lugar.

Observamos que el interface Task extiende el interface java.io.Serializable. El RMI utiliza el mecanismo de serialización de objetos para transportar objetos entre máquinas virtuales. Implementar Serializable hace que la clase sea capaz de convertirse en un stream de bytes auto-descriptor que puede ser utilizado para reconstruir una copia exacta del objeto serializado cuando el objeto es leído desde el stream.

Se pueden ejecutar diferentes tipos de tareas en un objeto Compute siempre que sean implementaciones del tipo Task. Las clases que implementen este interface pueden contener cualquier dato necesario para el cálculo de la tarea, y cualquier otro método necesario para ese cálculo.

Así es cómo RMI hace posible este sencillo motor de cálculo. Como RMI puede asumir que los objetos Task están escritos en Java, las implementaciones de los objetos Task que anteriormente eran desconocidas para el motor de cálculo son descargadas por el RMI dentro de la máquina virtual del motor de cálculo cuando sea necesario. Esto permite a los clientes del motor de cálculo definir nuevos tipos de tareas para ser ejecutadas en el servidor sin necesitar que el código sea instalado explícitamente en dicha máquina. Además, como el método executeTask devuelve un java.lang.Object, cualquier tipo de objeto Java puede ser pasado como valor de retorno en una llamada remota.

El motor de cálculo, implementado por la clase ComputeEngine, implementa el interface Compute, permitiendo que diferentes tareas le sean enviadas mediante llamadas a su método executeTask. Estas tareas se ejecutan utilizando la implementación de task del método execute. El motor de cálculo devuelve los resultados a su llamador a través de su valor de retorno: un Object.

Implementar un Interface Remoto

Empecemos la tarea de implementar una clase para el motor de cálculo. En general, la implementación de la clase para un interface remoto debería al menos:

- Declarar los Interfaces remotos que están siendo implementados.
- Definir el constructor del objeto remoto.
- Proporcionar una implementación para cada método remoto de cada interface remoto.

El servidor necesita crear e instalar los objetos remotos. Este proceso de configuración puede ser encapsulado en un método main en la propia clase de implementación del objeto remoto, o puede ser incluido completamente en otra clase. El proceso de configuración debería:

- Crear e instalar un controlador de seguridad.
- Crear uno o más ejemplares del objeto remoto.
- Registrar al menos uno de los objetos remotos con el registro de objetos remotos de RMI (a algún otro servicio de nombrado que utilice JNDI).

Abajo podemos ver la implementación completa del motor de cálculo. La clase [engine.ComputeEngine](#) implementa el interface remoto Compute y también incluye el método main para configurar el motor de cálculo:

```
package engine;

import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class ComputeEngine extends UnicastRemoteObject
    implements Compute
{
    public ComputeEngine() throws RemoteException {
        super();
    }
    public Object executeTask(Task t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//localhost/Compute";
        try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Ahora echaremos una mirada más cercana a cada uno de los componentes de la implementación del motor de cálculo.

Declarar los Interfaces Remotos que están siendo Implementados

La clase que implementa el motor de cálculo se declara como:

```
public class ComputeEngine extends UnicastRemoteObject
    implements Compute
```

Esta declaración indica que la clase implementa el interface remoto Compute (y, por lo tanto, define un objeto remoto) y extiende la clase `java.rmi.server.UnicastRemoteObject`.

`UnicastRemoteObject` es una clase de conveniencia, definida en el API público del RMI, que puede ser utilizada como superclase para la implementación de objetos remotos. La superclase `UnicastRemoteObject` suministra implementación para un gran número de métodos de `java.lang.Object` (`equals`, `hashCode`, `toString`) para que estén definidos apropiadamente para objetos remotos. `UnicastRemoteObject` también incluye constructores y métodos estáticos utilizados para exportar un objeto remoto, es decir, hacer que el objeto remoto pueda recibir llamadas de los clientes.

Una implementación de objeto remoto no tiene porque extender `UnicastRemoteObject`, y ninguna implementación que lo haga debe suministrar las implementaciones apropiadas de los métodos de `java.lang.Object`. Además, una implementación de un objeto remoto debe hacer una llamada explícita a uno de los métodos `exportObject` de `UnicastRemoteObject` para que el entorno RMI se de cuenta del objeto remoto para que éste pueda aceptar llamadas.

Al extender `UnicastRemoteObject`, la `ComputeEngine` puede ser utilizada pra crear un sólo objeto remoto que soporte comunicación remota (punto a punto) y que utilice el transporte de comunicación basado en sockets que tiene por defecto el RMI.

Si elegimos extender un objeto remoto de otra clase distinta de `UnicastRemoteObject`, o, alternativamente, los extendemos de la nueva clase `java.rmi.activation.Activatable` del JDK 1.2 (utilizada pra construir objetos remotos que puedan ser ejecutados sobre demanda), necesitamos exportar explícitamente el objeto remoto llamando a uno de los métodos `UnicastRemoteObject.exportObject` o `Activatable.exportObject` desde el constructor de nuestra clase (o cualquier otro método de inicialización, cuando sea apropiado).

El ejemplo del motor de cálculo define un objeto remoto que implementa un sólo interface remoto y ningún otro interface. La clase `ComputeEngine` también contiene algunos métodos que sólo pueden ser llamados localmente. El primero de ellos es un constructor para objetos `ComputeEngine`; el segundo es un método `main` que es utilizado para crear un objeto `ComputeEngine` y ponerlo a disposición de los clientes.

Definir el Constructor

La clase `ComputeEngine` tiene un único constructor que no toma argumentos:

```
public ComputeEngine() throws RemoteException {
    super();
}
```

Este constructor sólo llama al constructor de su superclase, que es el constructor sin argumentos de la clase `UnicastRemoteObject`. Aunque el constructor de la superclase obtiene la llamada incluso si la omitimos en el constructor de `ComputeEngine`, la hemos incluido por claridad.

Durante la construcción, un objeto `UnicastRemoteObject` es exportado, lo que significa que está disponible para aceptar peticiones de entrada al escuchar las llamadas de los clientes en un puerto anónimo.

Nota: En el JDK 1.2, podríamos indicar el puerto específico que un objeto remoto utiliza para aceptar peticiones.

El constructor sin argumentos de la superclase, `UnicastRemoteObject`, declara la excepción `RemoteException` en su cláusula `throws`, por eso el constructor de `ComputeEngine` también debe declarar que lanza una `RemoteException`. Esta excepción puede ocurrir durante la construcción si falla el intento de exportar el objeto (debido a que, por ejemplo, no están disponibles los recursos de comunicación o a que la clase stub apropiada no se encuentra).

Proporcionar una Implementación para cada Método Remoto

La clase para un objeto remoto proporciona implementaciones para todos los métodos remotos especificados en los interfaces remotos. El interface `Compute` contiene un sólo método remoto, `executeTask`, que se implementa de esta forma:

```
public Object executeTask(Task t) {  
    return t.execute();  
}
```

Este método implementa el protocolo entre el `ComputeEngine` y sus clientes. Los clientes proporcionan al `ComputeEngine` un objeto `Task`, que tiene una implementación del método `execute` de `task`. El `ComputeEngine` ejecuta la tarea y devuelve el resultado del método directamente a su llamador.

El método `executeTask` no necesita saber nada más sobre el resultado del método `execute` sólo que es un `Object`. El llamador presumiblemente sabe algo más sobre el tipo preciso del `Object` devuelto y puede tipar el resultado al tipo apropiado.

Pasar Objetos en RMI

Los argumentos y los tipos de retorno de los métodos remotos pueden ser de casi cualquier tipo Java, incluyendo objetos locales, objetos remotos y tipos primitivos. Más precisamente, una entidad de cualquier tipo Java puede ser pasada como un argumento o devuelta por un método remoto siempre que la entidad sea un ejemplar de un tipo que sea:

- Un tipo primitivo de Java,
- un objeto remoto, o
- un objeto serializable lo que significa que implementa el interface `java.io.Serializable`

Unos pocos tipos de objetos no cumplen con estos criterios y por lo tanto no pueden ser pasados ni devueltos por un método remoto. La mayoría de estos objetos (como un descriptor de fichero) encapsulan información que sólo tiene sentido en un espacio de dirección única. Muchas clase del corazón Java, incluso algunas de `java.lang` y `java.util`, implementan el interface `Serializable`.

Estas son las reglas que gobiernan el paso y retorno de valores:

- Los objetos remotos se pasan esencialmente por referencia. Una referencia a un objeto remoto es realmente un stub, que es un proxy del lado del cliente que implementa el conjunto completo de interfaces remotos que implementa el objeto remoto.
- Los objetos locales son pasados por copia utilizando el mecanismo de serialización de objetos de Java. Por defecto, todos los campos se copian, excepto aquellos que están marcados como `static` o `transient`. El comportamiento de la serialización por defecto puede ser sobrecargado en una básica clase-por-clase.

Pasar un objeto por referencia (como se hace con los objetos remotos) significa que cualquier cambio hecho en el estado del objeto por el método remoto es reflejado en el objeto remoto original. Cuando se pasa un objeto remoto, sólo aquellos interfaces que son interfaces remotos están disponibles para el receptor, cualquier otro método definido en la implementación de la clase o definido en un interface no remoto no estará disponible para el receptor.

Por ejemplo, si pasaríamos por referencia un ejemplar de la clase `ComputeEngine`, el receptor tendría acceso sólo al método `executeTask`. El receptor no vería ni el constructor `ComputeEngine`

ni su método `main` ni cualquier otro método de `java.lang.Object`.

En las llamadas a método remoto, los objetos -parámetros, valores de retorno y excepciones - que no son objetos remotos son pasados por valor. Esto significa que se crea una copia del objeto en la máquina virtual del receptor. Cualquier cambio en el estado del objeto en el receptor será reflejado sólo en la copia del receptor, no en el ejemplar original.

El método `main()` del Servidor

El método más complicado de la implementación de `ComputeEngine` es el método `main`. Este método es utilizado para arrancar el `ComputeEngine`, y, por lo tanto, necesita hacer la inicialización necesaria para preparar el servidor para aceptar llamadas de los clientes. Este método no es un método remoto, lo que significa que no puede ser llamado desde otra máquina virtual que no sea la suya. Como el método `main` se declara `static`, no está asociado con ningún objeto, sino con la clase `ComputeEngine`.

Crear e Instalar un Controlador de Seguridad

Lo primero que hace el método `main` es crear e instalar un controlador de seguridad. Éste protege los accesos a los recursos del sistema por parte de código no firmado que se ejecute dentro de la máquina virtual. El controlador de seguridad determina si el código descargado tiene acceso al sistema de ficheros local o puede realizar cualquier otra operación privilegiada.

Todos los programas que utilicen RMI deben instalar un controlador de seguridad o el RMI no descargará las clases (las que no se encuentren en el path local) para los objetos que se reciban como parámetros. Estas restricciones aseguran que las operaciones realizadas por el código descargado pasarán a través de unas pruebas de seguridad.

El `ComputeEngine` utiliza un ejemplo de controlador de seguridad suministrado como parte del RMI, el `RMI SecurityManager`. Este controlador de seguridad fuerza una política de seguridad similar al controlador de seguridad típico de los applets (es decir, es muy conservador con los accesos que permite). Una aplicación RMI podría definir y utilizar otra clase `SecurityManager` que diera un acceso más liberal a los recursos del sistema, o, en el JDK 1.2, utilizar un fichero de vigilancia que ofrezca más permisos.

Aquí temos el código que crea e instala el controlador de seguridad:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
```

Poner el Objeto Remoto a Disposición de los Clientes

Luego, el método `main` crea un ejemplar de `ComputeEngine`. Esto se hace con la sentencia:

```
Compute engine = new ComputeEngine();
```

Como se mencionó anteriormente, este constructor llama al constructor de su superclase `UnicastRemoteObject`, que exporta el objeto recién creado al sistema RMI. Una vez completada la exportación, el objeto remoto `ComputeEngine` está listo para aceptar llamadas de los clientes en un puerto anónimo (elegido por el RMI o por el sistema operativo). Observa que el tipo de la variable `engine` es `Compute`, y no `ComputeEngine`. Esta declaración enfatiza que el interface disponible para los clientes es el interface `Compute` y sus métodos, no la clase `ComputeEngine` y sus métodos.

Antes de que un llamador pueda invocar un método de un objeto remoto, debe obtener una referencia al objeto remoto. Esto puede hacerse de la misma forma que en que se obtiene cualquier otra referencia en un programa Java, que es obteniéndolo como parte del valor de retorno de un

método o como parte de una estructura de datos que contenga dicha referencia.

El sistema proporciona un objeto remoto particular, el registro RMI, para encontrar referencias a objetos remotos. El registro RMI es un sencillo servicio de nombrado para objetos remotos que permite a los clientes remotos obtener una referencia a un objeto remoto por su nombre. El registro se utiliza típicamente para localizar el primer objeto remoto que un cliente RMI necesita utilizar. Este primer objeto remoto, luego proporciona soporte para encontrar otros objetos.

El interfaz `java.rmi.Naming` es utilizado como un API final para la entrega (o registrado) y búsqueda de objetos remotos en el registro. Una vez registrado un objeto remoto en el registro RMI en el host local, los llamadores de cualquier host pueden buscar el objeto remoto por el nombre, obtener su referencia, y luego llamar a los métodos del objeto. El registro podría ser compartido por todos los servidores ejecutándose en un host, o un proceso servidor individual podría crear y utilizar su propio registro si así lo desea.

La clase `ComputeEngine` crea un nombre para el objeto con la sentencia:

```
String name = "//localhost  
/Compute";
```

Este nombre incluye el nombre del host `localhost`, en el que se están ejecutando el registro y el objeto remoto, y un nombre `Compute`, que identifica el objeto remoto en el registro. Luego está el código necesario para añadir el nombre al registro RMI que se está ejecutando en el servidor. Esto se hace después (dentro del bloque `try` con la sentencia:

```
Naming.rebind(name, engine);
```

Al llamar al método `rebind` se hace una llamada remota al registro RMI del host local. Esta llamada puede provocar que se genera una `RemoteException`, por eso tenemos que manejar la excepción. La clase `ComputeEngine` maneja la excepción dentro de los bloques `try/catch`. Si la excepción no fuese manejada de esta manera, tendríamos que añadir `RemoteException` a la cláusula `throws` (ahora inexistente) del método `main`.

Observemos lo siguiente sobre los argumentos de la llamada a `Naming.rebind`:

- El primer parámetro es un `java.lang.String` formateado como URL representando la localización y el nombre del objeto remoto:
 - Podríamos necesitar cambiar el valor de `localhost` por el nombre o dirección IP de nuestro servidor. Si se omite el `Host` en la URL, el host por defecto es el host local. Tampoco necesitamos especificar el protocolo, Por ejemplo, está permitido suministrar `"Compute"` como el nombre en la llamada a `Naming.rebind`.
 - Opcionalmente se puede suministrar un número de puerto en la URL, por ejemplo el nombre `"//host:1234/objectname"` es legal. Si se omite el puerto, por defecto se toma el 1099. Debemos especificar el puerto si un servidor crea un registro en otro puerto que no sea el 1099. El puerto por defecto es útil porque proporciona un lugar bien conocido para buscar los objetos remotos que ofrecen servicios en un host particular.
- El sistema RMI sustituye una referencia al stub por la referencia real al objeto especificado en el argumento. La implementación de objetos remotos como ejemplares de `ComputeEngine` nunca abandonan la máquina virtual en que se crearon, por eso, cuando un cliente realiza un búsqueda en el registro de objetos remotos del servidor, se le devuelve una referencia al stub. Como se explicó anteriormente, los objetos remotos en dichos casos se pasan por referencia, no por valor.
- Observe que por razones de seguridad, una aplicación puede entregar o eliminar referencias a objetos remotos sólo en un registro que se ejecute en el mismo host. Esta restricción evita que un cliente remoto elimine o sobreescriba cualquier entrada en el registro del servidor.

Una vez que el servidor se ha registrado en el registro RMI local, imprime un mensaje indicando que

está listo para empezar a manejar llamadas, y sale del método main. No es necesario tener un thread esperando para mantener vivo el servidor. Siempre que haya una referencia al objeto ComputeEngine en algún lugar de la máquina virtual (local o remota) el objeto ComputeEngine no será eliminado. Como el programa entrega una referencia de ComputeEngine en el registro, éste es alcanzable por un cliente remoto (¡el propio registro!). El sistema RMI tiene cuidado de mantener vivo el proceso ComputeEngine. El ComputeEngine está disponible para aceptar llamadas y no será reclamado hasta que:

- su nombre sea eliminado del registro, y
- ningún cliente remoto mantenga una referencia al objeto ComputeEngine.

La pieza final de código del método ComputeEngine.main maneja cualquier excepción que pudiera producirse. La única excepción que podría ser lanzada en el código es RemoteException, que podría ser lanzada por el constructor de la clase ComputeEngine o por la llamada al registro para entregar el nombre del objeto "Compute". En cualquier caso, el programa no puede hacer nada más que salir e imprimir un mensaje de error. En algunas aplicaciones distribuidas, es posible recuperar un fallo al hacer una llamada remota. Por ejemplo, la aplicación podría elegir otro servidor y continuar con la operación.

Crear un Programa Cliente

El motor de cálculo es un bonito y sencillo programa - ejecuta las tareas que le son enviadas. Los clientes del motor de cálculo son más complejos. Un cliente necesita llamar al motor de cálculo, pero también tiene que definir la tarea que éste va a realizar.

Nuestro ejemplo está compuesto por dos clases separadas. la primera clase `ComputePi`, busca y llama a un objeto `Compute`. La segunda clase `Pi`, implementa el interface `Task` y define el trabajo que va a hacer el motor de cálculo. El trabajo de la clase `Pi` es calcular el valor del número pi, con algún número de posiciones decimales.

Como recordaremos, el interface no-remoto [Task](#) se define de esta forma:

```
package compute;
public interface Task extends java.io.Serializable {
    Object execute();
}
```

El interface `Task` extiende `java.io.Serializable` por lo que cualquier objeto que lo implemente puede ser serializado por el sistema RMI y enviado a una máquina virtual remota como parte de una llamada a un método remoto. Podríamos haber elegido hacer que la implementación de nuestra clase implementara los interfaces `Task` y `Serializable`, y hubiera tenido el mismo efecto. Sin embargo, el único proposito del interface `Task` es permitir que las implementaciones de este interface sean pasadas a objetos `Compute`, por eso, una clase que implemente el interface `Task` no tiene sentido que también implemente el interface `Serializable`. Dado esto, hemos asociado explícitamente los dos interfaces en el tipo `system`, asegurando que todos los objetos `Task` sean serializables.

El código que llama a los métodos del objeto `Compute` debe obtener una referencia a ese objeto, crear un objeto `Task`, y luego pedir que se ejecute la tarea. Más adelante veremos la definición de la tarea `Pi`. Un objeto `Pi` se construye con un sólo argumento, la precisión deseada en el resultado. El resultado de la ejecución de la tarea es un `java.math.BigDecimal` que representa el número pi calculado con la precisión especificada.

La clase cliente [ComputePi](#):

```
package client;

import java.rmi.*;
import java.math.*;
import compute.*;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
```

```

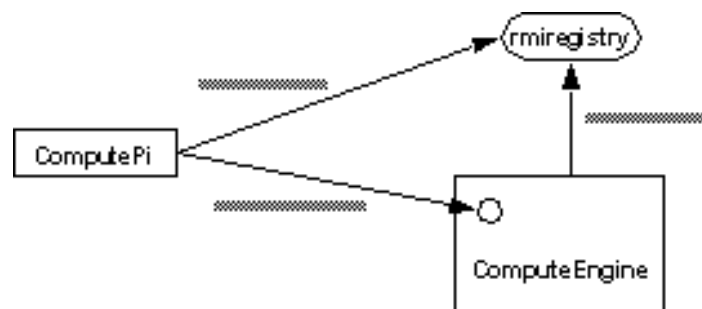
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        String name = "/" + args[0] + "/Compute";
        Compute comp = (Compute) Naming.lookup(name);
        Pi task = new Pi(Integer.parseInt(args[1]));
        BigDecimal pi = (BigDecimal) (comp.executeTask(task));
        System.out.println(pi);
    } catch (Exception e) {
        System.err.println("ComputePi exception: " + e.getMessage());
        e.printStackTrace();
    }
}
}
}

```

Al igual que el servidor ComputeEngine, el cliente empieza instalando un controlador de seguridad. Esto es necesario porque RMI podría descargar código en el cliente. En este ejemplo, el stub ComputeEngine es descargado al cliente. Siempre que el RMI descargue código, debe presentarse un controlador de seguridad. Al igual que el servidor, el cliente utiliza el controlador de seguridad proporcionado por el sistema RMI para este propósito.

Después de llamar al controlador de seguridad, el cliente construye un nombre utilizado para buscar un objeto remoto Compute. El valor del primer argumento de la línea de comandos args[0], es el nombre del host remoto, en el que se están ejecutando los objetos Compute. Usando el método Naming.lookup, el cliente busca el objeto remoto por su nombre en el registro del host remoto. Cuando se hace la búsqueda del nombre, el código crea una URL que especifica el host donde se está ejecutando el servidor. El nombre pasado en la llamada a Naming.lookup tiene la misma sintaxis URL que el nombre pasado a la llamada Naming.rebind que explicamos en páginas anteriores.

Luego, el cliente crea un objeto Pi pasando al constructor de Pi el segundo argumento de la línea de comandos, args[1], que indica el número de decimales utilizados en el cálculo. Finalmente, el cliente llama al método executeTask del objeto remoto Compute. El objeto pasado en la llamada a executeTask devuelve un objeto del tipo java.math.BigDecimal, por eso el programa fuerza el resultado a ese tipo y almacena en resultado en la variable result. Finalmente el programa imprime el resultado.



Flujo de mensajes entre el cliente ComputePi, el rmiregistry, y el ComputeEngine.

Finalmente, echemos un vistazo a la clase Pi. Esta clase implementa el interface Task y calcula el valor del número pi con un número de decimales especificado. Desde el punto de vista de este ejemplo, el algoritmo real no es importante (excepto, por supuesto, para la fiabilidad del cálculo). Todo lo importante es que el cálculo consume numéricamene muchos recursos (y por eso es el tipo que cosa que queríamos hacer en un servidor potente).

Aquí tenemos el código de la clase [Pi](#), que implementa Task:

```
package client;
import compute.*;
import java.math.*;

public class Pi implements Task {

    /** constantes utilizadas en el cálculo de pi*/
    private static final BigDecimal ZERO =
        BigDecimal.valueOf(0);
    private static final BigDecimal ONE =
        BigDecimal.valueOf(1);
    private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

    /** modo de redondeo utilizado durante el cálculo*/
    private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

    /** número de dígitos tras el punto decimal*/
    private int digits;

    /**
     * Construye una tarea para calcular el núemro pi
     * con la precisión especificada.
     */
    public Pi(int digits) {
        this.digits = digits;
    }

    /**
     * Calcula pi.
     */
    public Object execute() {
        return computePi(digits);
    }

    /**
     * Calcula el valor de Pi con el número de decimales especificados.
     */
}
```

```

* El valor se calcula utilizando la fórmula de Machin:
*
*      
$$\pi/4 = 4 \cdot \arctan(1/5) - \arctan(1/239)$$

*
* y una poderosa serie de expansiones de  $\arctan(x)$ 
* para una precisión suficiente.
*/
public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi =
arctan1_5.multiply(FOUR).subtract(arctan1_239).multiply(FOUR);
    return pi.setScale(digits,
                        BigDecimal.ROUND_HALF_UP);
}

/**
 * Calcula el valor, en radianes, de la arcotangente de la
 * inversa del entero suministrado para el número de decimales.
 * El valor se calcula utilizando la poderosa serie de
 * expansiones de arcotangente:
 *
 * 
$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \dots$$

 */
public static BigDecimal arctan(int inverseX,
                                int scale)
{
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 =
        BigDecimal.valueOf(inverseX * inverseX);

    numer = ONE.divide(invX, scale, roundingMode);

    result = numer;
    int i = 1;
    do {
        numer =
            numer.divide(invX2, scale, roundingMode);
        int denom = 2 * i + 1;
        term =
            numer.divide(BigDecimal.valueOf(denom),
                        scale, roundingMode);
        if ((i % 2) != 0) {
            result = result.subtract(term);
        }
    } while (true);
}

```

```
        } else {
            result = result.add(term);
        }
        i++;
    } while (term.compareTo(ZERO) != 0);
    return result;
}
}
```

La característica más interesante de este ejemplo es que el objeto Compute no necesita una definición de la clase Pi hasta que se le pasa un objeto Pi como un argumento del método `executeTask`. Hasta este punto, el código de la clase se ha cargado por el RMI dentro de la máquina virtual del objeto Compute, se ha llamado al método `execute`, y se ha ejecutado el código de la tarea. El Object resultante (que en el caso de la tarea Pi es realmente un objeto `java.math.BigDecimal`) es enviado de vuelta al cliente, donde se utiliza para imprimir el resultado.

El hecho de que el objeto Task suministrado calcule el valor de Pi es irrelevante para el objeto ComputeEngine. Por ejemplo, también podríamos implementar una tarea que generara un número primo aleatorio utilizando un algoritmo probabilístico. (Esto también consume muchos recursos y por tanto es un candidato para ser enviado al ComputeEngine). Este código también podría ser descargado cuando el objeto Task fuera pasado al objeto Compute. Todo lo que el objeto Compute sabe es que cada objeto que recibe implementa el método `execute`, no sabe (y tampoco le interesa) qué hace la implementación.

Compilar y Ejecutar el Ejemplo

Ahora que hemos visto el código que crea el cliente y el servidor para el ejemplo del motor de cálculo, vamos a compilarlo y ejecutarlo.

[Compilar los Programas de Ejemplo](#)

En esta página aprenderemos cómo compilar los programas cliente y servidor que componen el ejemplo del motor de cálculo.

[Ejecutar los Programas de Ejemplo](#)

Finalmente, ejecutaremos los programas del servidor y del cliente y consecuentemente, obtendremos el valor del número pi.

Compilar el Programa de Ejemplo

En un escenario del mundo real donde se desarrollara un servicio como el del motor de cálculo, un desarrollador querría crear un fichero JAR que contenga los interfaces `Compute` y `Task` para que los implementan las clases servidor y para que los utilicen los programas clientes.

Luego, un desarrollador (quizás el mismo que creo el fichero JAR con los interfaces) escribiría una implementación del interface `Compute` y desarrollaría ese servicio en una máquina disponible para los clientes.

Los desarrolladores de los programas clientes pueden utilizar los interfaces `Compute` y `Task` (contenidos en el fichero JAR) e independientemente desarrollar una tarea y un programa cliente que utilice un servicio `Compute`.

En esta página, aprenderemos cómo crear un fichero JAR, las clases del servidor, y las clases del cliente. Veremos como la clase `Pi` será descargada al servidor durante la ejecución. También veremos como el stub remoto `ComputeEngine` será descargado desde el servidor hasta el cliente durante la ejecución.

El ejemplo separa los interfaces, la implementación de los objetos remotos y el código del cliente en tres paquetes diferentes:

- `compute` (Los interfaces [Compute](#) y [Task](#))
- `engine` (Implementación de la clase, el interface y el stub de [ComputeEngine](#))
- `client` (la implementación del cliente [ComputePi](#) y de la tarea [Pi](#))

Primero construiremos el fichero JAR para proporcionar los interfaces del servidor y del cliente a los desarrolladores.

Construir un Fichero JAR con las Clases de Interfaces

Primero necesitamos compilar los ficheros fuente de los interfaces del paquete `compute` que construir un fichero JAR que contenga los ficheros `class`. Supongamos que el usuario `waldo` ha escrito estos interfaces particulares y ha situado los ficheros fuente en `c:\home\waldo\src\compute` (en UNIX sería, `/home/waldo/src/compute`). Con estos paths podemos utilizar los siguientes comandos para compilar los interfaces y crear el fichero JAR

Detalles específicos de la Plataforma: Construir un Fichero JAR

Windows:

```
cd c:\home\waldo\src
javac compute\Compute.java
javac compute\Task.java
jar cvf compute.jar compute\*.class
```

UNIX:

```
cd /home/waldo/src
javac compute/Compute.java
```

```
javac compute/Task.java
jar cvf compute.jar compute/*.class
```

El comando jar muestra la siguiente salida (debido a la opción -v):

```
added manifest
adding: compute/Compute.class (in=281) (out=196)
      (deflated 30%)
adding: compute/Task.class (in=200) (out=164)
      (deflated 18%)
```

Ahora podemos distribuir el fichero compute.jar a los desarrolladores de las aplicaciones del cliente y del servidor para que puedan hacer uso de los interfaces.

En general, cuando construimos las clases del servidor o del cliente con los compiladores javac y rmic, necesitaremos especificar donde deberían residir los ficheros de clase resultantes para que sean accesibles a la red. En este ejemplo, esta localización es, para Unix, /home/user/public_html/classes porque algunos servidores web permiten el acceso a public_html mediante una URL HTTP construida como http://host/~user/. Si nuestro servidor web no soporta esta convención, podríamos utilizar un fichero URL en su lugar. El fichero de URL toma la forma file:/home/user/public_html/classes/ en UNIX, o file:/c:\home\user\public_html\classes/ en Windows. También se puede seleccionar otro tipo de URL apropiado.

La accesibilidad en la red de los ficheros de clases permite al sistema RMI descargar código cuando sea necesario. En vez de definir su propio protocolo para descargar código, RMI utiliza un protocolo URL soportado por Java (por ejemplo, HTTP) para descargar el código. Observa que un servidor web completo y poderoso no necesita realizar esta descarga de fichero class. De hecho, un sencillo servidor HTTP proporciona toda la funcionalidad necesaria para hacer que las clases estén disponibles para su descarga en RMI mediante HTTP, puedes encontrar uno en:

```
ftp://java.sun.com/pub/jdk1.1/rmi/class-server.zip
```

Construir las Clases del Servidor

El paquete engine sólo contiene la implementación de la clase del lado del servidor, ComputeEngine, la implementación del objeto remoto del interface Compute. Como ComputeEngine es una implementación de un interface remoto, necesitamos generar un stub para el objeto remoto para que los clientes puedan contactar con él.

Digamos que, ana, la desarrolladora de la clase ComputeEngine, ha situado ComputeEngine.java en el directorio c:\home\ana\src\engine, y ha colocado el fichero class para que lo usen los clientes en un subdirectorio de su directorio public_html, c:\home\ana\public_html\classes (en UNIX podría ser /home/ana/public_html/classes, accesible mediante algún servidor web como http://host/~ana/classes/).

Asumamos que el fichero compute.jar esta localizado en el directorio c:\home\ana\public_html\classes. Para compilar la clase ComputeEngine, nuestro path de clases debe incluir el fichero compute.jar y el propio directorio fuente.

Una nota sobre el path de clases: Normalmente, recomendamos seleccionar el path de clases en la línea de comandos utilizando la opción `-classpath`. Sin embargo, por varias razones, este ejemplo utiliza la variable de entorno `CLASSPATH` (porque tanto `javac` como `rmic` necesitan un path de clases y la opción `-classpath` se trata de forma diferente en el JDK 1.1 y el JDK 1.2). Recomendamos que no selecciones el `CLASSPATH` en un fichero de login o de arranque y que los desactives después de haber terminado con este ejemplo.

Para más información sobre `CLASSPATH` puedes visitar <http://java.sun.com/products/jdk/1.2/docs/install.html>

Aquí podemos ver cómo seleccionar la variable de entorno `CLASSPATH`:

Detalles Específicos de la Plataforma: Seleccionar el `CLASSPATH`

Windows:

```
set CLASSPATH=c:\home\ana\src;c:\home\ana\public_html\classes\compute.jar
```

Unix:

```
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

Ahora compilamos el fichero fuente `ComputeEngine.java` y generamos un stub para la clase `ComputeEngine` y coloca el stub accesible a la red. Para crear el stub (y opcionalmente los ficheros esqueleto) ejecutamos el compilador `rmic` sobre los nombres totalmente cualificados de las clases de implementación de los objetos remotos que deberían encontrarse en el path de clases. El comando `rmic` toma uno o más nombres de clase como entrada y produce, como salida, ficheros de clases con la forma `ClassName_Stub.class` (y opcionalmente `ClassName_Skel.class`). El fichero esqueleto no será generado si llamamos a `rmic` con la opción `-v1.2`. Esta opción sólo debería utilizarse si todos nuestros clientes van a utilizar el JDK 1.2 o posterior.

Detalles Específicos de la Plataforma: Compilar el Motor de Cálculo y sus Stubs

Windows:

```
cd c:\home\ana\src
javac engine\ComputeEngine.java
rmic -d . engine.ComputeEngine
mkdir c:\home\ana\public_html\classes\engine
cp engine\ComputeEngine_*.class
   c:\home\ana\public_html\classes\engine
```

Unix:

```
cd /home/ana/src
javac engine/ComputeEngine.java
rmic -d . engine.ComputeEngine
mkdir /home/ana/public_html/classes/engine
```

```
cp engine/ComputeEngine_*.class  
/home/ana/public_html/classes/engine
```

La opción -d le dice al compilador rmic que sitúe los ficheros de clases generados, ComputeEngine_Stub y ComputeEngine_Skel, en el directorio c:\home\ana\src\engine. También necesitamos poner estos ficheros accesibles en la red, por eso debemos copiarlos en el área public_html\classes.

Como el stub de ComputeEngine implementa el interface Compute, que referencia al interface Task, también necesitamos poner estas clases disponibles en la red. Por eso, el paso final es desempaquetar el fichero compute.jar en el directorio c:\home\ana\public_html\classes para hacer que los interfaces Compute y Task estén disponibles para su descarga.

Detalles Específicos de la Plataforma: Desempaquetar el Fichero JAR

Windows:

```
cd c:\home\ana\public_html\classes  
jar xvf compute.jar
```

Unix:

```
cd /home/ana/public_html/classes  
jar xvf compute.jar
```

El comando jar muestra esta salida:

```
created: META-INF/  
extracted: META-INF/MANIFEST.MF  
extracted: compute/Compute.class  
extracted: compute/Task.class
```

Construir las clases del Cliente

Asumamos que el usuario jones ha creado el código del cliente en el directorio c:\home\jones\src\client y colocará la clase Pi (para que sea descargada por el motor de cálculo) en el directorio accesible a la red c:\home\jones\public_html\classes (también disponible mediante algunos servidores como http://host/~jones/classes/). Las dos clases del lado del cliente están contenidas en los ficheros Pi.java y ComputePi.java en el subdirectorio client.

Para construir el código del cliente, necesitamos el fichero compute.jar que contiene los interfaces Compute y Task que utiliza el cliente. Digamos que el fichero compute.jar está situado en c:\home\jones\public_html\classes. Las clases del cliente se pueden construir así:

Detalles Específicos de la Plataforma: Compilar el Cliente

Windows:


```
set CLASSPATH=c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
cd c:\home\jones\src
javac client\ComputePi.java
javac -d c:\home\jones\public_html\classes client\Pi.java
```

UNIX:

```
setenv CLASSPATH /home/jones/src:/home/jones/public_html/classes/compute.jar
cd /home/jones/src
javac client/ComputePi.java
javac -d /home/jones/public_html/classes client/Pi.java
```

Sólo necesitamos situar la clase Pi en el directorio public_html\classes\client (el directorio client lo crea el javac si no existe). Esto es así por esta clase es la única que necesita ser desacargada por la máquina virtual del motor de cálculo.

Ahora podemos [ejecutar](#) el servidor y luego el cliente.

Ozito

Ejecutar los Programas de Ejemplo

Una Nota sobre la Seguridad

El modelo de seguridad del JDK 1.2 es más sofisticado que el modelo utilizado en el JDK 1.1. Contiene ampliaciones para seguridad de grano fino y requiere código que permita los permisos específicos para realizar ciertas operaciones.

En el JDK 1.1, todo el código que haya en el path de clases se considera firmado y puede realizar cualquier operación, el código descargado está gobernado por las reglas del controlador de seguridad instalado. Si ejecutamos este ejemplo en el JDK 1.2 necesitaremos especificar un fichero de policía cuando ejecutemos el servidor y el cliente. Aquí tenemos un [fichero de policía general](#) que permite al código descargado desde cualquier codebase, hacer dos cosas:

- conectar o aceptar conexiones en puertos no privilegiados (puertos por encima del 1024) de cualquier host, y
- conectar con el puerto 80 (el puerto HTTP).

```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
};
```

Si hacemos nuestro código disponible mediante URLs HTTP, deberíamos ejecutar el fichero de policía anterior cuando ejecutemos este ejemplo. Sin embargo, si utilizará un fichero de URLs en su lugar, podemos utilizar el [fichero de policía](#) siguiente. Observa que en entornos windows, la barra invertida necesita ser representada con dos barras invertidas en el fichero de policía:

```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.io.FilePermission  
        "c:\\home\\ana\\public_html\\classes\\-", "read";  
    permission java.io.FilePermission  
        "c:\\home\\jones\\public_html\\classes\\-", "read";  
};
```

Este ejemplo asume que el fichero de policía se llama java.policy y contiene los permisos apropiados. Si ejecutamos este ejemplo en el JDK 1.1, no necesitamos un fichero de policía ya que el RMI SecurityManager proporciona toda la protección que necesitamos.

Arrancar el Servidor

Antes de arrancar el motor de cálculo, necesitamos arrancar el registro de RMI con el comando rmiregistry. Como explicamos en páginas anteriores el registro RMI es una facilidad de nombrado que permite a los clientes obtener una referencia a un objeto remoto.

Observa que antes de arrancar el rmiregistry, debemos asegurarnos de que el shell o

ventana en la que ejecutaremos rmiregistry no tiene la variable de entorno CLASSPATH, o si la tiene ésta no incluye el path a ninguna clase, incluyendo los stubs de nuestras clases de implementación de los objetos remotos, que querramos descargar a los clientes de nuestros objetos remotos.

Si arrancamos el rmiregistry y éste puede encontrar nuestras clases stub en el CLASSPATH, no recordará que las clases stub cargadas pueden ser cargadas desde el codebase de nuestro servidor (que fue especificado por la propiedad java.rmi.server.codebase cuando se arrancó la aplicación servidor). Como resultado, el rmiregistry no enviará a los clientes un codebase asociado con las clases stub, y consecuentemente, nuestros clientes no podrán localizar y cargar las clases stub (u otras clases del lado del servidor).

Para arrancar el registro en el servidor, se ejecuta el comando rmiregistry. Este comando no produce ninguna salida y normalmente se ejecuta en segundo plano.

Detalles Específicos de la Plataforma: Arrancar el Registro en el Puerto por Defecto

Windows (utilizar javaw si no está disponible start):

```
unset CLASSPATH
start rmiregistry
```

UNIX:

```
unsetenv CLASSPATH
rmiregistry &
```

Por defecto el registro se ejecuta sobre el puerto 1099. Para arrancar el registro sobre un puerto diferente, se especifica el número de puerto en la línea de comandos. No olvidemos borrar el CLASSPATH.

Detalles Específicos de la Plataforma: Arrancar el Registro en el Puerto 2001

Windows:

```
start rmiregistry 2001
```

UNIX:

```
rmiregistry 2001 &
```

Una vez arrancado el registro, podemos arrancar el servidor. Primero, necesitamos asegurarnos de que el fichero compute.jar y la implementación del objeto remoto (que es lo que vamos a arrancar) están en nuestro path de clases.

Detalles Específicos de la Plataforma - Seleccionar la variable CLASSPATH

Windows:

```
set CLASSPATH=c:\home\ana\src;c:\home\ana\public_html\classes\compute.jar
```

Unix:

```
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

Cuando arrancamos el motor de cálculo, necesitamos especificar, utilizando la propiedad `java.rmi.server.codebase`, donde están disponibles las clases del servidor. En este ejemplo, las clases del lado del servidor disponibles son el stub de `ComputeEngine` y los interfaces `Compute` y `Task` disponibles en el directorio `public_html\classes` de `ana`.

Detalles Específicos de la Plataforma: Arrancar el Motor de Cálculo

Windows:

```
java -Djava.rmi.server.codebase=file:/c:\home\ana\public_html\classes/  
-Djava.rmi.server.hostname=zaphod.east.sun.com  
-Djava.security.policy=java.policy  
engine.ComputeEngine
```

UNIX:

```
java -Djava.rmi.server.codebase=http://zaphod/~ana/classes/  
-Djava.rmi.server.hostname=zaphod.east.sun.com  
-Djava.security.policy=java.policy  
engine.ComputeEngine
```

El comando `java` anterior define varias propiedades:

- `java.rmi.server.codebase`, una propiedad que especifica una localización, una URL codebase, de las clases originarias desde este servidor para que la información de las clases enviadas a otras máquinas virtuales incluya la localización de la clase que el receptor pueda descargar. Si el codebase especifica un directorio (como oposición a un fichero JAR), debemos incluir la barra inclinada en la URL.
- `java.rmi.server.hostname`, una propiedad que indica el nombre totalmente cualificado de nuestro servidor. En algunos entornos de red, el nombre totalmente cualificado del host no se puede obtener utilizando el API de Java. RMI hace el mejor esfuerzo para obtener ese nombre. Si uno de ellos no puede ser determinado, fallará y utilizará la dirección IP. Para asegurarnos de que el RMI utilizará un nombre de Host, podríamos seleccionar la propiedad `java.rmi.server.hostname` como medida de seguridad.
- `java.security.policy`, una propiedad utilizada para especificar el fichero de policía que contiene los permisos concedidos a los codebases especificados.

La clase stub de `ComputeEngine` se carga dinámicamente en la máquina virtual del cliente sólo cuando la clase no está disponible localmente y la propiedad `java.rmi.server.codebase` ha sido configurada apropiadamente, para la localización de la clase stub, cuando se arrancó el servidor. Una vez cargada la clase stub no necesitamos recargarla más veces para referencias adicionales a objetos `ComputeEngine`.

Arrancar el Cliente

Una vez que el registro y el motor se están ejecutando, podemos arrancar el cliente, especificando:

- la localización donde el cliente sirve sus clases (la clase Pi) utilizando la propiedad `java.rmi.server.codebase`.
- como argumentos de la línea de comandos, el nombre del host (para que el cliente sepa donde localizar el objeto remoto) y el número de decimales utilizado en el cálculo del número Pi.
- `java.security.policy`, una propiedad utilizada para especificar el fichero de policía que contiene los permisos adecuados.

Primero seleccionamos el CLASSPATH para ver el cliente de jones y el fichero JAR que contiene los interfaces. Luego se arranca el cliente de esta forma:

Detalles Específicos de la Plataforma: Arrancar el Cliente

Windows:

```
set CLASSPATH c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
      -Djava.security.policy=java.policy
      client.ComputePi localhost 20
```

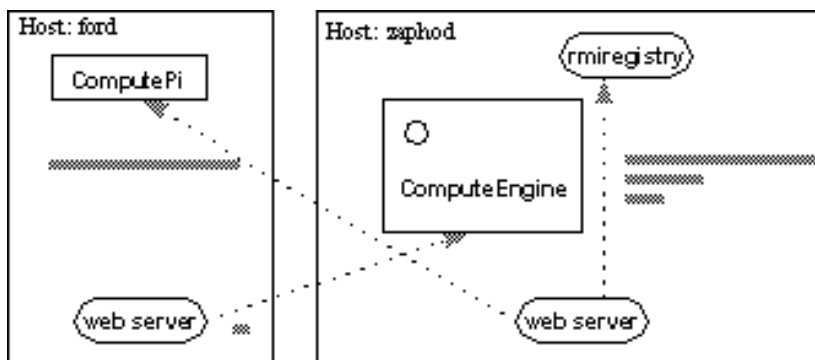
UNIX:

```
setenv CLASSPATH /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
      -Djava.security.policy=java.policy
      client.ComputePi zaphod.east.sun.com 20
```

Después de arrancar el cliente, deberíamos ver la siguiente salida en nuestra pantalla:

3.14159265358979323846

La siguiente figura muestra de dónde obtienen las clases el rmiregistry, el servidor ComputeEngine y el cliente ComputePi durante la ejecución del programa:



Cuando el servidor ComputeEngine coloca su referencia al objeto remoto en el registro, éste descarga el ComputeEngine_Stub, y también los interfaces Compute y Task de los

que la clase stub depende. Estas clases son descargadas del servidor web del ComputeEngine (o del sistema de ficheros, dado el caso).

El cliente ComputePi también carga ComputeEngine_Stub, desde el servidor web de ComputeEngine, como resultado de la llamada a Naming.lookup. Como el cliente tiene los dos interfaces disponibles en su path de clases, estas clases son cargadas desde allí, no de la localización remota.

Finalmente, la clase Pi se carga en la máquina virtual de ComputeEngine cuando el objeto Pi es pasado en la llamada al método remoto executeTask del objeto ComputeEngine. La clase Pi se carga desde la página web del cliente.

Ozito

Hacia el 1.1 -- y posteriores

Esta lección describe el API que se ha añadido al JDK desde el 1.0. Principalmente, trata sobre la versión 1.1 y sus diferencias con la versión 1.0, cómo utilizar las características de la 1.1, etc. También ofrece una visión sobre lo que podrás esperar después del 1.1.

Nota: El JDK 1.1 fue liberado en Diciembre de 1996. Y fue rápidamente seguida por el jdk 1.1.1 que corregía algunos errores. Generalmente hablando, las versiones correctoras de errores no añaden características nuevas sobre sus predecesores. Es por eso, que esta documentación que describe el JDK 1.1 se puede aplicar igualmente a las futuras versiones correctoras de errores.

[¿Qué hay de nuevo en la 1.1?](#) será el lugar para aprender sobre la versión 1.1. Esta podría ser tu primera parada si utilizar 1.0. Incluso si no lo utilizas, probablemente encontrarás esta lección muy útil como introducción a las capacidades que proporciona esta versión del JDK. Hasta que se termine esta lección puedes encontrar información en la [documentación de Sun](#) para la versión 1.1 del JDK. Para más información sobre cómo escribir programas que puedan utilizar las características del 1.1 y aún así funciones en la versión runtime de la 1.0, puedes ver [Escribir Programas Compatibles](#).

[Cambios en el GUI: El AWT crece](#) Describe cómo trabajan las clases relacionadas con el GUI -- en particular el AWT --, y las nuevas características que puedes esperar después de la 1.1. Esta lección incluye información detallada sobre la utilización del nuevo sistema de eventos, escribir componentes de peso ligero, y cómo convertir programas que utilizan el AWT 1.0 al 1.1.

[\[PENDIENTE...\] Convertir Programas 1.0 a 1.1](#) . . . Esta en construcción. Mientras tanto puedes ver la documentación sobre el 1.1 que describe [Cómo convertir programas al API del AWT 1.1.](#)

¿Qué hay nuevo en el JDK 1.1?

La versión 1.1 del JDK añade un montón de nuevas características que han sido reclamadas por los programadores de Java. Esta lección lista y describe brevemente estas nuevas características, te muestra donde encontrar información sobre ellas, y apunta la información de este tutorial que es afectada por estos cambios.

Nuevas características en el JDK 1.1

Aquí se listan las nuevas características de la versión 1.1 del JDK, Pulsa sobre los enlaces para obtener un breve resumen sobre esa característica.

- [Internacionalización](#)
- [Seguridad y Applets Firmados](#)
- [AWT Ampliado](#)
- [JavaBeans\(tm\)](#)
- [Formato de Ficheros JAR](#)
- [Ampliaciones del Trabajo en Red](#)
- [Ampliaciones de la I/O](#)
- [El paquete Math](#)
- [Invocación de Métodos Remotos](#)
- [Serialización de Objetos](#)
- [Reflexión](#)
- [Conectividad a Bases de Datos en Java](#)
- [Clases Internas](#)
- [El Interface Nativo](#)
- [Aumentos de Rendimiento](#)
- [Miscelanea](#)

[Cómo afecta la versión 1.1 a las lecciones existentes](#)

La versión 1.1 del JDK cambia algunas características que ya fueron documentadas en el tutorial para la versión 1.0.2. Como la mayoría de los navegadores más populares no soportan todavía la versión, 1.1, es un poco prematuro actualizar completamente el tutorial a esta versión. En su lugar, en aquellos lugares en que el tutorial describe características que han cambiado para la versión 1.1, se proporcionarán enlaces a las notas que describen estos cambios, cómo migrar tu código

a esta nueva tecnología, y algunas veces, ejemplos de código.

Podrás encontrar estas notas en el contexto de la lección.

Ozito

Internacionalización

La Internacionalización fue una de las características más ampliamente solicitadas por los clientes de Java. JavaSoft ha licenciado la solución de internacionalización proporcionada en el JDK 1.1 por [Taligent](#).

La caracterísitca de internacionalización del JDK está totalmente integrada y hace que los programadores tengan más sencillo el escribir programas y applets gobales desde el principio (no cómo un sobreesfuerzo).

Las aplicaciones incluyen el display de caracteres UNICODE, un mecanismo local, soporte de mensajes locales, fecha, hora y zona horaria sensibles y manejo de números, conversores de juegos de caracteres, formateo de parámetros y soporte para encontrar los límites de caracter/palabra/sentencia.

¿Dónde encontrar documentación?

- La nueva lección [Escribir programas globales](#), documenta estas nuevas características, y muestra cómo internacionalizar un programa existente.
- Además, en la site sun podrás encontrar información sobre cómo añadir fuentes a la runtime de Java, y algunas demos de Applets del JDK 1.1. [Internacionalización](#).

Seguridad y Applets Firmados

El API de seguridad de Java está diseñado para permitir a los desarrolladores que incorporen funcionalidad de seguridad tanto a bajo como a alto nivel dentro de sus aplicaciones Java. La primera versión de la seguridad Java en el JDK 1.1 contiene un subjuego de estas funcionalidades, incluyendo un API para firmas digitales y resumen de mensajes. Además existen interfaces abstractos para manejo de claves, manejo de certificados y control de acceso. Específicamente el API soporta certificados X.509 v3 y otros formatos de certificados, y una rica funcionalidad en el área de control de acceso, que continuará en versiones posteriores del JDK.

el JDK 1.1 también proporciona una herramienta que puede firmar archivos (JAR), que pueden contener clases y otros datos (cómo imágenes o sonidos). El appletviewer permite descargar applets en ficheros JAR firmados (utilizando la herramienta) mediante una entidad de confirmación para ejecutarlos con los mismos derechos que una aplicación local. Esto es, dichos applets no están sujetos a las restricciones de seguridad del modelo de seguridad original de Java. Las versiones posteriores del JDK proporcionarán unas políticas de seguridad más sofisticadas, incluyendo mayor granulidad en los niveles de conformidad permitidos.

¿Dónde encontrar documentación?

- Se ha creado una nueva lección titulada [Java Security 1.1](#). Si la información que necesitas no está ahí puedes leer la documentación en la site de Sun [Seguridad y Applets Firmados](#).

Ampliación del AWT

En el JDK 1.1, la arquitectura del AWT ha sido ampliada para hacer que el desarrollo de GUI a gran escala sea más fácil y para añadir funcionalidades básicas que se habían perdido en la versión 1.0.2. Lo más significativo es la creación de un nuevo modelo de eventos y la capacidad para crear componentes de poco peso. Se han añadido un API para imprimir, mejor desplazamiento de pantalla, menús popup, portapapeles (copiar/pegar), y cursores para componentes. También se han ampliado las capacidades gráficas y de imagen, y el soporte de fuentes se ha hecho más flexible para acomodarse a la internacionalización.

¿Dónde encontrar documentación?

- [Cambios en el GUI: El AWT crece](#) describe como trabajan las nuevas clases relacionadas con el GUI en el JDK 1.1. La lección incluye información detallada sobre la utilización del nuevo sistema de eventos, escribir componetes de peso ligero, y cómo convertir programas que utilizan el aWT 1.0 a la versión 1.1.
- Hasta que se complete la lección anterior, puedes encontrar las espcificaciones de diseño del AWT en la site de Sun [Ampliación del AWT](#).

JavaBeans(tm)

JavaBeans define el modelo de componente software reutilizable de Java. Estos APIs permiten a terceras partes crear y lanzar componentes Java reutilizables (Beans), como procesadores de texto, hojas de cálculo, o puentes gráficos, que juntos, pueden ser convertidos en aplicaciones por los no programadores.

¿Dónde encontrar documentación?

- En la lección de este tutorial titulada [JavaBeans](#), que incluye varios ejemplos.
- También puedes leer las [Especificaciones de los JavaBeans\(tm\)](#).

El formato de Ficheros JAR

JAR (Java ARchive) es un formato de fichero independiente de la plataforma que te permite almacenar un applet Java y sus componentes requeridos (ficheros .class, imágenes y sonidos) en un sólo fichero JAR. Utilizando el nuevo atributo ARCHIVE de la etiqueta <APPLET>, este fichero puede ser desargado por el navegador con un sola transacción HTTP, aumentando gratamente la velocidad de descarga. Además, JAR soporta compresión, lo que reduce el tamaño del fichero, y optimiza más aún el tiempo de descarga.

Finalmente, el autor del applet puede firma digitalmente entradas en el fichero JAR para autenticar su origen. Puedes leer más sobre esto en la lección de seguridad.

¿Dónde encontrar más documentación?

Actualmente, la cantidad de documentación disponible dentro del tutorial sobre los ficheros JAR está de algún modo limitada. Sin embargo, es suficiente para empezar.

- [Cambios en el JDK 1.1: Archivos JAR y los Applets](#) te enseña como utilizar los ficheros JAR en tus applets y explica los beneficios de hacerlo.
- La lección sobre los beans contiene algunas páginas sobre [Empaquetar Beans para su distribución](#).
- Los ficheros JAR se utilizan en un gran número de ejemplos de applets en el tutorial, incluyendo el ejemplo AroundtheWorld en [Escribir programas globales](#), y en la [Introducción al nuevo modelo de eventos del AWT](#).
- Y para ayudarte a empezar, aquí tienes una breve introducción a algunos de los conceptos básicos para utilizar ficheros JAR.

Puedes crear y manipular ficheros JAR con el programa de utilidad jar. Los argumentos de la línea de comandos para este programa son similares a los del programa tar del UNIX. Las cosas más comunes que puedes hacer con un fichero JAR son: crear un fichero JAR, listar el contenido de un fichero JAR, extraer el contenido de un fichero JAR, y utilizar un fichero JAR en el atributo ARCHIVE de la etiqueta <APPLET>. Brevemente, aquí tienes cómo hacer cada una de esas cosas (utilizando Solaris o Windows):

Para crear un fichero JAR:	jar cvf NombredeFicheroJAR ListadeFicheros
Para listar el contenido de un fichero JAR:	jar tvf NombredeFicheroJAR
Para extraer el contenido completo de un fichero JAR:	jar xvf NombredeFicheroJAR

Para especificar la utilización de un fichero JAR con un applet:

```
<applet code=AppletClassName.class  
archive="JarFileName.jar"  
width=width height=height>  
</applet>
```

Además, de la documentación anteriormente mencionada puedes encontrar, la Guía y Manifiesto de las Especificaciones del Formato JAR en la página [JAR - Java Archive](#) de la site de Sun.

Ampliaciones del Trabajo en Red

El JDK 1.1 hace muchas ampliaciones en el paquete java.net.

¿Dónde encontrar documentación?

- Estas ampliaciones se suman en la página [Notas sobre la 1.1](#) que se ha añadido a la lección "Seguridad y Conectividad de Cliente". Además se han aumentado los ejemplos que utilizan las ampliaciones de la versión 1.1 para que puedas ver los cambios que podrían afectar a tus programas.
- La página [Ampliaciones del Trabajo en Red](#) en el site de Sun, proporciona algún material descriptivo sobre los cambios en java.net.

Ampliaciones de la I/O

El paquete I/O ha sido ampliado con steams de carácter, que son como los stream de byte excepto que contienen caracteres Unicode de 16-bit en vez de bytes de ocho bits.

¿Dónde encontrar documentación?

- Estos nuevos streams están documentados en las páginas [Notas sobre la versión 1.1](#).
- [Ampliaciones de la I/O](#) tiene más documentación sobre estos cambios en el site de Sun.

El paquete Math

Se ha añadido un nuevo paquete al JDK 1.1, el paquete `java.math`. Este paquete contiene dos nuevas clases : `BigInteger` y `BigDecimal`.

Los números `BigInteger` son enteros de precisión arbitraria inmutable, lo que proporciona analogía con todos los operadores de enteros primitivos de Java, y todos los métodos estáticos relevantes de las clases de `java.lang.Math`.

Adicionalmente, los números `BigInteger` proporcionan operaciones para aritmética modular, cálculo GCD, testeo primario, primera generación, manipulación de bits, etc.

Los números `BigDecimal` son números decimales inmutables de precisión arbitraria, diseñados para el cálculo monetario. Estos números proporcionan operaciones para aritmética básica, manipulación de escalas, comparación y conversión de formatos.

¿Dónde encontrar documentación?

- Actualmente la única documentación disponible sobre estas clases se encuentra en la documentación sobre el API en el site de SUN:
[java.math.BigInteger](#), y [java.math.BigDecimal](#)

Invocación Remota de Métodos (RMI)

La Invocación Remota de Métodos (RMI) permite a los programados crear aplicaciones distribuidas Java-a-Java, en la que los métodos de objetos Java remotos pueden ser invocados desde otras máquinas virtuales Java, posiblemente en diferentes servidores.

Un programa Java puede hacer una llamada a un objeto remoto una vez que ha obtenido una referencia hacia ese objeto, o bien buscando el objeto en un servidor de nombres proporcionado por el RMI o recibiendo la referencia como un argumento o un valor de retorno. Un cliente puede llamar a un objeto remoto en el servidor, y este servidor puede a su vez ser un cliente de otros objetos remotos. RMI utiliza Serialización de Objetos para ordenar y desordenar parámetros y no trunca los tipos, soportando verdadero polimorfismo orientado a objetos.

¿Dónde encontrar documentación?

- Puedes encontrar el tutorial sobre el RMI y las especificaciones en el site de Sun [Documentación RMI](#).

Serialización de Objetos

La Serialización de Objetos extiende el corazón de las clases de I/O de Java con soporte para Objetos. La serialización soporta la codificación de objetos y los objetos alcanzables desde dentro de un stream de bytes y la reconstrucción complementaria del objeto desde el stream. La serialización es utilizada para persistencia de peso ligero y para comunicación mediante sockets o RMI. La codificación de objetos por defecto protege la transmisión de datos, y soporta la evolución de las clases. Una clase podría implementar su propia codificación externa y ser completamente responsable del formato externo.

¿Dónde encontrar documentación?

- Las especificaciones sobre serialización de objetos pueden encontrarse en [Serialización de Objetos](#).
- Además puedes ver la documentación sobre el API de [java.io](#).

Reflexión

Permite al código Java descubrir información sobre campos, métodos y constructores de las clases cargadas, y utilizar campos reflejados, métodos y constructores para operar con sus partes subrayadas de los objetos, dentro de las restricciones de seguridad. El API acomoda las aplicaciones que necesitan acceso a los miembros públicos o al objeto fuente (basándose en su clases runtime) o a los miembros declarados por un clase dada.

¿Dónde encontrar documentación?

- La expecificación sobre la Reflexión del Corazón de Java se puede encontrar en el site del JDK [Reflection Documentation](#).

Conectividad con Bases de Datos en Java

Conectividad con Bases de Datos en Java es un interface de acceso a base de datos SQL estándar. proporcionando un acceso uniforme a un amplio rango de bases de datos relacionales. También proporciona una base común a las herramientas de alto nivel y los interfaces que se pueden construir. Esto viene como un "Puente ODBC" (excepto en Mac 68K). El puente es una librería que implementa el JDBC en términos del API estándar del ODBC de "C".

¿Dónde encontrar documentación?

- Puedes encontrar el tutorial sobre JDBC en el site de Sun [JDBC Documentation](#).

Clases Internas

Las versiones previas del lenguaje Java requerían que todas las clases fueran declaradas miembros de un paquete (llamadas clases de máximo nivel). La versión del JDK 1.1 elimina esta restricción y permite declarar clases en cualquier ámbito (llamadas clases Internas).

¿Dónde encontrar documentación?

- Puedes ver la página [Notas sobre el JDK 1.1](#).
- Puedes ver las [Especificaciones de las clases internas de Java](#).

Interface Nativo de Java

Los métodos nativos, son metodos utilizados por Java pero escritos en un lenguaje diferente, han estado en el JDK desde el principio. Y como se prometió, el interface de los métodos nativos del 1.0.2 ha sido reescrito completamente. Este interface se llama ahora Interface de Java Nativo o JNI para acortar.

¿Dónde encontrar documentación?

- Los nuevos métodos nativos en el JDK 1.1 están documentados en la lección [Integrar código nativo y programas Java](#)
- O si lo prefieres, puedes leer en la site de Sun las [Especificaciones de los Métodos Nativos en Java](#).

Aumento del Rendimiento en el JDK 1.1

LA siguiente lista describe los aumentos de rendimiento realizados en la versión 1.1 del JDK:

- Bucle de Interprete en código ensamblador en Win32 y Solaris/SPARC
Ya que algunas porciones de la Máquina Virtual Java se han reescrito en lenguaje ensamblador, la máquina resultante ahora se ejecuta hasta 5 veces más rápido en ciertas operaciones.
- Soporte de Pila no-continua para Mac
Aumentar la utilización de la memoria en Macintosh permite que Java se ejecute de forma más correcta con otras aplicaciones.
- Aceleración de los Monitores
Los métodos sincronizados permiten operaciones para ejecutarse más rápida y eficientemente.
- Recolección de basura de Clases
Esta ampliación descarta automáticamente las clases no utilizadas. Al aumentar la utilización de memoria de la máquina virtual permite a Java operar más eficientemente y con menos memoria.
- Las clases pares del AWT re-escritas para Win32
Para alcanzar un alto rendimiento, las clases pares del AWT ha sido completamente re-escrita para aumentar su velocidad en Win32.
- JAR (Archivos Java) empaquetado de recursos para una sola transacción HTTP
JAR es un nuevo formato de ficheros independiente de la plataforma proporcionado por JDK 1.1 que permite empaquetar varios ficheros en uno sólo. Los applets Java y sus componentes - como fichero .class, imágenes o sonido -- pueden ser empaquetados en un fichero JAR comprimido y descargado por el navegador con una sola transacción HTTP para reducir el tiempo de descarga. Además, los autores de los applets pueden firmar digitalmente entradas individuales en el fichero JAR para autenticar su origen.

¿Dónde encontrar documentación?

- Para las medidas de rendimiento del JDK puedes ver [Performance Measurements](#) en el site de JavaSoft.

Miscelánea

Los siguientes cambios también se han realizado en la versión 1.1 del JDK pero no pueden ser categorizados con ningún otro.

- Las clases Byte, Short y Void
Los Bytes y los Shorts se han acomodado como número recortador mediante la adición de las clases Byte y Short. La clase abstracta Number obtiene dos nuevos métodos concretos: byteValue y shortValue; las implementaciones por defecto de estos utilizan el método intValue. También incluye una clase Void que es una clase no ejemplarizable.
- La etiqueta @deprecated
Utilizada en documentación para marcar las clases no ambíguas, los métodos y campos que han sido superados por los nuevos APIs. El compilador lanzará un aviso cuando procese código fuente que utilice la característica despreciada.
- Acceso a ficheros de recursos
Este API proporciona un mecanismo para localizar ficheros de recursos en una forma independiente de la localización de los recursos. Por ejemplo, este mecanismo puede localizar un fichero de recursos tanto si en un applet cargar desde la red utilizando múltiples conexiones HTTP, un applet cargado utilizando fichero JAR, o una librería instalada en el CLASSPATH.
- Adiciones a la etiqueta APPLET (HTML)
Ampliaciones de la etiqueta <APPLET> utilizada en HTML.

¿Dónde encontrar documentación?

- La única información disponible actualmente para las clases [Byte](#), [Short](#), y [Void](#) está en la documentación generada por JavaDoc.
- En la página [The @deprecated Tag](#) de Sun puedes encontrar información sobre los viejos APIs.
- [Accessing Resources](#) Describe cómo puedes localizar recursos de una forma independiente de la posición.
- [APPLET Tag](#) Documenta todo los componentes de la etiqueta <APPLET> incluyendo los nuevos del JDK 1.1.

Cómo afecta el JDK 1.1 a las lecciones existentes

Esta página resume cómo afectan los cambios del JDK 1.1 a las lecciones de este tutorial.

Escribir applets

El API básico de Applet no ha cambiado, excepto por la adición de un atributo a la etiqueta <APPLET>. El nuevo parámetro ARCHIVE, te permite especificar el fichero o ficheros JAR que contienen las clases y los ficheros de datos.

Como los applets pueden utilizar la mayoría de las clases del JDK, la mayoría de los cambios descritos en las otras lecciones, también se aplican a los applets.

Crear un Interface de usuario

El AWT ha cambiado significativamente, pero mantiene su compatibilidad. Puedes ver [Cambios del GUI: el AWT Crece](#) para dar un vistazo a la documentación detallada del nuevo API.

Trabajo en Red

El JDK 1.1 ha realizado varias ampliaciones en el paquete de red, más notablemente ha extendido los sockets y las opciones del estilo BSD.

Cambios del GUI: el AWT crece

Después de liberar la versión 1.0 de la plataforma Java, JavaSoft empezó a trabajar en formas para permitir a los programadores crear mejores interfaces gráficos de usuario (GUIs) en el menor tiempo.

En el JDK 1.1, la arquitectura del AWT se ha ampliado para hacer el desarrollo de GUI a gran escala más fácil y para añadir funcionalidades básicas que se habían perdido. La más significativa de las ampliaciones es un nuevo modelo de eventos y la capacidad para crear componentes de peso ligero.

Justo después de finalizar la 1.1, JavaSoft anunció la Java Foundation Classes (JFC). El JFC incluye el AWT del 1.1 más funcionalidades adicionales, como una infraestructura para crear componentes de peso ligero, un juego de estos componentes listos para utilizar, y una capacidad de dibujo completamente caracterizada. Para obtener realimentación y probar estas funcionalidades adicionales de JFC, JavaSoft proporciona acceso a las primeras versiones de los proyectos Java 2D (dibujo) y Swing (componentes de peso ligero).

Cambios en el GUI 1.1 y posteriores

Esta sección discute el JFC -- los cambios del AWT 1.1, el proyecto Swing y un poco sobre [Java 2D](#).

[El nuevo modelo de eventos del AWT](#)

Esta sección describe el modelo de eventos del AWT 1.1, ofreciendote cantidad de ejemplos a seguir.

[Utilizar la versión de "Swing" del JFC](#)

La versión de Swing es una primera versión de una parte del JFC. Esta sección de cuenta como descargar la versión de Swing, y construir aplicaciones que utilicen componentes de peso ligero ya creados.

Escribir componentes de peso ligero

Esta sección no está escrita todavía. Puedes ver la página de ejemplos en el site de JavaSoft [Lightweight Components](#) para ver documentación y ejemplos. Cuando la versión de Swing se incorpore al JDK, escribir componentes de peso ligero -- y componentes de usuario en general -- será más sencillo.

Cómo convertir programar al API del AWT 1.1

Para esta información, debes ir a la site de JavaSoft [Cómo convertir](#)

[programas al API del aWT 1.1](#), que es una parte de la documentación del paquete 1.1. Estos contenidos se añadirán pronto a este tutorial.

Sobre los ejemplos: Muchos de los ejemplos de esta sección utilizan componentes de peso pesado. Sin embargo, te recomendamos que utilices componentes ligeros siempre que sea posible!

El nuevo modelo de eventos del AWT

Así como el JDK, el AWT tiene un nuevo modelo de eventos. El viejo modelo [basado en contenido](#) todavía funciona, pero su uso está desaconsejado porque el nuevo modelo es mucho más flexible, poderoso y eficiente. El nuevo modelo está basado en el modelo de eventos de los [JavaBeans](#), como un paso hacia adelante soportando componentes AWT como Beans.

[Introducción al nuevo modelo de eventos del AWT](#)

Esta sección explica el nuevo modelo de evento y contiene algunos applets ilustrativos.

[Utilizar adaptadores y clases internas para manejar eventos del AWT.](#)

Para reducir el código innecesario, puedes utilizar adaptadores y clases internas. Esta sección muestra cómo y cuándo hacer esto

[Manejar eventos estándar del AWT](#)

Esta sección lista todos los eventos que pueden generar los componentes del AWT. Y dará ejemplos de su manejo.

Generar eventos AWT

Tus objetos pueden generar tanto eventos estándar como clientes de AWT. Esta sección te lo contará. Por ahora puedes ver estas páginas en la site de JavaSoft [Lightweight Components](#) para ejemplos de la generación de eventos. O [JDK 1.1 Event Examples](#) para ver un ejemplo de la forma de redespachar un evento AWT.

Sumario del modelo de eventos del AWT

Esta sección lo pondrá todo junto, sumando lo que has aprendido y proporcionando algunos detalles adicionales.

Introducción al nuevo modelo de ventos del AWT

En el nuevo modelo de eventos, los eventos son generados por fuentes de ventos. Uno o más oyentes pueden registrarse para ser notificados sobre los eventos de un tipo particular sobre una fuente particular. Algunas veces este modelo es llamado delegación, ya que permite al programador delegar la autoridad del manejo del evento a cualquier objeto que implemente el interface de oyente apropiado. El nuevo modelo de eventos del AWT de permite tanto manejar como generar eventos AWT.

Los manejadores de eventos pueden ser ejemplares de cualquier clase. Siempre que una clase implemente el interface de oyente de eventos, sus ejemplares pueden manejar eventos. En todo programa que tenga un manejador de eventos, veras tres trozos de código:

1. En la sentencoa class del manejador de eventos, El código que declara que la clase implementa un interace oyente (o extiende una clase que implementa el interface oyente). Por ejemplo:

```
public class MiClase implements ActionListener {
```

2. El código que registra un ejemplar de la clase manejadora de eventos como un oyente para uno o más componentes. Por ejemplo:

```
someComponent.addActionListener(instancedeMiClase);
```

3. La implementación de métodos en el interface oyente. Por ejemplo:

```
public void actionPerformed(ActionEvent e) {  
    ...//código que reacciona a la acción...  
}
```

Un ejemplo sencillo

Aquí tienes un applet del 1.1 que ilustra el manejo de eventos. Contiene un sólo botón que hace un Beep cuando lo pulsas.

Puedes encontrar el programa completo en [Beeper.java](#). Aquí sólo el código que implementa el manejo de eventos para el botón:

```
public class Beeper ... implements ActionListener {  
    ...  
    //Donde ocurre la inicialización:  
    button.addActionListener(this);  
    ...  
    public void actionPerformed(ActionEvent e) {  
        ...//Hace un sonido Beep...  
    }  
}
```

¿No es sencillo? La clase Beeper implementa el interface [ActionListener](#), que contiene

un método: actionPerformed.

Cómo Beeper implementa ActionListener, un objeto Beeper puede registrarse como oyente de los eventos de acción que generen los botones. Una vez que Beeper ha sido registrado utilizando el método addActionListener del button, el método actionPerformed del Beeper es llamado cada vez que se pulsa el botón.

Un ejemplo más complejo

El modelo de eventos del 1.1 que has podido ver en su forma más sencilla en el ejemplo anterior, es bastante poderoso y flexible. Cualquier número de objetos oyentes puede escuchar todas las clases de eventos desde cualquier número de objetos fuentes. Por ejemplo, un programa podría crear un oyente para cada fuente de evento. O un programa podría tener un sólo oyente para todos los eventos de todas las fuentes. Incluso un programa puede tener más de un oyente para una sola clase de evento desde un sólo objeto fuente.

El siguiente applet ofrece un ejemplo de utilización de múltiples oyentes por objeto. El applet contiene dos fuentes de eventos (ejemplares de Button y dos oyentes de eventos. Uno de los oyentes (un ejemplar de la clase llamada MultiListiner escucha los eventos de los dos botones. Cuando recibe un evento, añade el "comando de acción" del evento (el texto de la etiqueta del botón) en la parte superior del área de texto. El segundo oyente (un ejemplar de la clase llamada Eavesdropper) escuchas los eventos de uno sólo de los botones. Cuando recibe un evento, añade el comando de acción en la parte inferior del área de texto.

Puedes encontrar el programa completo en [MultiListener.java](#). Aquí sólo tienes el código que implementa el manejo de eventos para el botón:

```
public class MultiListener ... implements ActionListener {
    ...
    //donde ocurra la inicialización:
    button1.addActionListener(this);
    button2.addActionListener(this);

    button2.addActionListener(new Eavesdropper(bottomTextArea));
}

public void actionPerformed(ActionEvent e) {
    topTextArea.append(e.getActionCommand() + "\n");
}

}

class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        myTextArea.append(e.getActionCommand() + "\n");
    }
}
```


En el código anterior, tanto MultiListener como Eavesdropper implementan el interface ActionListener y se registran como oyentes de acción utilizando el método addActionListener de la clase Button. La implementación del método actionPerformed en ambas clases es similar: sólo añaden el comando de acción al área de texto.

Un ejemplo de manejo de eventos de otro tipo

Hasta ahora, la única clase de eventos que has visto han sido los eventos action. Echemos un vistazo a un programa que maneja otra clase de eventos: eventos del ratón.

El siguiente applet muestra un rectángulo elevado y un área de texto. Cuando ocurre un evento del ratón -- un click, pulsación, liberación, entrar o salir -- dentro del área del rectángulo (BlankAreaMouseDemo), el área de texto muestra la cadena que describe el evento.

Puedes encontrar el programa completo en [MouseDemo.java](#). Aquí sólo está el código que implementa el manejo de eventos:

```
public class MouseDemo ... implements MouseListener {
    ...
    //Donde ocurra la inicialización:
    //Registra los eventos del ratón en blankArea y applet (panel).
    blankArea.addMouseListener(this);
    addMouseListener(this);
}

public void mousePressed(MouseEvent e) {
    saySomething("Mouse button press", e);
}

public void mouseReleased(MouseEvent e) {
    saySomething("Mouse button release", e);
}

public void mouseEntered(MouseEvent e) {
    saySomething("Cursor enter", e);
}

public void mouseExited(MouseEvent e) {
    saySomething("Cursor exit", e);
}

public void mouseClicked(MouseEvent e) {
    saySomething("Mouse button click", e);
}

void saySomething(String eventDescription, MouseEvent e) {
```

```
        textArea.append(eventDescription + " detected on "
                        + e.getComponent().getClass().getName()
                        + ".\n");
        textArea.setCaretPosition(maxInt); //hack to scroll to bottom
    }
}
```

Podrás ver el código explicado en [Implementar un oyente del Ratón.](#)

Ozito

Utilizar adaptadores y clases internas para manejar eventos del AWT

Esta sección te cuenta cómo utilizar los adaptadores y las clases internas (inner) para reducir el atestamiento de tu código. Si no te importa esto, puedes saltar libremente a la [siguiente sección](#).

La mayoría de los interfaces oyentes del AWT, al contrario que ActionListener, contienen más de un método. Por ejemplo, el interface MouseListener contiene cinco métodos: mousePressed, mouseReleased, mouseEntered, mouseExited, y mouseClicked. Incluso si sólo te importan los clicks del ratón, si tu clase implementa directamente MouseListener, debes implementar estos cinco métodos. Los métodos que no te interesen pueden tener cuerpos vacíos. Aquí tienes un ejemplo:

```
//Un ejemplo de atestamiento pero con código válido.
MyClass implements MouseListener {
    ...
    someObject.addMouseListener(this);
    ...
    /* Definición de método vacío. */
    public void mousePressed(MouseEvent e) {
    }

    /* Definición de método vacío. */
    public void mouseReleased(MouseEvent e) {
    }

    /* Definición de método vacío. */
    public void mouseEntered(MouseEvent e) {
    }

    /* Definición de método vacío. */
    public void mouseExited(MouseEvent e) {
    }

    public void mouseClicked(MouseEvent e) {
        ...//La implementación del manejador del eventos va aquí...
    }
}
```

Desafortunadamente, el resultado de la colección de cuerpos de métodos vacíos puede hacer el código difícil de leer y de mantener. Para ayudarte a evitar el atestamiento de tu código con cuerpos de métodos vacíos, el AWT proporciona una clase adaptadora para cada interface oyente con más de un método. ([Manejar eventos estándar del AWT](#) lista todos los oyentes y sus adaptadores.) Por ejemplo, la clase MouseAdapter implementa el interface MouseListener. Una clase adaptador implementa versiones vacías para todos sus métodos del interface.

Para utilizar un adaptador, crea una subclase de él, en lugar de implementar directamente un interface oyente. Por ejemplo, extendiendo `MouseAdapter`, tu clase puede heredar definiciones vacías para los cinco métodos contenidos por `MouseListener`.

```
/*
 * Un ejemplo de extensión de una clase adaptador en vez de
 * implementar directamente un interface oyente.
 */
MyClass extends MouseAdapter {
    ...
    someObject.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
        ...//La implementación del manejador de eventos va aquí...
    }
}
```

¿Qué pasa si no quieres que tu clase manejadora de eventos descienda de una clase adaptador? Por ejemplo, supón que has escrito un applet, y quieres que tu subclase de `Applet` contenga algo de código para menajar eventos del ratón. Como el lenguaje Java no permite la herencia múltiple, tu clase no puede descender de `Applet` y `MouseAdapter` a la vez. La solución es definir una clase inner -- una clase dentro de tu subclase de `Applet` -- que extienda la clase `MouseAdapter`,

```
//Un ejemplo de implementación de las clases internas.
MyClass extends Applet {
    ...
    someObject.addMouseListener(new MyAdapter());
    ...
    class MyAdapter extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            ...//La implementación del manejador de eventos va aquí...
        }
    }
}
```

Las clases internas funcionan bien incluso si tu manejador de eventos necesita acceder a variables privadas de la clase superior. Siempre que no declares una clase interna como `static`, ésta puede referirse a ejemplares de variables y métodos como si su código estuviera en la clase superior.

A lo largo de esta lección verás el uso de las clases internas. Para más información sobre las clases inner puedes ver la [Especificación de las Clases Inner](#) en la site de JavaSoft.

Manejo de Eventos Estándar del AWT

Esta sección te cuenta cómo escribir un oyente para cada clase de evento que define el AWT 1.1. Primero te ofrece una introducción a los oyentes del AWT. Después cada tipo de oyente es explicado en su propia sub-sección.

En la tabla que sigue, cada fila describe un grupo particular de eventos que corresponde a un interface oyente. La primera columna da el nombre del interface oyente, con un enlace a la página del tutorial que lo describe. La segunda columna nombra la clase adaptador correspondiente, si la hay. (Para una explicación sobre la utilización de los adaptadores, puedes ver la [página anterior](#).) La tercera columna lista los métodos que contiene el interface oyente.

Para saber qué componentes del AWT 1.1 generan qué tipos de eventos puedes ver, [Eventos Generados por los componentes del AWT](#).

Interface Oyente	Clases Adaptador	Métodos
ActionListener	ninguna	actionPerformed(ActionEvent)
AdjustmentListener	ninguna	adjustmentValueChanged(AdjustmentEvent)
ComponentListener	ComponentAdapter	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
ContainerListener	ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
ItemListener	ninguna	itemStateChanged(ItemEvent)
KeyListener	KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
TextListener	ninguna	textValueChanged(TextEvent)

<u>WindowListener</u>	WindowAdapter	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)
---------------------------------------	---------------	--

Los eventos AWT descritos en la tabla anterior se pueden dividir en dos grupos: eventos de bajo nivel y eventos semánticos. Los eventos de bajo nivel representan las ocurrencias del sistema de ventanas o entradas a bajo nivel, Claramente, los eventos del ratón y del teclado -- ambos son resultado de una entrada directa del usuario -- son eventos de bajo nivel.

Los eventos de componentes, contenedores, foco y ventanas, también son eventos de bajo nivel. Los eventos de componente te permiten seguir los cambios de posición, tamaño y visibilidad de un componente. Los eventos de contenedor te permiten conocer si se ha añadido o eliminado cualquier componente a un contenedor particular. Los eventos de foco, te dicen cuando un componente obtiene o pierde el foco del teclado -- la habilidad de recibir caracteres tecleados en el teclado. Los eventos de ventana te mantienen informado sobre el estado de cualquier clase de Window, como un Dialog o un Frame.

Los eventos del ratón se dividen en dos grupos -- eventos de movimiento del ratón y el resto -- por lo que un objeto puede oír los eventos del ratón para las pulsaciones sin tener que sobrecargar el sistema generando y enviando eventos de movimiento del ratón, lo que tiende a ocurrir frecuentemente.

Los eventos semánticos incluyen los eventos de acción, ajuste, ítem, y texto. Estos eventos son el resultado de una interacción del usuario con un componente específico. Por ejemplo, un botón genera un evento acción cuando el usuario lo pulsa, y una lista genera un evento acción cuando el usuario hace doble click sobre un ítem. Los eventos de ajuste ocurren cuando un usuario cambia del valor de una barra de desplazamiento de cualquier forma. Cuando un usuario selecciona un ítem de un grupo de elementos (como una lista) se genera un evento de ítem. Los eventos de texto ocurren siempre que se cambia el texto dentro de un área o campo de texto.

Las siguientes páginas explican con más detalles cada tipo de evento.

Eventos Generados por los Componentes AWT

Esta tabla lista las clases de eventos que puede generar cada componente del AWT 1.1. Para ver una tabla que lista los oyentes y los tipos de adaptadores junto con los métodos que contienen, puedes ver [Manejar Eventos Estándar del AWT](#).

Componente AWT	Tipos de Eventos que puede Generar										
	action	adjustment	component	container	focus	item	key	mouse	mouse motion	text	window
Button	X		X		X		X	X	X		
Canvas			X		X		X	X	X		
Checkbox			X		X	X	X	X	X		
CheckboxMenuItem Nota: No es una subclase de Component!	*					X					
Choice			X		X	X	X	X	X		
Component			X		X		X	X	X		
Container			X	X	X		X	X	X		
Dialog			X	X	X		X	X	X		X
Frame			X	X	X		X	X	X		X
Label			X		X		X	X	X		
List	X		X		X	X	X	X	X		
MenuItem Nota: No es una subclase de Component!	X										
Panel			X	X	X		X	X	X		
Scrollbar		X	X		X		X	X	X		
ScrollPane			X	X	X		X	X	X		
TextArea			X		X		X	X	X	X	
TextComponent			X		X		X	X	X	X	
TextField	X		X		X		X	X	X	X	
Window			X	X	X		X	X	X		X

*CheckboxMenuItem hereda addActionListener de MenuItem, pero no genera eventos action.

Escribir un Oyente de "Action"

Los oyentes de "Action" probablemente son los más sencillos -- y más comunes -- manejadores de eventos para implementar. Se implementan para responder a una indicación del usuario de que una acción dependiente de la implementación debería ocurrir. Por ejemplo, cuando un usuario pulsa un [botón](#), hace doble click sobre un ítem de una [lista](#), elige un ítem de un [menú](#), o pulsa return en un [campo de texto](#), ocurre un evento action. El resultado es que se envía un mensaje actionPerformed a todos los oyentes que están registrados en un componente relevante

Métodos del Evento Action

El interface [ActionListener](#) contiene un sólo método y no tiene clase adaptador correspondiente. Aquí tienes el único método de ActionListener:

```
void actionPerformed(ActionEvent)
```

Llamado por el AWT justo después de que el usuario informe al componentes que debería ocurrir una acción.

Ejemplos de manejo del evento Action

Aquí tienes el código de manejo del evento Action para un applet llamado [Beeper.java](#):

```
public class Beeper ... implements ActionListener {
    ...
    //Donde ocurra la inicialización:
    button.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent e) {
        ...//Make a beep sound...
    }
}
```

Puedes encontrar ejemplos de oyentes de Action en las siguientes secciones:

[Un ejemplo sencillo](#)

Contiene y explica el applet cuyo código has visto arriba.

[Un ejemplo más complejo](#)

Contiene y explica un applet que tiene dos fuentes de Action y dos oyentes de Action, con un oyente que escucha las dos fuentes y el otro sólo escucha uno de ellos.

La clase `ActionEvent`

El método `actionPerformed` tiene un sólo parámetro, un objeto [ActionEvent](#). La clase `ActionEvent` define dos métodos útiles:

`getActionCommand`

Devuelve la cadena asociada con esta acción. Generalmente, ésta es la etiqueta del componente -- o del ítem seleccionado dentro del componente -- que genera la acción.

`getModifiers`

Devuelve un entero que representa las teclas modificadores que el usuario estaba pulsando cuando ocurrió el evento `Action`. Se pueden utilizar las constantes `ActionEvent.SHIFT_MASK`, `ActionEvent.CTRL_MASK`, `ActionEvent.META_MASK`, y `ActionEvent.ALT_MASK` para determinar qué teclas estaban pulsadas. Por ejemplo, si el usuario seleccionó con Shift un ítem de menú, la siguiente expresión será distinta de cero:

```
actionEvent.getModifiers() & ActionEvent.SHIFT_MASK
```

También es útil el método `getSource`, que devuelve el objeto (un componente o menú) que generó el evento `Action`. El método `getSource` está definido en una de las superclases de `ActionEvent`, [EventObject](#).

Escribir un Oyente de Adjustment

Los eventos Adjustment notifican los cambios en los valores de los componentes que implementan el interface [Adjustable](#). Los objetos Adjustable tienen un valor entero y generan eventos Adjustment cuando el valor cambia.

Además de la clase [Scrollbar](#), que implementa directamente Adjustable, las clases de barras de desplazamiento vertical y horizontal [ScrollPane](#) implementan Adjustable. Si quieres cazar los eventos Adjustment dentro de un panel desplazable, puedes obtener el objeto Adjustable necesario utilizando los métodos `getVAdjustable` y `getHAdjustable`.

Existen cinco tipos de eventos Adjustment:

`track`

El usuario ajusta explícitamente el valor del componente. Para una barra de desplazamiento, esto podría ser el resultado de un arrastre del cursor de la barra de desplazamiento.

`unit increment`

`unit decrement`

El usuario indica que desea un ajuste ligero del valor del componente. Para una barra de desplazamiento, esto podría ser el resultado de una pulsación del usuario sobre una de las flechas laterales.

`block increment`

`block decrement`

El usuario indica que desea ajustar el valor del componente en una cantidad superior. Para una barra de desplazamiento, esto podría ser el resultado de una pulsación del usuario sobre el cuerpo de la barra de desplazamiento, entre las flechas y el cursor.

Métodos del Evento Adjustment

El interface [AdjustmentListener](#) tiene sólo un método, por lo tanto no tiene clase adaptador correspondiente. Aquí tienes el método:

```
void adjustmentValueChanged(AdjustmentEvent)
```

Llamado por el AWT justo después de que cambie el valor del componente escuchado.

Ejemplos de manejo de eventos Adjustment

[No disponible].

La clase AdjustmentEvent

Cada método tiene un sólo parámetro: un objeto [AdjustmentEvent](#). La

clase AdjustmentEvent define los siguientes métodos:

Adjustable getAdjustable()

Devuelve el componente que generó el evento.

int getAdjustmentType()

Devuelve el tipo de ajuste que ha ocurrido. El valor devuelto es una de las siguientes constantes definidas en la clase

AdjustmentEvent: UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK.

int getValue()

Devuelve el valor del componente justo después de que haya ocurrido el ajuste.

Escribir un Oyente de Component

Uno o más eventos de Component son generados por un objeto Component justo después de que el componente se haya ocultado, se haya movido, hecho visible o redimensionado. Un ejemplo de un oyente de Component podría ser una herramienta de construcción de GUI que mostrara información sobre el tamaño del componente seleccionado, y que necesite conocer cuando cambia el tamaño del componente. Necesitarías utilizar eventos de componente para manejar la disposición y manejo básico.

Los eventos de componente oculto y componente visible ocurren sólo como resultados de llamadas al método setVisible de Component (o sus equivalentes show y hide). Por ejemplo, una ventana podría ser miniaturizada en un ícono (iconificada) sin que se haya generado un evento de componente oculto.

Métodos de Evento de Component

El interface [ComponentListener](#) y su clase adaptador correspondiente, [ComponentAdapter](#), contienen cuatro métodos:

`void componentHidden(ComponentEvent)`

Llamado por el AWT después de que el componente este oculto como resultado de una llamada al método setVisible.

`void componentMoved(ComponentEvent)`

Llamado por el AWT después de que el componente se haya movido, relativo a su contenedor. Por ejemplo, si una ventana se mueve, esta ventana genera un evento de componente movido, pero el componente que la contiene no.

`void componentResized(ComponentEvent)`

Llamado por el AWT después de que el componente haya cambiado su tamaño.

`void componentShown(ComponentEvent)`

Llamado por el AWT después de que el componente se vuelva visible como resultado de una llamada al método setVisible.

Ejemplos de manejo de eventos de Component

El siguiente applet demuestra los eventos de componente. El applet trae una ventana (Frame) que contiene una etiqueta y un checkbox. El checkbox controla si la etiqueta es visible o no. Siempre que el applet arranca (como cuando visitas o revisitas la página que contiene el applet), la ventana se hace visible. Siempre que el applet se para (cuando dejas la página que contiene al applet), esta ventana se oculta. Un área de texto muestra un mensaje cada vez que la ventana, la etiqueta o el checkbox generan un evento de componente.

Intenta esto:

1. Pulsa sobre el checkbox para ocultar la etiqueta.
La etiqueta genera un evento de componente oculto.
 2. Pulsa de nuevo sobre el checkbox para mostrar la etiqueta.
La etiqueta genera un evento de componente visible.
 3. Iconifica y des-iconifica la ventana llamada "SomeFrame".
No se obtiene ningún evento de componente oculto o visible.
 4. Cambia el tamaño de la ventana "SomeFrame".
Verás el evento de componente redimensionado (y posiblemente el de componente movido) para los tres componentes -- etiqueta, checkbox y ventana. Si el manejador de distribución de la ventana no hace cada componente tan ancho como sea posible, la etiqueta y el checkbox no se habrán redimensionado.
-

Puedes encontrar el código del applet en [ComponentDemo.java](#). Aquí tienes el código de manejo de eventos Component del applet:

```
public class ComponentDemo ... implements ComponentListener {
    ...
    //Donde ocurra la inicialización:
    someFrame = new SomeFrame(this);
    ...

    public void componentHidden(ComponentEvent e) {
        displayMessage("componentHidden event from "
            + e.getComponent().getClass().getName());
    }

    public void componentMoved(ComponentEvent e) {
        displayMessage("componentMoved event from "
            + e.getComponent().getClass().getName());
    }

    public void componentResized(ComponentEvent e) {
        displayMessage("componentResized event from "
            + e.getComponent().getClass().getName());
    }

    public void componentShown(ComponentEvent e) {
        displayMessage("componentShown event from "
            + e.getComponent().getClass().getName());
    }
}
```

```
class SomeFrame extends Frame ... {  
    ...  
    SomeFrame(ComponentListener listener) {  
        ...  
        label.addComponentListener(listener);  
        checkbox.addComponentListener(listener);  
        this.addComponentListener(listener);  
        ...  
    }  
    ...  
}
```

La clase **ComponentEvent**

Cada método de evento de Component tiene un sólo parámetro, un objeto [ComponentEvent](#). La clase ComponentEvent define un método útil, `getComponent`, que devuelve el Componente que generó el evento.

Escribir un Oyente de Container

Los eventos de Container son generados por un Container justo después de que sea añadido o eliminado algún componente. Estos eventos son sólo para notificación -- no se necesita que esté presente ningún oyente de Container para que los componentes sean añadidos o eliminados satisfactoriamente.

Métodos de Eventos de Container

El interface [ContainerListener](#) y su correspondiente clase adaptador, [ContainerAdapter](#), contienen dos métodos:

`void componentAdded(ContainerEvent)`

Llamado por el AWT justo después de que se haya añadido un componente al contenedor.

`void componentRemoved(ContainerEvent)`

Llamado por el AWT justo después de que se haya eliminado un componente del contenedor.

Ejemplos de Manejo de Eventos de Container

El siguiente applet demuestra los eventos de container. Pulsando sobre los botones de "Add a button" o "Remove a button", puedes añadir o eliminar componentes en el panel que hay debajo del applet. Cada vez que un componente es añadido o eliminado, el panel dispara un evento de container, y se le notifica al panel oyente. El oyente muestra un mensaje descriptivo en el área de texto que hay encima del applet.

Intena esto:

1. Pulsa el botón llamado "Add a button".
Verás que aparece un botón en la parte inferior del applet. El contenedor oyente (en este ejemplo, un ejemplar de `ContainerDemo`) reacciona al evento de añadir un componente mostrando el mensaje "Button #1 was added to java.awt.Panel" en la parte superior del applet.
 2. Pulsa el botón llamado "Remove a button".
Esto elimina el último botón añadido al panel, haciendo que el oyente reciba un evento de componente eliminado.
-

Puedes encontrar el código de este applet en [ContainerDemo.java](#). Aquí tienes el código de manejo de eventos del applet:

```
public class ContainerDemo ... implements ContainerListener ... {  
    ...//Donde ocurra la inicialización:  
    buttonPanel = new Panel();  
    buttonPanel.addContainerListener(this);  
}
```

```

...
public void componentAdded(ContainerEvent e) {
    displayMessage(" added to ", e);
}

public void componentRemoved(ContainerEvent e) {
    displayMessage(" removed from ", e);
}

void displayMessage(String action, ContainerEvent e) {
    display.append(((Button)e.getChild()).getLabel()
        + " was"
        + action
        + e.getContainer().getClass().getName()
        + "\n");
}
...
}

```

La clase ContainerEvent

Cada método de evento de contenedor tiene un sólo parámetro: un objeto [ContainerEvent](#) La clase ContainerEvent define dos métodos útiles:

Component getChild()

Devuelve el componente cuya adición o eliminación disparo el evento.

Container getContainer()

Devuelve el contenedor que disparó el evento.

Escribir un Oyente de Foco

Los eventos de Foco se generan cada vez que un componente obtiene o pierde el foco del teclado -- la habilidad de recibir eventos del teclado. Desde el punto de vista del usuario, el componente con el foco del teclado normalmente es más prominente -- con un borde distinto del usual, por ejemplo -- y la ventana del componente también es más prominente que otras ventanas de la pantalla. Este permite al usuario saber a que componente irá todo lo que teclee. Casi la mayoría de los componentes del sistema de ventanas pueden tener el foco del teclado.

La forma exacta en la que los componentes obtienen el foco depende del sistema de ventanas. Normalmente, el usuario selecciona el foco pulsando sobre una ventana o componente, pulsando TAB entre componentes, o mediante cualquier otra interacción con un componente. Se puede pedir que un componente tenga el foco llamando al método `requestFocus` de `Component`

Método de Eventos de Foco

El interface [FocusListener](#) y su correspondiente clase adaptador, [FocusAdapter](#), contienen dos métodos:

`void focusGained(FocusEvent)`

Llamado por el AWT justo después de que el componente obtenga el foco.

`void focusLost(FocusEvent)`

Llamado por el AWT justo después de que el componente pierda el foco.

Ejemplos de Manejo de eventos de Foco

El siguiente applet demuestra los eventos de foco. Pulsando sobre el botón que hay en la parte superior del applet, podrás traer una ventana que contiene una gran variedad de componentes. Un oyente de Foco escucha los eventos de foco de cada componente de la ventana, incluyendo la propia ventana (que es un ejemplar de una subclase de `Frame` llamada `FocusWindow`).

Intenta esto:

1. Trae la ventana llamada `Focus Demo Window` pulsando sobre el botón de la parte superior del applet.
2. Si es necesario, pulsa sobre la ventana para que sus contenidos puedan obtener el foco.
Verás un mensaje de "Focus gained" en el área del applet. La forma de obtener el foco es dependiente del sistema. Puedes detectar cuando la ventana obtiene o

pierde el foco implementado un [oyente de ventana](#) y escuchar los eventos de activación o desactivación de ventana.

3. Pulsa el botón que hay a la derecha de la ventana Focus Demo Window, y luego pulsa sobre algún otro componente, como el campo de texto.
Observa que cuando el foco cambia de un componente a otro, el primer componente genera un evento de foco perdido antes de que el segundo componente genere un evento de foco obtenido.
 4. Intenta cambiar el foco pulsando Tab o Shift-Tab.
La mayoría de los sistemas permiten utilizar la tecla Tab para circular a través de los componentes que pueden obtener el foco.
 5. Miniaturiza la ventana Focus Demo Window.
Deberías ver un mensaje "Focus lost" del componente que tuvo el foco por última vez.
-

Puedes encontrar el código completo del applet en [FocusDemo.java](#). Aquí tienes el código de manejo de eventos del applet:

```
public class FocusDemo ... implements FocusListener ... {
    ...//Donde ocurra la inicialización
    window = new FocusWindow(this);

    ...
    public void focusGained(FocusEvent e) {
        displayMessage("Focus gained", e);
    }

    public void focusLost(FocusEvent e) {
        displayMessage("Focus lost", e);
    }

    void displayMessage(String prefix, FocusEvent e) {
        display.append(prefix
            + ": "
            + e.getSource() //XXX
            + "\n");
    }
    ...
}

class FocusWindow extends Frame {
    ...
}
```

```

public FocusWindow(FocusListener listener) {
    super("Focus Demo Window");
    this.addFocusListener(listener);
    ...
    Label label = new Label("A Label");
    label.addFocusListener(listener);
    ...
    Choice choice = new Choice();
    ...
    choice.addFocusListener(listener);
    ...
    Button button = new Button("A Button");
    button.addFocusListener(listener);
    ...
    List list = new List();
    ...
    list.addFocusListener(listener);
}
}

```

La clase FocusEvent

Cada método de evento de foco tiene un sólo parámetro: un objeto [FocusEvent](#). La clase FocusEvent define un método, isTemporary, que devuelve true si un evento de pérdida de foco es temporal. Es un conocimiento especial cuando deseas indicar que un componente particular obtenga el foco si la ventana vuelve a obtener el foco.

El mensaje más común que enviarás a un objeto FocusEvent es getComponent (definido en [ComponentEvent](#)), que devuelve el componente que acaba de obtener o perder el foco, disparando el evento.

Escribir un Oyente de Item

Los eventos de Ítem son generados por componentes que implementan el interface [ItemSelectable](#). Estos son componentes que mantienen el estado -- generalmente el estado on/off para uno o más ítems. Los componentes del AWT 1.1 que generan eventos de ítem son [checkboxes](#), [checkbox menu items](#), [choices](#), y [lists](#).

Métodos de Evento de Item

El interface [ItemListener](#) tiene sólo un método, y por lo tanto no tiene la correspondiente clase adaptador. Aquí tienes el método:

```
void itemStateChanged(ItemEvent)
```

Llamado por el AWT justo después de un cambio de estado en el componente escuchado.

La clase ItemEvent

Cada evento de ítem tiene un sólo parámetro: un objeto [ItemEvent](#). La clase ItemEvent define los siguientes métodos:

```
Object getItem()
```

Devuelve el objeto del componente específico con el ítem cuyo estado ha cambiado. Normalmente esta es una cadena que contiene el texto del ítem seleccionado. Otras posibilidades podría ser un objeto image o un objeto sin representación visual.

```
ItemSelectable getItemSelectable()
```

Devuelve el componente que generó el evento de ítem.

```
int getStateChange()
```

Devuelve el nuevo estado del ítem. La clase ItemEvent define dos estados: SELECTED y DESELECTED.

Escribir un Oyente de Teclas

Los eventos de Tecla te dicen cuando el usuario ha pulsado el teclado.

Específicamente, los eventos de tecla son generados por los componentes que tienen el foco del teclado cuando el usuario pulsa o libera una tecla. (Para más información sobre el foco, puedes ver la página: [Escribir un oyente de Foco](#).)

Se pueden notificar dos tipos básicos de eventos: el tecleo de un carácter Unicode, y la pulsación o liberación de una tecla del teclado. El primer tipo de evento es llamado un evento de tecla tecleada. El segundo tipo son los eventos tecla pulsada y tecla liberada.

En general, deberías intentar utilizar los eventos de tecla tecleada, a menos que necesites conocer cuando el usuario pulsa una tecla que no corresponde con los caracteres. Por ejemplo, si quieres saber cuando el usuario teclea algún carácter Unicode -- incluso aquellos que son el resultado de la pulsación de varias teclas por parte del usuario -- debes escuchar por los eventos de tecla tecleada. De otra forma, si quieres saber cuando el usuario pulsa el carácter F1, necesitarás escuchar los eventos de tecla pulsada/liberada.

Si necesitas detectar los eventos de tecla disparados por un componente de cliente, asegúrate de que el componente pide el foco del teclado. De otra forma, nunca obtendrá el foco y nunca disparará eventos de teclas.

Métodos de eventos de Tecla

El interface [KeyListener](#) y su correspondiente clase adaptador, [KeyAdapter](#), contienen tres métodos:

`void keyTyped(KeyEvent)`

Llamado por el AWT justo después de que el usuario teclee un carácter Unicode dentro del componente escuchado.

`void keyPressed(KeyEvent)`

Llamado por el AWT justo después de que el usuario pulse una tecla del teclado.

`void keyReleased(KeyEvent)`

Llamado por el AWT justo después de que el usuario libere una tecla del teclado.

Ejemplos de Manejo de Eventos de Tecla

El siguiente applet demuestra los eventos de tecla. Consiste en un campo de texto donde puedes teclear, seguido por un área de texto que muestra un mensaje cada vez que el campo de texto dispara un evento de tecla. Un botón en la parte inferior del applet te permite limpiar el campo y el área de texto.

Intenta esto:

1. Pulsa el campo de texto del applet para obtener el foco del teclado.
 2. Pulsa y libera la tecla A en el teclado.
El campo de texto dispara tres eventos: un evento de tecla pulsada, un evento de tecla tecleada y un evento de tecla liberada. Observa que el evento de tecla tecleada no tiene información sobre el código de tecla, tampoco tiene información sobre los modificadores. Si sólo te importan los caracteres que el usuario ha pulsado, deberías manejar los eventos de tecla tecleada. Si te importan las teclas que ha pulsado/liberado el usuario deberías manejar los eventos de tecla pulsada/liberada.
 3. Pulsa el botón Clear.
Porías querer hacer esto después de cada uno de los siguientes pasos
 4. Pulsa y libera la tecla Shift.
El campo de texto dispara dos eventos: uno de tecla pulsada y otro de tecla liberada. El campo de texto no genera un evento de tecla tecleada porque Shift, por sí misma, no corresponde a ningún carácter.
 5. Pulsa una A mayúscula pulsando las teclas Shift y A.
Verás los siguientes eventos, aunque quizás no en este orden: tecla pulsada (Shift), tecla pulsada (A), tecla tecleada ('A'), tecla liberada (A) y tecla liberada (Shift).
 6. Teclea una A mayúsculas pulsando y liberando la tecla Caps Lock, y pulsando la tecla A.
Deberías ver los siguientes eventos: tecla pulsada (Caps Lock), tecla pulsada (A), tecla tecleada ('A'), tecla liberada (A). Observa que la tecla Caps Lock no genera un evento de tecla liberada hasta que la pulses y la sueltes otra vez. Esto es igual para otras teclas de estado como Scroll Lock o Num Lock.
 7. Pulsa y manten la tecla A.
¿Se repite automáticamente? Si es así, verás los mismos eventos que si pulsaras y liberaras la tecla A repetidamente.
-

Puedes encontrar el código completo del Applet en [KeyDemo.java](#). Aquí tienes el código de manejo de eventos del applet:

```
public class KeyDemo ... implements KeyListener ... {
    ...//Donde ocurra la inicialización:
        typingArea = new TextField(20);
        typingArea.addKeyListener(this);
    ...
    /** Maneja el evento de tecla tecleada del campo de texto. */
```

```

    public void keyTyped(KeyEvent e) {
        displayInfo(e, "KEY TYPED: ");
    }

    /** Maneja el evento de tecla pulsada del campo de texto. */
    public void keyPressed(KeyEvent e) {
        displayInfo(e, "KEY PRESSED: ");
    }

    /** Maneja el evento de tecla liberada del campo de texto. */
    public void keyReleased(KeyEvent e) {
        displayInfo(e, "KEY RELEASED: ");
    }
    ...
    protected void displayInfo(KeyEvent e, String s){
        ...
        char c = e.getKeyChar();
        int keyCode = e.getKeyCode();
        int modifiers = e.getModifiers();
        ...
        tmpString = KeyEvent.getKeyModifiersText(modifiers);

        ...//muestra información sobre el evento...
    }
}

```

La clase KeyEvent

Cada método de evento de tecla tiene un sólo parámetro: un objeto [KeyEvent](#). La clase KeyEvent define los siguientes métodos:

int getKeyChar()

void setKeyChar(char)

Obtiene o selecciona el carácter de tecla asociado con el evento de tecla tecleada. El carácter es un carácter Unicode.

int getKeyCode()

void setKeyCode(int)

Obtiene o selecciona el código de tecla asociado con este evento. El código de tecla identifica una tecla particular del teclado que el usuario ha pulsado o liberado. La clase KeyEvent define muchas constantes para códigos de tecla. Por ejemplo, VK_A especifica la tecla etiquetada A, y VK_ESCAPE especifica la tecla ESCAPE.

void setModifiers(int)

Selecciona el estado de las teclas modificadoras para este evento. También puedes ver los métodos getModifiers de InputEvent.

Otros métodos potencialmente útiles de KeyEvent incluyen métodos que

generan descripciones de texto localizables de códigos de teclas y teclas modificadoras.

La clase KeyEvent descende de [InputEvent](#), que a su vez descende de [ComponentEvent](#). ComponentEvent proporciona el método getComponent. InputEvent proporciona los siguientes métodos:

int getWhen()

Devuelve el momento en el que ocurrió este evento. Cuanto más alto sea el tiempo, más recientemente ha ocurrido el evento.

boolean isAltDown()

boolean isControlDown()

boolean isMetaDown()

boolean isShiftDown()

Estos métodos ofrecen el estado de las teclas modificadoras cuando el evento fue generado.

int getModifiers()

Devuelve una bandera que indica el estado de todas las banderas cuando el evento fue generado.

Escribir un Oyente de Ratón

Los eventos de Ratón te dicen cuando el usuario utiliza el ratón (u otro dispositivo de entrada similar) para interactuar con un componente. Los eventos de ratón ocurren cuando el cursor entra o sale del área de pantalla de un componente o cuando el usuario presiona o libera un botón del ratón. Como seguir la pista de los movimientos del cursor envuelve un sobrecarga del sistema más significativa que los eventos del ratón, los eventos de movimiento se han separado en otro tipo de oyente. (Puedes ver [Escribir un Oyente de Movimientos del Ratón](#)).

Métodos de Eventos del Ratón

El interface [MouseListener](#) y su correspondiente clase adaptador, [MouseAdapter](#), contienen tres métodos:

`void mouseClicked(MouseEvent)`

Llamado por el AWT justo después de que el usuario pulse sobre el componente escuchado.

`void mouseEntered(MouseEvent)`

Llamado por el AWT justo después de que el cursor entre en los límites del componente escuchado, si no está pulsado el botón del ratón.

`void mouseExited(MouseEvent)`

Llamado por el AWT justo después de que el cursor abandone los límites del componente escuchado, si no está pulsado el botón del ratón.

`void mousePressed(MouseEvent)`

Llamado por el AWT justo después de que el usuario pulse un botón del ratón mientras el cursor está sobre el componente escuchado.

`void mouseReleased(MouseEvent)`

Llamado por el AWT justo después de que el usuario libere un botón del ratón. El evento de liberación de ratón es disparado por el mismo componente que acaba de generar el evento de ratón pulsado, no importa donde esté el cursor cuando ocurra la liberación del botón.

Ejemplos de Manejo de Eventos de Ratón

El siguiente applet demuestra los eventos del ratón. En la parte superior del applet hay un área vacía (implementada por una clase llamada `BlankArea`). Un oyente de ratón escucha los eventos tanto del `BlankArea` como de su propio contenedor, que es un ejemplar de `MouseDemo`. Cada vez que ocurre un evento de ratón, se muestra un mensaje descriptivo debajo del área vacía. Moviendo el cursor sobre el área vacía y pulsando los botones del ratón podrás ver los eventos

generados por el ratón.

Intenta esto:

1. Mueve el cursor dentro del rectángulo que hay en la parte superior del applet.
Verás uno o más eventos de "mouse enter" [que deberían ser mostrador como "Cursor enter"].
2. Pulsa y manten el botón del ratón.
Verás un evento de "mouse press". Podrías ver algunos eventos extras como "mouse exit" o "mouse enter".
3. Libera el botón del ratón.
Verás un evento de "mouse release". Si no has movido el ratón seguirá un evento de "mouse click".
4. Pulsa y manten el botón del ratón y luego arrastra el ratón hasta el exterior del área del applet. Libera el botón del ratón.
Verás un evento de "mouse press", seguido por un evento de "mouse exit" y seguido por un evento de "mouse release". No has sido notificado sobre el movimiento del cursor. Para obtener los movimientos del ratón, necesitas implementar un [Oyente de movimiento de ratón](#).

Aquí podrás encontrar el código completo del applet [MouseDemo.java](#). Y aquí tienes el código de manejo de eventos del applet:

```
public class MouseDemo ... implements MouseListener {
    ...//Donde ocurra la inicialización:
    //Registrado para los eventos del ratón en el blankArea
        //y en el applet (panel).
    blankArea.addMouseListener(this);
    addMouseListener(this);
    ...

    public void mousePressed(MouseEvent e) {
        saySomething("Mouse button press", e);
    }

    public void mouseReleased(MouseEvent e) {
        saySomething("Mouse button release", e);
    }

    public void mouseEntered(MouseEvent e) {
        saySomething("Cursor enter", e);
    }
}
```

```

        public void mouseExited(MouseEvent e) {
            saySomething("Cursor exit", e);
        }

        public void mouseClicked(MouseEvent e) {
            saySomething("Mouse button click", e);
        }

        void saySomething(String eventDescription, MouseEvent e) {
            textArea.append(eventDescription + " detected on "
                            + e.getComponent().getClass().getName()
                            + ".\n");
            textArea.setCaretPosition(maxInt); //scroll to bottom
        }
    }

```

La clase MouseEvent

Cada método de evento de ratón tiene un sólo parámetro: un objeto [MouseEvent](#) La clase MouseEvent define los siguientes métodos:

int getClickCount()

Devuelve el número de rápidos, clicks consecutivos que ha realizado el usuario (incluido este evento).

int getX()

int getY()

Point getPoint()

Devuelve la posición (x,y) del cursor cuando ocurrió el evento, relativo al componente sobre el que ocurrió el evento.

boolean isPopupTrigger()

Devuelve true si el evento debería causar que apareciera un menú popup. Como los disparos de popup son dependientes de la plataforma, si tu programa utiliza menús popup, deberías llamar a isPopupTrigger para los eventos de "mouse down" y "mouse up".

La clase MouseEvent desciende de [InputEvent](#), que desciende de [ComponentEvent](#). ComponentEvent proporciona el método getComponent. InputEvent proporciona los siguientes métodos:

int getWhen()

Devuelve el momento en el que ocurrió el evento. Un mayor número aquí indica que el evento ha ocurrido más recientemente.

boolean isAltDown()

boolean isControlDown()

boolean isMetaDown()

boolean isShiftDown()

Estos métodos te ofrecen el estado de las teclas modificadoras cuando el evento fue generado.

`int getModifiers()`

Devuelve una bandera indicando el estado de todas las teclas modificadoras cuando el evento fue generado.

Escribir un Oyente de Movimiento del Ratón

Los eventos de movimiento del ratón te dicen cuando el usuario utiliza un ratón (o un dispositivo de entrada similar) para mover el cursor por la pantalla. Para más información sobre otro tipos de eventos del ratón, como los clicks, puedes ver la página, [Escribir un Oyente del Ratón](#).

Métodos de Eventos de Movimiento del Ratón

El interface [MouseMotionListener](#) y su correspondiente clase adaptador, [MouseMotionAdapter](#) contienen dos métodos:

`void mouseDragged(MouseEvent)`

Llamado por el AWT en respuesta a un movimiento del ratón por parte del usuario mientras está pulsado un botón. Este evento es disparado por el componente que disparó el evento de ratón pulsado precedente.

`void mouseMoved(MouseEvent)`

Llamado por el AWT en respuesta a un movimiento del ratón por parte del usuario sin que esté pulsado el botón. Este evento es disparado por el componente sobre el que se encuentra el cursor.

La clase MouseEvent

Cada método de evento de movimiento de ratón tiene un sólo parámetro -- y no se llama `MouseMotionEvent`!. En su lugar se utilizan objetos [MouseEvent](#). Puedes ver [Escribir un Oyente de Ratón](#) para más información sobre la clase `MouseEvent`.

Escribir un Oyente de Texto

Los eventos de texto son generados después de que el texto de un componente de texto haya cambiado de algún modo. Los componentes del AWT 1.1 que pueden generar eventos de texto son los [campos y las áreas de texto](#). Para obtener una notificación anterior a los cambios de texto -- por ejemplo, para interceptar caracteres incorrectos -- deberías escribir un [oyente de teclas](#).

Métodos de eventos de Texto

El interface [TextListener](#) tiene sólo un método, por lo tanto no tiene la correspondiente clase adaptador. Aquí tienes el método:

```
void textValueChanged(TextEvent)
```

Llamado por el AWT justo después de que haya cambiado el texto del componente escuchado.

Ejemplos de Manejo de Eventos de Texto

El siguiente applet demuestra los eventos de texto. Contiene dos componentes de texto editable -- un campo de texto y un área de texto. Pulsar Return en el campo de texto hace que el contenido de éste sea añadido al área de texto. Cada componente de texto editable posee un oyente de texto. Estos dos oyentes, que son dos ejemplares de una misma clase, añaden un mensaje a un área de texto no editable que está a la derecha del applet. Un botón situado en la parte inferior derecha del applet te permite limpiar el área de mensajes.

Intenta esto:

1. Pulsa sobre el campo de texto situado en la parte superior izquierda del applet, luego pulsa el carácter A en el teclado. Ocurre un evento de texto, y verás un mensaje en el área situada a la derecha del applet.
 2. Teclea algunos caracteres más. Ocurre un evento de texto cada vez que tecleas un carácter.
 3. Pulsa Return. El campo de texto no genera un evento de texto. En su lugar genera un evento "action", y el manejador de este evento copia el texto contenido en el campo de texto dentro del área de texto. El área de texto reacciona generando un sólo evento de texto, sin importar el número de caracteres copiado.
 4. Pulsa sobre el área de texto -- el área grande situado en la parte inferior izquierda del applet -- y luego pulsa un carácter en el teclado. El área de texto dispara un evento de texto.
-

Aquí podrás encontrar el código completo del applet [TextDemo.java](#). Sucede que este applet implementa su manejo de eventos de texto dentro de una clase interna llamada MyTextListener. El applet crea y registra dos ejemplares de

MyTextListener, uno para cada componente de texto editable. El constructor de MyTextListener toma una cadena que describe la fuente del evento. Cuando un ejemplar de MyTextListener detecta un evento de texto, (esto es, se llama al método `textValueChanged` de MyTextListener), el ejemplar añade un mensaje al área situada a la derecha del applet. Aquí tienes el código de manejo de eventos del applet:

```
public class TextDemo ... {
    TextField textField;
    TextArea textArea;
    TextArea displayArea;
    ...
    //Donde ocurra la inicialización:
    textField = new TextField(20);
    ...
    textField.addTextListener(new MyTextListener("Text Field"));

    textArea = new TextArea(5, 20);
    textArea.addTextListener(new MyTextListener("Text Area"));
    ...
}

class MyTextListener implements TextListener {
    String preface;

    public MyTextListener(String source) {
        preface = source
            + " text value changed.\n"
            + "    First 10 characters: \";
    }

    public void textValueChanged(TextEvent e) {
        TextComponent tc = (TextComponent)e.getSource();
        String s = tc.getText();
        ...//truncate s to 10 characters...

        displayArea.append(preface + s + "\"\n");
        ...
    }
}
...
}
```

La clase `TextEvent`

Cada método de evento de texto tiene un sólo parámetro: un objeto [TextEvent](#). La clase `TextEvent` no define métodos útiles. Utilizando el método `getSource` que hereda de `EventObject`, puedes obtener el componente que generó el evento y luego enviarle un mensaje.

Escribir un Oyente de Ventana

Los eventos de ventana son generados por una Ventana justo después de que la ventana sea abierta, cerrada, iconificada, desiconificada, activada o desactivada. Abrir una ventana significa hacerla visible en la pantalla; cerrarla significa eliminar la ventana de la pantalla. Iconificarla significa sustituir la ventana por un pequeño icono en el escritorio, desiconificarla significa lo contrario. Una ventana es activada si uno de sus componentes obtiene el foco del teclado; la desactivación ocurre cuando la ventana o uno de sus componente pierdan el foco del teclado.

Probablemente el uso más común de los oyentes de ventana es para cerrar ventanas. Si un programa no maneja los eventos de cerrar ventana, no sucederá nada cuando el usuario intente cerrarla. Una aplicación que consiste en una sola ventana podría reaccionar a los eventos de cerrar ventana saliendo de la aplicación. Un applet u otro programa en el que existan más de una ventana normalmente llama al método `dispose` para cerrar la ventana.

Otro uso común de los oyentes de ventanas es parar los threads y liberar los recursos cuando una ventana es iconificada, y para arrancar de nuevo cuando la ventana se agranda de nuevo. De esta forma, podrás evitar la utilización innecesaria del procesador u otros recursos. Por ejemplo, un programa que realiza continuas animaciones, no es útil cuando la ventana no es visible, por eso debería liberar los recursos del sistema cuando está iconificada. Específicamente, debería parar el thread de animación y liberar cualquier gran buffer cuando la ventana es iconificada, y arrancar el thread de nuevo y recrear los buffers cuando la ventana se desiconificada.

Métodos de Eventos de Ventana

El interface [WindowListener](#) y su correspondiente clase adaptador, [WindowAdapter](#), contienen siete métodos:

`void windowOpened(WindowEvent)`

Llamado por el AWT justo después de que la ventana escuchada haya sido abierta por primera vez [o mostrada después de haber sido cerrada].

`void windowClosing(WindowEvent)`

Llamado por el AWT en respuesta a una petición del usuario de que la ventana escuchada sea cerrada. Para cerrar realmente la ventana, el oyente debe llama al método `dispose` de la ventana.

`void windowClosed(WindowEvent)`

Llamada por el AWT justo después de que la ventana escuchada haya sido cerrada.

`void windowIconified(WindowEvent)`

`void windowDeiconified(WindowEvent)`

Llamados justo después de que la ventana escuchada haya sido iconificada o desiconificada, respectivamente.

`void windowActivated(WindowEvent)`

`void windowDeactivated(WindowEvent)`

Llamada por el AWT justo después de que la ventana escuchada sea activada o desactivada, respectivamente.

La clase `WindowEvent`

Cada método de evento de ventana tiene un sólo parámetro: un objeto [WindowEvent](#). La clase `WindowEvent` define un método útil, `getWindow`, que devuelve la ventana que generó el evento.

Utilizar la versión "Swing" del JFC

El proyecto Swing -- que está desarrollando algo del API de JFC (Java Foundation Classes) -- ha desarrollado muchos componentes listos para utilizar y otras clases convenientes. Esta sección te cuenta cómo utilizar los componentes Swing. Pronto podrás ver como crear los tuyos propios.

Este es un Trabajo en Progreso!

Las páginas de esta sección se actualizan constantemente. Date una vuelta de vez en cuando por aquí o por la [Home page del JFC](#) para obtener la última información sobre el proyecto Swing. Esta sección se actualizó por última vez en **8 de Agosto de 1997**.

[Introducción a los Componentes de Swing](#)

Esta sección te muestra todos los componentes de Swing y luego enlaza con la dirección donde podrás aprender a utilizar los componentes.

[Empezar con Swing](#)

Para escribir programas que utilicen componentes Swing, primero debes descargar la última versión de Swing. Para asegurarte de que has bajado la versión correcta y que tu entorno está configurado correctamente, puedes ejecutar la demo SwingSet. La prueba final es compilar y ejecutar programas Swing.

[Utilizar los Componentes Swing](#)

Los componentes de Swing conforman la arquitectura de Swing, lo que significa que son de peso ligero, tiene con carácter conectable, de acuerdo a los requerimientos de los JavaBeans, etc.

Esta sección primero ofrece una introducción al código que podrías utilizar cuando utilices componentes Swing. Luego cada componente tiene una subsección que explica todo sobre la utilización del componente.

Escribir Componentes de Peso Ligero utilizando la versión de Swing

Esta sección te contará como escribir dos clases de componentes de peso ligero:

- Componentes sencillos, utilizando subclases de componentes Swing. (Así es fácil!)
 - Componentes que se aprovechan de la arquitectura de Swing.
-

Introducción a los Componentes de Swing

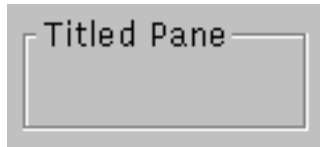
Esta página muestra los componentes Swing, con enlaces a las páginas donde cada componente se describe en más detalle. Para tener una los componentes listos para utilizar que se han añadido a Swing, puedes comparar esta sección con las paginas de [Componentes del AWT](#).

[Label](#)

[Bordered pane](#)

[Progress bar](#)

[Tool tip](#)

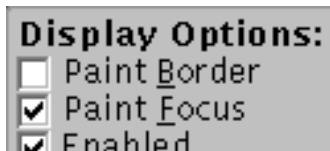
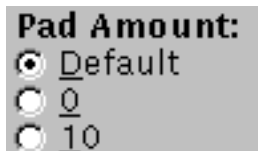


[Button](#)

[Radio buttons](#)

[Checkboxes](#)

[Tool bar](#)

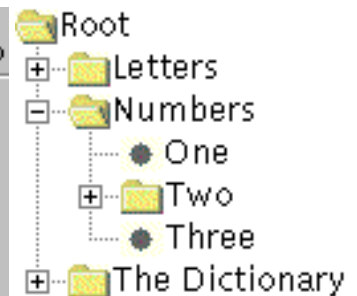
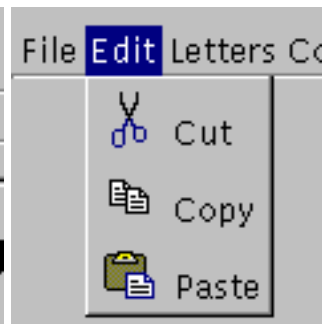
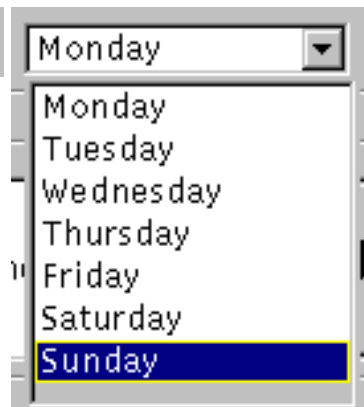


[Slider](#)

[Combo box](#)

[Menu](#)

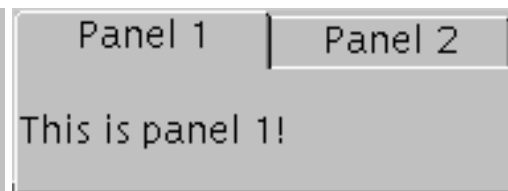
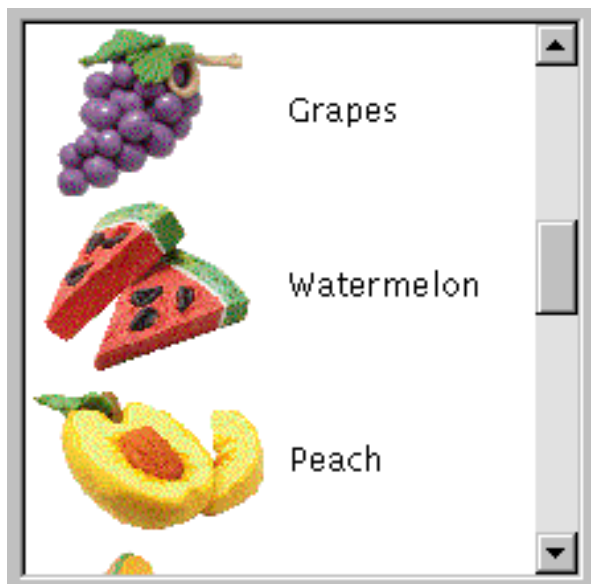
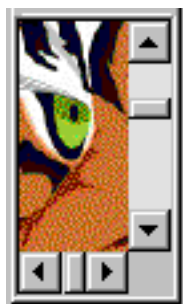
[Tree](#)



[Scroll bar](#)

[List box](#)

[Tabbed pane](#)



[Table](#)

First Name	Last Name	Favorite Color	Favorite Number	Vegetarian
Tim	Prinzing	Blue	22	<input type="checkbox"/>
Chester	Rose	Black	0	<input type="checkbox"/>
Ray	Ryan	Gray	77	<input type="checkbox"/>
Georges	Saab	Red	4	<input type="checkbox"/>
Kathy	Walrath	Blue	8	<input type="checkbox"/>
Arnaud	Weber	Green	44	<input type="checkbox"/>

Ozito

Empezar con Swing

Para empezar a utilizar el entorno de programación de Swing, debes seguir estos pasos:

- [Descargar la última versión del JDK](#), si no lo has hecho todavía.
- [Descargar la última versión de Swing](#), si no lo has hecho todavía.
- [Ejecutar la aplicaciónRun Swing](#).
- [Compilar y ejecutar una aplicación Swing](#).

Incluso puedes ejecutar [applet Swing](#), si estas dispuesto a seguir un par de atajos.

Descargar la última versión del JDK

La versión de Swing requiere la corrección de algunos errores que aparecieron en el JDK 1.1.2. Deberías [descargar la última versión del JDK](#).

Descargar la última versión de Swing

Puedes [Descargar la versión de Swing](#) desde la "Java Developers Connection". Necesitarás registrarte, si aún no lo has hecho.

Ejecutar una aplicación Swing

Antes de escribir tus propios programas Swing, deberás asegurarte de que puedes ejecutar una ya existente. Podrías querer empezar ejecutando el ejemplo SwingSet, que te presenta todos los componentes de Swing. El fichero README.txt situado en el directorio superior de la versión de Swing te explica como ejecutar la demo SeingSet.

Aquí tienes una explicación general de como ejecutar aplicaciones Swing:

1. Toma nota de donde tienes instalado la copia de la versión de Swing. Necesitarás esto para poder encontrar las clases de Swing. Podrías querer seleccionar la [variable de entorno](#) SWINGHOME con el directorio superior de la versión de Swing.

El fichero que contiene las clases Swing está en el directorio de nivel más alto en un fichero llamado swing.jar. Los distintos ficheros Rosas están archivados en el mismo directorio, en un fichero llamado rose.jar.

Nota: Ten cuidado de no eliminar estos ficheros. La versión de Swing depende de estos ficheros.

2. Toma nota de donde tienes instalada la copia de la versión del JDK

1.1.N (donde $N \geq 2$). Necesitarás esto para poder encontrar las versiones apropiadas de las clases del JDK y el intérprete. Podrías querer seleccionar la [variable de entorno](#) JDKHOME con el directorio superior de la versión del JDK 1.1.N.

Las clases del JDK están en el directorio lib de la versión del JDK, en un fichero llamado classes.zip. ¡No descomprimas este fichero! El intérprete Java está en el directorio bin de la versión del JDK.

3. [Ejecutar la aplicación](#), especificando un classpath que incluya el fichero swing.jar, el fichero classes.zip del JDK, y el directorio que contiene las clases del programa. Asegúrate de que las clases y el intérprete del JDK que estás utilizando son de la misma versión del JDK 1.1.N.!

Compilar y ejecutar una aplicación Swing

1. Elige un directorio de trabajo.
2. Graba el fichero fuente Java en tu directorio de trabajo. Aquí tienes uno que puedes utilizar: [ButtonDemo.java](#).
3. Desde tu directorio de trabajo, [llama al compilador Java](#), especificando el classpath como se describió en el paso 3 de la sección anterior.
4. Desde tu directorio de trabajo, llama al intérprete Java, como se describió en el paso 3 de la sección anterior.

Si has seguido todo esto, estás listo para empezar a escribir tus propios programas swing! Puedes aprender cosas sobre Swing a través de las páginas de este tutorial o a través de la documentación de la versión de swing. Para leer la documentación de Swing, utiliza un navegador para ver el fichero doc/index.html (que está en el directorio superior de la versión de Swing).

Ejecutar un Applet Swing

Debido a las restricciones de seguridad, actualmente no puedes distribuir clases Swing en un Applet. Sin embargo, si las clases de Swing están en tu path, algunas veces podrás ejecutar applets basados en swing en navegadores 1.1 que estén escritos en Java. Los applets tienden a cascar después de un rato. Obviamente se está planeando corregir esto.

¿Puedes ver tres botones debajo de este párrafo?. Si es así, estás ejecutado con éxito un applet basado en Swing!

El resto de esta página ofrece instrucciones paso a paso para ejecutar applets basados en Swing y explica las limitaciones de estos applets.

Paso a paso: Ejecutar un applet basado en Swing.

1. Encuentra un navegador 1.1 que esté escrito en Java. Asegurate de que tienes la última versión, ya que las versiones posteriores tienden a corregir errores que hacen que Swing trabaje mejor. Los navegadores Java 1.1 son [HotJava](#) y el (appletviewer), que se distribuye en el [JDK](#),
2. Asegurate que las clases Swing están el [path de clases de tu navegador](#).
3. Arranca tu navegador Java y apunta a una página como ésta que contenga un applet basado en Swing.

Limitaciones de los Applets basados en Swing

Los applets de Swing generan excepciones de seguridad. Mientras el applet trabaje correctamente, puedes ignorar estas excepciones.

Quizás no podrás utilizar imágenes en los applets basados en Swing. Por ejemplo, el applet de esta página está basado en la aplicación explicada en la página [la clase Button](#). Esta aplicación utiliza imágenes en botones. Hemos tenido que eliminar las imágenes de los botones para conseguir que el código se ejecute en un applet.

Utilizar todos los Componentes de Swing

[Reglas generales para el uso de componentes Swing](#)

Antes de empezar a utilizar los componentes Swing, deberías saber las clases que proporciona el JComponent

Como se utiliza ...

El siguiente grupo de páginas te explica como utilizar los componentes Swing. Cada tipo de componente tiene su propia página:

- [Cómo utilizar Button](#)
- [Cómo utilizar Checkbox](#)
- Cómo utilizar Canvas
- Cómo utilizar Choice
- Cómo utilizar Color Chooser [not implemented yet?]
- Cómo utilizar Combo Boxe
- Cómo utilizar File Chooser [not implemented yet?]
- Cómo utilizar Font Chooser [not implemented yet?]
- Cómo utilizar Internal Frames
- [Cómo utilizar Label](#)
- Cómo utilizar Layered Pane
- Cómo utilizar List Boxe
- Cómo utilizar Menu
- Cómo utilizar Panel
- Cómo utilizar Popup Menu
- Cómo utilizar Progress Bar
- [Cómo utilizar Radio Button](#)
- Cómo utilizar Scroll Bar y Pane
- Cómo utilizar Separator
- Cómo utilizar Slider
- [Cómo utilizar Tabbed Pane](#)
- [Cómo utilizar Table](#)
- Cómo utilizar Text
- Cómo utilizar Titled Pane [serán renombrados como Bordered Pane]
- Cómo utilizar Tool Bar
- [Cómo utilizar Tool Tip](#)

- [Cómo utilizar Tree](#)
 - Cómo utilizar Viewport
-

Ozito

Reglas Generales para el uso de Componentes Swing

Esta sección tiene información general sobre lo que tienen en común los componentes de swing. Explica las características que proporciona la clase `JComponent` a la mayoría de los componentes Swing, incluyendo código sobre cómo utilizar estas características. Un día, quizás podríamos mostrarte como se diferencia el código de programas Swing de los programas que utilizan componentes de peso pesado.

Qué proporciona la clase `JComponent`

La mayoría de los componentes Swing están implementados como una subclase de la clase [JComponent](#), que descende de la clase `Component`. De `JComponent`, los componentes Swing heredan las siguientes funcionalidades:

- [Tool tips](#)
- [acciones generados por teclado](#)
- [Soporte para escrolado](#)
- [Propiedades](#)
- [Aplicación ampliamente configurable](#)
- [Soporte para distribución](#)

Tool tips.

Especificando una cadena con el método `setToolTip`, puedes proporcionar ayuda a los usuarios de un componente. Cuando el cursor se pare sobre el componente, la cadena especificada se mostrará en una pequeña ventana que aparece cerca del componente. Puedes ver [Cómo utilizar Tool Tips](#) para más información.

Acciones generadas por teclado.

Utilizando el método `registerKeyboardAction`, puedes permitir que el usuario utilice el teclado, en vez del ratón, para indicar qué acción del componente debería ocurrir.

Nota: Algunas clases proporcionan métodos convenientes para acciones de teclado. Por ejemplo `AbstractButton` proporciona `setKeyAccelerator`, para especificar los caracteres que, en combinación con una tecla modificadora especificada, hace que la acción del botón sea realizada. Puedes ver [Cómo utilizar Buttons](#) para ver un ejemplo de utilización de teclas aceleradoras en

botones.

La combinación de caracteres y teclas modificadoras que el usuario debe pulsar para arrancar una acción está representada por un objeto `JKeyStroke`. El evento `action` resultante es manejado por un objeto `JAction`. Toda acción de teclado funciona bajo una de estas dos condiciones: cuando el componente actual tiene el foco o cuando cualquier componente de su ventana tiene el foco.

Soporte para scrolling.

Con los métodos `computeVisibleRect` y `getVisibleRect`, puedes determinar que parte de un componente podría ser visible en la pantalla. Esto es especialmente útil para aumentar el rendimiento del escrolado. El método `scrollRectToVisible` envía un mensaje al padre del `JComponent`, pidiendo que muestre el área especificada. Con el método `setAutoscrolls`, puedes especificar que un componente se escole automáticamente cuando sea arrastrado, si su contenedor soporta escrolado (como un `JViewport`).

Propiedades.

Con el método `putProperty`, puedes asociar una o más propiedades (parejas nombre/objeto) con cualquier `JComponent`. Por ejemplo, un controlador de disposición podría utilizar las propiedades para asociar un objeto con cada `JComponent` que maneja. Puedes obtener propiedades utilizando el método `getProperty`.

Aplicación ampliamente configurable.

Cada máquina virtual Java tiene un objeto `UIFactory` que determina el aspecto y comportamiento de los componentes Swing en tiempo de ejecución. Sujeto a las restricciones de seguridad puedes elegir el aspecto y comportamiento de todos los componentes Swing, llamando al método `UIManager.setUIFactory`. Detrás de la escena, cada objeto `JComponent` tiene su correspondiente objeto `ComponentUI` (creado por el `UIFactory`) que realiza las acciones de dibujo, manejo de eventos, determinación de tamaño, etc para cada `JComponent`.

Nota: Si quieres cambiar el aspecto y comportamiento de un componente individual de Swing, en vez de cambiar todo el conjunto completo de componentes, deberías reimplementar el componente individual.

Soporte para Distribución.

Con métodos como `setPreferredSize`, `setMinimumSize`, `setMaximumSize`, `setAlignmentX`, `setAlignmentY`, y `setInsets`, puedes especificar restricciones de distribución sin tener

que escribir tu propio componente.

Consideraciones para la utilización de componentes de peso ligero

Para evitar el "parpadeo" cuando se redibujan componentes de peso ligero, necesitarás poner los componentes de peso ligero en contenedores con doble buffer. La versión de Swing incluye dos clases que puedes utilizar juntas para obtener el doble buffer: JPanel y JFrame. La clase JPanel proporciona contenedores de doble buffer y de peso ligero. La clase JFrame que descende de Frame simplemente implementa una corrección de dibujo para permitir que JPanel funcione. Para más información sobre el parpadeo puedes ir a [Eliminar el Parpadeo](#).

Cómo utilizar Buttons

Para crear un botón, puedes ejemplarizar uan de las muchas sublcases de la clase [AbstractButton](#). Esta sección explica el API basico de botones que define `AbstractButton` -- y que tienen en común todos los botones Swing. Como `JButton` subclasifica `AbstractButton` no define un nuevo API público, esta página lo utiliza para mostrar como funcionan los botones. La siguiente tabla muestra todas las subclases de `AbstractButton` definidas en Swing, que podrías querer ejemplarizar:

Clase	Sumario	Dónde se describe
JButton	Un botón normal.	Esta sección.
JCheckbox	Un checkbox típico.	Cómo utilizar Checkboxes
JRadioButton	Uno de los grupos de botones de radio.	Cómo utilizar Radio Buttons
JMenuItem	Un ítem en un menú.	[todavía en ningún sitio]
JMenu	Un menú.	[todavía en ningún sitio]

Aquí tienes un gráfico de una aplicación que muestra tres botones:



- Try this:
1. Compila y ejecuta la aplicacion. El fichero fuente es [ButtonDemo.java](#). Puedes ver [Empezar con Swing](#) si necesitas ayuda
 2. Pulsa el botón de la izquierda.
Desactiva el botón centarl (y a sí mismo, ya que no es necesario) y activa el botón de la derecha.
 3. Pulsa el botón de la derecha.
Activa el botón central y el de la izquierda y se desactiva a sí mismo.

Cómo muestra el ejemplo `ButtonDemo`, un botón Swing puede mostrar tanto texto como una imagen. En `ButtonDemo`, cada botón tiene su texto en un lugar diferente, relativo a su imagen. La letra subrayada del texto de cada botón muestra la tecla alternativa para cada botón.

Caundo un botón se desariva, automáticamente se genera una apariencia de botón desactivado. Sin embargo, se podría proporcionar una imagen que sustituyera a la imagen normal. Por ejemplo, podrías proporcionar un versión gris de las imágenes utilizadas en los botoes derecho e izquierdo.

Cómo se implementa el manejo de eventos depende del tipo de botón y de su utilización. Todos los botones Swing, pueden generar evento `action`, `change` e `ítem`, así como los eventos usuales de bajo nivel. El ejemplo `ButtonDemo` implementa sólo el oyente de `action`.

Abajo está el código de [ButtonDemo.java](#) para crear los botones del ejemplo anterior y reacciona a los clicks en los botones. El código en **negrita** es el código que permanecería si los botones no tuvieran imágenes.

```
//En el código de inicialización:
ImageIcon leftButtonIcon = new ImageIcon("LEFT.gif");
ImageIcon middleButtonIcon = new ImageIcon("middle.gif");
ImageIcon LEFTButtonIcon = new ImageIcon("left.gif");

b1 = new JButton("Disable middle button", leftButtonIcon);
b1.setVerticalTextPosition(AbstractButton.CENTER);
b1.setHorizontalTextPosition(AbstractButton.LEFT);
b1.setKeyAccelerator('d');
b1.setActionCommand("disable");

b2 = new JButton("Middle button", middleButtonIcon);
b2.setVerticalTextPosition(AbstractButton.BOTTOM);
b2.setHorizontalTextPosition(AbstractButton.CENTER);
b2.setKeyAccelerator('m');

b3 = new JButton("Enable middle button", LEFTButtonIcon);
//Use the default text position of CENTER, LEFT.
b3.setKeyAccelerator('e');
b3.setActionCommand("enable");
b3.setEnabled(false);

//Oyente para actions en los botones 1 y 3.
b1.addActionListener(this);
b3.addActionListener(this);
. . .
}

public void actionPerformed(java.awt.event.ActionEvent e) {
    if (e.getActionCommand().equals("disable")) {
        b2.setEnabled(false);
        b1.setEnabled(false);
        b3.setEnabled(true);
    } else {
        b2.setEnabled(true);
        b1.setEnabled(true);
        b3.setEnabled(false);
    }
}
```

La tabla siguiente lista los métodos utilizados de AbstractButton y los constructores de JButton. Podrás ver más API en acción jugando con los botones del ejemplo SwingSet que es parte de la versión de Swing..

El API de utilización de botones se divide en tres categorías:

- [Seleccionar u Obtener los Contenidos del Botón](#)
- [Ajuste Fino de la Apariencia del Botón](#)
- [Implementar la Funcionalidad del Botón](#)

Seleccionar u Obtener los Contenidos de un Botón

Método o Constructor	Propósito	Ejemplo
JButton(String, Icon) JButton(String) JButton(Icon) JButton()	Crear un ejemplar de JButton, la inicialización debe especificar el texto o la imagen	ButtonDemo.java
void setText(String) String getText()	Selecciona u obtiene el texto mostrado por el botón.	[todavía no]
void setIcon(Icon) Icon getIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando el botón no está seleccionado o pulsado.	[todavía no]
void setDisabledIcon(Icon) Icon getDisabledIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando está desactivado. Si no especificas ninguno, se crea uno por defecto manipulando la imagen.	[todavía no]
void setPressedIcon(Icon) Icon getPressedIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando es pulsado.	[todavía no]
void setSelectedIcon(Icon) Icon getSelectedIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando está seleccionado.	[todavía no]
void setRolloverIcon(Icon) Icon getRolloverIcon()	Selecciona u obtiene la imagen mostrada por el botón cuando el cursor pasa sobre él sin que esté pulsado.	[característica no implementada todavía]

Ajuste Fino de la Apariencia del Botón

Método o Constructor	Propósito	Ejemplo
void setHorizontalAlignment(int) void setVerticalAlignment(int) int getHorizontalAlignment() int getVerticalAlignment()	Seleccionan u obtienen donde se deben situar los contenidos del botón. La clase AbstractButton define tres posibles valores para alineación horizontal: LEFT, CENTER (por defecto), y RIGHT. Para alineación vertical: TOP, CENTER (por defecto), y BOTTOM.	[todavía no]
void setHorizontalTextPosition(int) void setVerticalTextPosition(int) int getHorizontalTextPosition() int getVerticalTextPosition()	Seleccionan u obtienen donde se debe situar el texto del botón, en relación a la imagen del botón. La clase AbstractButton define tres posibles valores para alineación horizontal: LEFT, CENTER y RIGHT (por defecto). Para alineación vertical: TOP, CENTER (por defecto), y BOTTOM.	[todavía no]
void setPad(Insets) Insets getPad()	Selecciona u obtiene el número de pixels entre el borde del botón y sus contenidos.	[todavía no]

void setFocusPainted(boolean) boolean isFocusPainted()	Selecciona u obtiene si el botón debería parecer diferente cuando tiene el foco.	[todavía no]
void setBorderPainted(boolean) boolean isBorderPainted()	Selecciona u obtiene su el border del botón debería ser dibujado.	[todavía no]

Implementar la Funcionalidad del Botón

Metodos o Constructor	Propósito	Ejemplo
void setSelected(boolean) boolean isSelected()	Selecciona u obtiene si el botón está seleccionado. Solo tiene sentido si el botón tiene un estado de on/off, como los checkboxes.	[todavía no]
Object[] getSelectedObjects()	Obtiene los objetos seleccionados dentro del botón. Tiene sentido sólo para los botones como Menus que contienen muchos ítems seleccionables.	[todavía no]
void setActionCommand(String) String getActionCommand(void)	Selecciona u obtiene el nombre de la acción realizada por el botón.	[todavía no]
void setKeyAccelerator(char) char getKeyAccelerator()	Selecciona u obtiene la tecla alternativa para pulsar el botón.	ButtonDemo.java
void addActionListener(ActionListener) ActionListener removeActionListener()	Añade o elimina un objeto que escucha los eventos acción disparados por el botón.	[todavía no]
void addChangeListener(ChangeListener) ChangeListener removeChangeListener()	Añade o elimina un objeto qu escucha los eventos change disparados por el botón.	[todavía no]
void addItemListener(ItemListener) ItemListener removeItemListener()	Añade o elimina un objeto que esucha los eventos ítem disparados por el botón.	[todavía no]
void doClick()	Programaticamente realiza un "click".	[todavía no]

Cómo utilizar Checkboxes

La versión Swing soporta checkboxes con las clases [JCheckbox](#) y [ButtonGroup](#). Como JCheckbox desciende de AbstractButton, los checkboxes de Swing tienen todas las características normales de los botones, como se explica en [la página anterior](#). Por ejemplo, se pueden especificar imágenes para utilizarlas en los checkboxes.

Aquí tienes una imagen de una aplicación que tiene dos checkboxes:



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en [CheckboxDemo.java](#). Puedes ver la página [Empezando con Swing](#) si necesitas ayuda.
 2. Pulsa el botón 2.
El botón 2 se selecciona y el botón 1 permanece seleccionado.
 3. Mira el mensaje mostrado en la salida estándar.
Esta aplicación registra un oyente para cada tipo de evento que un botón puede enviar. Cada vez que recibe un evento, la aplicación muestra un mensaje que describe el evento.
 4. Pulsa de nuevo el botón 2, y mira los mensajes mostrados en la salida estándar.
-

Un checkbox genera un evento de ítem y un evento de acción por cada pulsación,. Normalmente, lo único que necesita un manejador de eventos de checkboxes es un oyente de ítem. Si tu prefieres utilizar el API asociado con los eventos acción, puedes utilizar un oyente de acción en su lugar. No se necesita implementar un oyente de cambio a menos que tu programa necesite saber en todo momento los cambios de apariencia del botón.

Abajo tienes el código de [CheckboxDemo.java](#) que crea los checkboxes del ejemplo anterior y reacciona a los clics.

```
//en el código de inicialización:
// Crear los botones.
JCheckbox firstButton = new JCheckbox(first);
firstButton.setKeyAccelerator('1');
firstButton.setActionCommand(first);
firstButton.setSelected(true);

JCheckbox secondButton = new JCheckbox(second);
secondButton.setKeyAccelerator('2');
secondButton.setActionCommand(second);

// Registra un oyente para los checkboxes.
CheckboxListener myListener = new CheckboxListener();
firstButton.addActionListener(myListener);
firstButton.addChangeListener(myListener);
firstButton.addItemListener(myListener);
secondButton.addActionListener(myListener);
secondButton.addChangeListener(myListener);
secondButton.addItemListener(myListener);

...
class CheckboxListener implements ItemListener,           //Sólo es necesario el tipo
                                   ActionListener,      //sólo por curiosidad
```

```

        ChangeListener {                                //sólo por curiosidad
public void itemStateChanged(ItemEvent e) {
    System.out.println("ItemEvent received: "
        + e.getItem()
        + " is now "
        + ((e.getStateChange() == ItemEvent.SELECTED)?
            "selected.":"unselected"));
}

public void actionPerformed(ActionEvent e) {
    String factoryName = null;

    System.out.print("ActionEvent received: ");
    if (e.getActionCommand() == first) {
        System.out.println(first + " pressed.");
    } else {
        System.out.println(second + " pressed.");
    }
}

public void stateChanged(ChangeEvent e) {
    System.out.println("ChangeEvent received from: "
        + e.getSource());
}
}

```

Puedes ver la página [How to Use Buttons](#) para más información sobre el API de AbstractButton del que desciende JCheckbox. El único API definido por JCheckbox que te gustará utilizar son los constructores. JCheckbox define siete constructores:

- JCheckbox(String)
- JCheckbox(String, boolean)
- JCheckbox(Icon)
- JCheckbox(Icon, boolean)
- JCheckbox(String, Icon)
- JCheckbox(String, Icon, boolean)
- JCheckbox()

Los argumentos son correctos:

String

Especifica el texto que debería mostrar el checkbox.

Icon

Especifica la imagen que debería mostrar el checkbox. A menos que se especifique una imagen se utilizan las imágenes definidas por la máquina virtual.

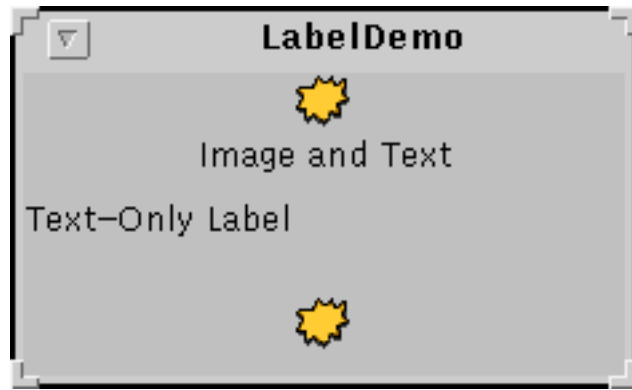
boolean

Especifica si el checkbox está seleccionado. Por defecto es false [no seleccionado].

Cómo utilizar Labels

Con la clase [JLabel](#), puedes mostrar un texto seleccionable e imágenes. Las etiquetas no pueden obtener el foco del teclado.

Aquí tienes un gráfico de una aplicación que muestra tres etiquetas. La ventana está dividida en tres columnas de la misma altura; la etiqueta de cada columna es tan ancha como sea posible.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en [LabelDemo.java](#).
Puedes ver la página [Empezar con Swing](#) si necesitas ayuda.

2. Redimensiona la ventana para poder ver cómo se alinean las etiquetas.

Todas las etiquetas tienen la alineación vertical por defecto -- están en el centro vertical de su espacio. La etiqueta del nivel superior que contiene tanto imagen como texto, está especificada para tener alineamiento horizontal centrado. La segunda etiqueta, que sólo contiene texto, tiene alineamiento a la izquierda que es por defecto para las etiquetas de sólo texto. La tercera etiqueta, que sólo contiene una imagen, tiene alineamiento horizontal centrado que es por defecto para las etiquetas de sólo imagen.

Abajo tienes el código de [LabelDemo.java](#) que crea las etiquetas en el ejemplo anterior.

```
ImageIcon icon = new ImageIcon("middle.gif");  
.  
.  
.  
label1 = new JLabel("Image and Text", icon, JLabel.CENTER);  
label1.setVerticalTextPosition(JLabel.BOTTOM);  
label1.setHorizontalTextPosition(JLabel.CENTER);  
  
label2 = new JLabel("Text-Only Label");
```

```
label3 = new JLabel(icon);

//Add labels to the JPanel.
add(label1);
add(label2);
add(label3);
```

Las siguientes tablas listan los constructores y métodos de JLabel más utilizados. Otros métodos que podrías utilizar están definidos en la clase [Component](#). Entre ellas se incluyen setFont y setForeground.

Seleccionar u Obtener los contenidos de una Etiqueta

Método o Constructor	Propósito	Ejemplo
JLabel(Icon) JLabel(Icon, int) JLabel(String) JLabel(String, Icon, int) JLabel(String, int) JLabel()	Crea un ejemplar de JLabel, inicializándolo para que tenga el texto/imagen/alineamiento especificado. El argumento int especifica el alineamiento horizontal del contenido de la etiqueta dentro de su área de dibujo. El alineamiento debe ser una de las siguientes constantes definidas en el interface GraphicsUtilsConstants (que implementa JLabel): LEFT, CENTER, or LEFT.	LabelDemo.java
void setText(String) String getText()	Selecciona u obtiene el texto mostrado en la etiqueta.	[Todavía ninguno]
void setIcon(Icon) Icon getIcon()	Selecciona u obtiene la imagen mostrada por la etiqueta.	[Todavía ninguno]
void setRepresentedKeyAccelerator(char) char getRepresentedKeyAccelerator()	Selecciona u obtiene la letra que debería aparecer como atajo de teclado. Esto es útil cuando una etiqueta describe un componente [como un campo de texto] que tiene una tecla de atajo de teclado pero no puede mostrarlo.	[todavía ninguno]
void setDisabledIcon(Icon) Icon getDisabledIcon()	Selecciona la imagen que se muestra en la etiqueta cuando está desactivada. Si no se especifica una imagen, la máquina virtual crea una mediante una manipulación de la imagen por defecto.	[Todavía ninguno]

Ajuste Fino de la Apariencia de una Etiqueta

Método	Propósito	Ejemplo

void setHorizontalAlignment(int) void setVerticalAlignment(int) int getHorizontalAlignment() int getVerticalAlignment()	Selecciona u obtiene dónde se debe situar el contenido de la etiqueta. El interface GraphicsUtilsConstants define tres posibles valores para el alineamiento horizontal: LEFT (por defecto para etiquetas de sólo texto), CENTER (Por defecto para etiquetas de sólo imagen), o LEFT. Para alineamiento vertical: TOP, CENTER (por defecto), y BOTTOM.	[Todavía ninguno]
void setHorizontalTextPosition(int) setVerticalTextPosition(int) int getHorizontalTextPosition() int getVerticalTextPosition()	Selecciona u obtiene dónde se debería situar el texto, en relación con la imagen. El interface GraphicsUtilsConstants define tres posibles valores para posición horizontal: LEFT, CENTER, y LEFT (por defecto). Para posición vertical: TOP, CENTER (por defecto), y BOTTOM.	LabelDemo.java
void setIconTextGap(int) int getIconTextGap()	Selecciona el número de pixels entre el texto y la imagen de la etiqueta.	[Todavía ninguno]

Cómo utilizar Botones de Radio

Los botones de Radio son grupos de botones en los que sólo uno de ellos puede ser seleccionado a la vez. La versión Swing soporta botones de radio con las clases [JRadioButton](#) y [ButtonGroup](#).

Como [JRadioButton](#) descende de [AbstractButton](#), los botones de radio de Swing tienen las características normales de los botones, como se explica en la página [Cómo utilizar Botones](#). Por ejemplo, puedes especificar el texto (si lo hay) y la imagen en un botón de radio.

Aquí tienes un gráfico de una aplicación que tiene dos botones de radio:



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en [RadioButtonDemo.java](#). Puedes ver la página [Empezar con Swing](#) si necesitas ayuda.
 2. Pulsa el botón 2.
El botón 2 se selecciona, lo que hace que el botón 1 se deselectione.
 3. Mira los mensajes mostrados en la salida estándar.
Esta aplicación registra un oyente para cada tipo de evento que puede enviar un botón -- action, change e ítem. Cada vez que recibe un evento la aplicación muestra un mensaje que describe el evento.
 4. Pulsa de nuevo el botón 1, y mira los mensajes mostrados en la salida estándar.
-

Normalmente, lo único que necesita un manejador de eventos de un botón de radio es un oyente de action. Puedes utilizar un oyente de ítem en su lugar, si sólo estás monitorizando los cambios de estado, en vez de la propia acción. No se necesita implementar un oyente de cambio a menos que necesites saber en cada momento si la apariencia del botón ha cambiado.

Abajo tienes el código de [RadioButtonDemo.java](#) que crea los botones de radio del ejemplo anterior y reacciona a los clicks.

```
//En el código de inicialización:
// Crear los botones.
JRadioButton firstButton = new JRadioButton(first);
firstButton.setKeyAccelerator('1');
firstButton.setActionCommand(first);
firstButton.setSelected(true);

JRadioButton secondButton = new JRadioButton(second);
secondButton.setKeyAccelerator('2');
secondButton.setActionCommand(second);

// Agrupa los botones de radio.
ButtonGroup group = new ButtonGroup();
group.add(firstButton);
group.add(secondButton);

// Registra un oyente para los botones de radio.
RadioListener myListener = new RadioListener();
firstButton.addActionListener(myListener);
```



```

firstButton.addChangeListener(myListener);
firstButton.addItemListener(myListener);
secondButton.addActionListener(myListener);
secondButton.addChangeListener(myListener);
secondButton.addItemListener(myListener);
. . .
class RadioListener implements ActionListener, ChangeListener, ItemListener {
    public void actionPerformed(ActionEvent e) {
        String factoryName = null;

        System.out.print("ActionEvent received: ");
        if (e.getActionCommand() == first) {
            System.out.println(first + " pressed.");
        } else {
            System.out.println(second + " pressed.");
        }
    }

    public void itemStateChanged(ItemEvent e) {
        System.out.println("ItemEvent received: "
            + e.getItem()
            + " is now "
            + ((e.getStateChange() == ItemEvent.SELECTED)?
                "selected.":"unselected"));
    }

    public void stateChanged(ChangeEvent e) {
        System.out.println("ChangeEvent received from: "
            + e.getSource());
    }
}

```

Puedes ver la página [Cómo utilizar Botones](#) para más información sobre el API de AbstractButton del que descende JRadioButton. JRadioButton define siete constructores:

- JRadioButton(String)
- JRadioButton(String, boolean)
- JRadioButton(Icon)
- JRadioButton(Icon, boolean)
- JRadioButton(String, Icon)
- JRadioButton(String, Icon, boolean)
- JRadioButton()

Los argumentos son:

String

Especifica el texto que debería mostrar el botón de radio.

Icon

Especifica la imagen que debería mostrar el botón de radio. A menos que se especifique una imagen, la máquina virtual utilizará las imágenes predefinidas.

boolean

Especifica si el botón de radio está seleccionado. Por defecto es false (no seleccionado).

Para cada grupo de botones de radio, necesitas crear un ejemplar de ButtonGroup y añadirle

cada uno de los botones de radio. El ButtonGroup asegura que sólo uno de los botones de grupo esté seleccionado. Aquí tienes los métodos y constructores de ButtonGroup:

ButtonGroup()

 Crea un ejemplar de ButtonGroup.

add(AbstractButton)

 Añade al grupo el botón especificado.

remove(AbstractButton)

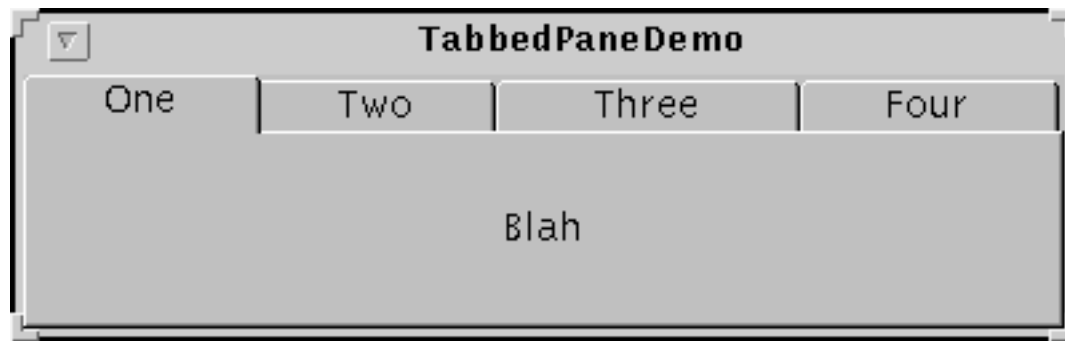
 Elimina el botón especificado del grupo.

Cómo utilizar Pestañas

Con la clase [JTabbedPane](#), puedes tener varios componentes [normalmente objetos JPanel] compartiendo el mismo espacio. El usuario puede elegir el componente que quiere ver seleccionando la pestaña correspondiente al componente deseado. Si quieres una funcionalidad similar sin el interface de pestañas, podrías utilizar el controlador de disposición [CardLayout](#) en vez de JTabbedPane.

Para crear una pestaña, sólo debes ejemplarizar JTabbedPane, crear los componentes que deseas mostrar, y luego añadir los componetes a las pestañas.

Aquí tienes una imagen de una aplicación que utiliza cuatro pestañas:



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en [TabbedPaneDemo.java](#).
Puedes ver la página [Empezar con Swing](#) si necesitas ayuda.
 2. Pon el cursor sobre una pestaña.
Después de un momento, verás el aviso asociado con la etiqueta.
Como es conveniente, puedes especificar un texto de aviso cuando añadas un componente a una pestaña.
 3. Selecciona una pestaña.
La pestaña muestra el componente correspondiente.
-

Una pestaña puede mostrar tanto texto como imágenes.

Abajo tienes el código de [TabbedPaneDemo.java](#) que crea las pestañas del ejemplo anterior. Observa que no es necesario el código de manejo de eventos. El objeto JTabbedPane se ocupa de las entradas del usuario por tí.

```
JTabbedPane tabbedPane = new JTabbedPane();

Component panel1 = makeTextPanel("Blah");
tabbedPane.addTab("One", null, panel1, "Does nothing");
tabbedPane.setSelectedIndex(0);
```

```
Component panel2 = makeTextPanel("Blah blah");
tabbedPane.addTab("Two", null, panel2, "Does nothing");

Component panel3 = makeTextPanel("Blah blah blah");
tabbedPane.addTab("Three", null, panel3, "Does nothing");

Component panel4 = makeTextPanel("Blah blah blah blah");
tabbedPane.addTab("Four", null, panel4, "Does nothing");
. . .
add(tabbedPane);
```

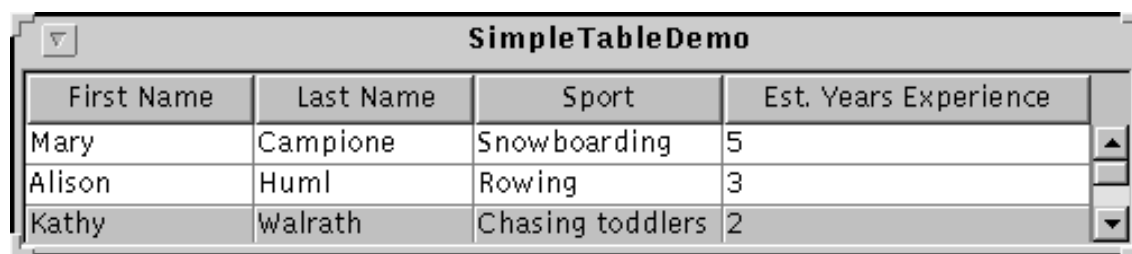
Cómo utilizar Tablas

Con la clase [JTable](#), puedes mostrar tablas de datos. JTable no contiene datos, es sólo una vista de tus datos. Esta forma de caché hace que JTable funcione bien, incluso cuando los datos vienen de una enorme base de datos. JTable tiene demasiadas características para poder ser descritas completamente en este tutorial, por eso sólo nos referiremos a las características más utilizadas por la mayoría de los programadores.

Para utilizar JTable, deberás escribir un modelo de datos -- un objeto que suministre datos a la tabla. Puedes hacer esto creando una subclase de [JTableDataModelAdapter](#). Un modelo de datos contiene métodos para obtener y seleccionar valores de datos, obtener el número de filas (registros) de datos, y para añadir o eliminar oyentes de eventos. [Los oyentes de Tabla son objetos como JTable que son notificados cada vez que el modelo de datos cambia.]

Cuando inicialices una JTable, deberás proporcionar un objeto [JTableColumn](#) para cada columna. Cada objeto JTableColumn identifica una columna de datos y especifica como deberían ser mostrados.

Aquí tienes una imagen de una aplicación que muestra una tabla en un panel desplazable:



Intenta esto:

1. compila y ejecuta la aplicación. El fichero fuente está en [SimpleTableDemo.java](#). Puedes ver la página [Empezar con Swing](#) si necesitas ayuda.
 2. Pulsa sobre una celda.
La fila entera es seleccionada. Esto es intencionado para recordarte que JTable implementa una base de datos no una hoja de cálculo.
 3. Posiciona el cursor sobre la cabecera "First Name". Ahora pulsa el botón del ratón y arrastra la cabecera a la derecha.
Como puedes ver, los usuarios pueden reordenar las columnas de la tabla. Este posicionamiento flexible es porque las columnas están especificadas como objetos en vez de índices [como se utiliza para las filas].
 4. Posiciona el cursor al borde derecho de la cabecera. Ahora pulsa el botón del ratón y arrastralo a la derecha o la izquierda.
La columna cambia de tamaño.
 5. Redimensiona la ventana que contiene la tabla para que sea mayor que el contenido completo de la tabla.
Las celdas de la tabla permanecen con el mismo tamaño, y permanecen en la esquina superior izquierda del área de pantalla.
-

Abajo tienes el código de [SimpleTableDemo.java](#) que implementa la tabla del ejemplo anterior.

```
//En el código de inicialización de una subclase de JPanel:  
MyDataModel myDataModel = new MyDataModel();  
JTable table = new JTable(myDataModel);
```

```

        for (int columnIndex = 0;
            columnIndex < myDataModel.numColumns;
            columnIndex++) {
            JTableColumn newColumn = new JTableColumn(
                myDataModel.getColumnName(columnIndex));
            table.addColumn(newColumn);
            newColumn.setWidthToFit();
        }

        //Crea el panel desplazable y le añade la tabla.
        JScrollPane scrollPane = new JScrollPane();
        scrollPane.getViewPort().add(table);

        //hace que la cabecera de la tabla no se desplace.
        JViewport columnHeader = new JViewport();
        columnHeader.setView(table.getTableHeader());
        columnHeader.setLayout(new BorderLayout(columnHeader, //HACK
            BorderLayout.X_AXIS));
        scrollPane.setColumnHeading(columnHeader);
    }

    . . .
    class MyDataModel extends JTableDataModelAdapter {

        //Datos inútiles.
        final Object[][] data = {
            {"First Name", "Mary", "Alison", "Kathy", "Mark", "Angela"},
            {"Last Name", "Campione", "Huml", "Walrath", "Andrews", "Lih"},
            {"Sport", "Snowboarding", "Rowing", "Chasing toddlers", "Speed reading",
"Teaching high school"},
            {"Est. Years Experience", "5", "3", "2", "20", "4"},
        };

        public int numColumns = data.length;
        protected int numRows = data[0].length - 1;

        public int getRowCount() {
            return numRows;
        }

        public Object getValueAt(Object columnIdentifier, int rowIndex) {
            for (int columnIndex = 0; columnIndex < numColumns; columnIndex++) {
                if (data[columnIndex][0].equals(columnIdentifier)) {
                    return data[columnIndex][rowIndex+1];
                }
            }
            return "NO DATA";
        }

        public void setValueAt(Object aValue,
                                Object columnIdentifier,
                                int rowIndex) {
        }

        /** This method isn't in TableDataModel. */
        public String getColumnName(int columnIndex) {
            return (String)data[columnIndex][0];
        }
    }

```

}

}

Ozito

Cómo utilizar Tool Tips

Crear un Tool Tip [etiqueta de ayuda] para cualquier JComponent es sencillo, Sólo debes utilizar el método `setToolTipText` para activar el Tool Tip del componente. Por ejemplo, para añadir [etiquetas de ayuda] a los tres botones del programa `ButtonDemo` (explicado en la página [Cómo Utilizar Buttons](#)), sólo tienes que añadir tres líneas de código:

```
b1.setToolTipText("Click this button to disable the middle button.");  
b2.setToolTipText("This middle button does nothing when you click it.");  
b3.setToolTipText("Click this button to enable the middle button.");
```

Cuando el usuario del programa pase el cursor sobre uno de los botones del programa la [etiqueta de ayuda] se mostrará. Por ejemplo:

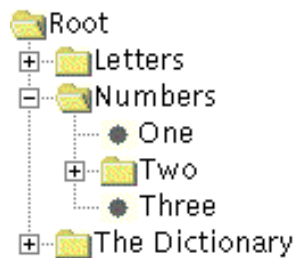


[Por favor imagínate un cursor sobre el botón izquierdo. Gracias.]

Para más información sobre Tool Tips puedes leer la documentación de las clases [JToolTip](#) y [JComponent](#) en la site de Sun.

Cómo Utilizar Trees

Con la clase [JTree](#), puedes mostrar datos en forma de árbol. JTree realmente no contiene tus datos, simplemente es una vista de ellos. Aquí tienes una imagen de una árbol:



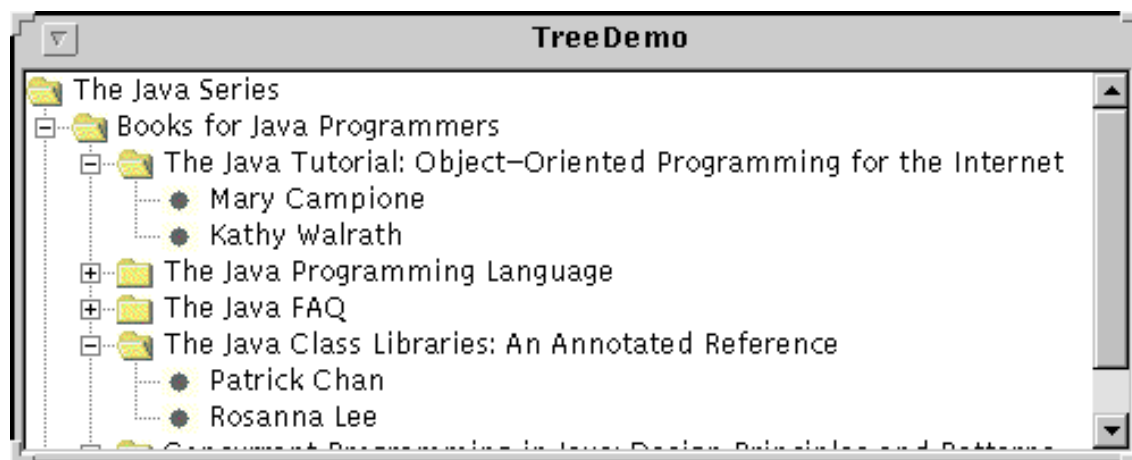
Como muestra la imagen anterior, JTree muestra sus datos verticalmente. Cada fila contiene exactamente un ítem de datos [llamado nodo]. Cada árbol tiene un nodo raíz [llamado Root en la figura anterior] del que descienden todos los nodos. Los nodos que no pueden tener hijos se llaman nodos hoja. En la figura anterior se marcan los nodos hoja con un círculo.

Los nodos que no son hojas pueden tener cualquier número de hijos, o incluso ninguno. En la imagen superior los estos nodos están representados por una carpeta. Normalmente el usuario podrá expandir y contraer los nodos que no hojas -- haciendo que sus hijos sean visibles o invisibles. Por defecto estos nodos arrancas contraídos.

Cuando inicializas un Tree, creas un ejemplar de [TreeNode](#) para cada uno de los nodos. Incluyendo el raíz. Para hacer un nodo hoja, llama a su método `setAllowsChildren(false)`.

Cuando los datos cambian -- por ejemplo, se añaden nodos -- puedes permitir que `JTree[Model?]` sepan que hay nuevos datos incocando uno de los métodos de [TreeModelListener](#). Entonces JTree muestra los nuevos datos.

Aquí tienes un gráfico de una aplicación que muestra el tree en un Scroll Pane:



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en [TreeDemo.java](#). Puedes ver la página [Getting Started with Swing](#) si necesitas ayuda.
2. Expande un nodo.
Si estás utilizando una Máquina virtual básica, puedes hacer esto pulsado el signo + que hay a la izquierda del ítem.
3. Selecciona un nodo.
Si estás utilizando una Máquina virtual básica, puedes hacer esto pulsado el texto del nodo o el icono que hay a la izquierda.

abajo tienes el código de [TreeDemo.java](#) que implementa el árbol del ejemplo anterior.

```
//en el código de inicialización de JPanel:
//Crea los nodos.
TreeNode top = new TreeNode("The Java Series");
TreeNode category;
TreeNode book;

category = new TreeNode("Books for Java Programmers");
top.add(category);

//Tutorial
book = new TreeNode("The Java Tutorial: Object-Oriented Programming for the
Internet");
book.add(new TreeNode("Mary Campione"));
book.add(new TreeNode("Kathy Walrath"));
category.add(book);

...//hacer lo mismo para cad libro...

category = new TreeNode("Books for Java Implementers");
top.add(category);

//VM
book = new TreeNode("The Java Virtual Machine Specification");
book.add(new TreeNode("Tim Lindholm"));
book.add(new TreeNode("Frank Yellin"));
category.add(book);

//Liberar todos los nodos sin hijos.
makeLeaves(top);

JTree tree = new JTree(top);
. . .
//Este método se añadirá al API de TreeNode API en una futura versión.
protected void makeLeaves(TreeNode top) {
    Enumeration      childEnum = top.preorderEnumeration();
    TreeNode          descendant;

    while(childEnum.hasMoreElements()) {
        descendant = (TreeNode)childEnum.nextElement();
        if(descendant.getChildCount() == 0)
            descendant.setAllowsChildren(false);
    }
}
```