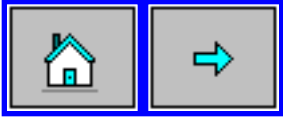


Programación en el lenguaje Java



[Fundamentos del lenguaje Java](#)

[Creación de applets](#)

[Enlaces a webs de Java](#)

El análisis es la base de la programación estructurada, es decir, la descomposición de una tarea en tareas más pequeñas. Un programa no es otra cosa que una colección de funciones que son llamadas sucesivamente por la función principal y única del programa. Cada función tiene sus propias variables, es un módulo independiente. La Programación Orientada a Objetos es el siguiente paso en la evolución de los lenguajes de programación, que combina funciones y datos en una unidad autoconsistente denominada clase.

La creación de interfaces gráficos como el Windows de Microsoft es uno de los ejemplos más claros de las ventajas que tiene la programación orientada a objetos sobre la programación estructurada. Las librerías nos proporcionan clases que describen el aspecto y la conducta de elementos como la ventana, el diálogo, los controles, etc. Derivando una clase de otra base, estamos utilizando el código de la clase base para nuestras propias necesidades, y le añadimos el código que implementa la conducta específica.

Un lenguaje de Programación Orientada a Objetos nos permite organizar el código en entidades como las clases compuestas de datos y funciones, y a través de la característica de la herencia podemos organizar las clases en jerarquías.

Aunque la Programación Orientada a Objetos es la manera más natural de programar, aclimatarse a su ambiente requiere cierto tiempo, no se producen resultados inmediatos. Los numerosos ejemplos que se comentarán a lo largo de las siguientes páginas tratarán de ayudar a efectuar dicha transición de un modo suave y ordenado. Una vez aclimatado, sentirá por sí mismo la necesidad de organizar el código en clases, y posteriormente, si lo requiere el problema, establecer jerarquías entre dichas clases.

A través de estas páginas, pretendemos mostrar la potencia y elegancia de la programación en lenguaje Java en dos situaciones concretas: en la creación de interfaces gráficas avanzadas (applets), y en la [resolución de problemas numéricos](#).

Se pondrá especial énfasis en la Programación Orientada a Objetos que es la parte que los no iniciados encuentran más difícil. Aprender a extraer las propiedades y el comportamiento comunes a un determinado tipo de objetos y transformarlas en la definición de una clase, estructurar las clases en jerarquías, constituyen procesos de abstracción que implican un cambio de mentalidad en la resolución de los problemas de software que precisan de entrenamiento y de un esfuerzo intenso.

El objetivo es el de enseñar al lector a traducir la descripción de un problema o situación que se plantea a código, a organizar el código en funciones, a agrupar datos y funciones en clases y las clases en jerarquías.

Una vez entendidos los conceptos básicos, la [creación de applets](#) no presentará grandes dificultades si se hace con la ayuda de un Entorno Integrado de Desarrollo como JBuilder de Borland (ahora Inprise), el cual genera casi todo el código correspondiente al interfaz dejando al programador la tarea de darle funcionalidad, es decir, la de definir la respuesta a las distintas acciones.

Fundamentos del lenguaje Java



[Introducción](#)

[Clases y objetos](#)

[La herencia y el polimorfismo](#)

[Las excepciones](#)

[Pasando datos a una función](#)

[Las clases *Vector* y *StringTokenizer*](#)

[Archivos](#)

El lenguaje Java se parece al lenguaje C++ de modo que un programador que conozca este lenguaje ha dado un gran paso adelante.

Sin embargo, existen también grandes diferencias entre ambos lenguajes. Un programador puede haber usado el lenguaje C++ como un lenguaje C mejorado sin haber usado para nada la Programación Orientada a Objetos. Sin embargo, Java es un lenguaje plenamente orientado a objetos, y para escribir el programa más simple hemos de definir una clase. Los tipos básicos de datos son similares, pero los arrays son distintos, y las cadenas de caracteres en Java son objetos de la clase *String*.

[Introducción](#)

Muchos lectores que hayan programado en algún lenguaje, comprenderán sin dificultad este capítulo. Sin embargo, se recomienda leerlo con atención ya que incluso en los aspectos básicos hay diferencias entre unos lenguajes y otros. En esta introducción aprenderemos aspectos básicos del lenguaje Java: la primera aplicación, los comentarios, los tipos básicos de datos, los operadores, las sentencias condicionales e

iterativas.

Clases y objetos

Este capítulo es fundamental para entender la Programación Orientada a Objetos. Aprenderemos el concepto de clase, a crear objetos de una determinada clase, a acceder desde dichos objetos a los miembros dato y a los miembros función, a distinguir entre miembros estáticos y no estáticos, y muchas otras cosas más. Estudiaremos dos entidades muy importantes en cualquier lenguaje de programación: los arrays, y las cadenas de caracteres o strings, además de otras clases importantes en lenguaje Java

La herencia y el polimorfismo

Este es otro de los aspectos fundamentales de la Programación Orientada a Objetos. Trataremos de la herencia, de la reutilización del código, del concepto de clase abstracta y su diferencia con el concepto de interface. El significado de polimorfismo y de enlace dinámico. Aprenderemos que todas las clases en Java descienden de la clase base *Object*, que proporciona una funcionalidad mínima. La utilidad de las clases y de los métodos finales.

La parte más difícil de entender es el polimorfismo, es decir, la técnica que permite pasar un objeto de una clase derivada a funciones que conocen el objeto solamente por su clase base.

Las excepciones

Los programadores en cualquier lenguaje se esfuerzan por escribir programas libres de errores. Sin embargo, es muy difícil que los programas reales se vean libres de ellos. Aunque el lenguaje Java es muy robusto, existen situaciones que pueden provocar un fallo en el programa, a estas situaciones se denominan excepciones. Vamos a ver cómo se capturan las excepciones estándar, y aprenderemos a definir nuestras propias excepciones, a lanzarlas cuando se produce una situación peculiar, y a capturarlas.

Pasando datos a una función

Pasar datos a una función es uno de las dificultades con las que se enfrenta un programador del lenguaje C++, ya que hay tres formas: por valor, por dirección y por referencia. En el lenguaje Java lo simplifica extraordinariamente, ya que todo se pasa por valor, pero los efectos son distintos, cuando se pasan tipos básicos de datos, de cuando se pasan objetos de una determinada clase.

También, hay una gran diferencia en el mecanismo que nos permite duplicar un objeto, o de crear un objeto que es idéntico a otro dado. En el lenguaje C++ se emplea el constructor copia, mientras que en el lenguaje Java se redefine la función miembro *clone* de la clase base *Object*.

Las clases *Vector* y *StringTokenizer*

Finalmente trataremos dos clases que son muy útiles para el programador: la clase *Vector*, y la clase *StringTokenizer*. La clase *Vector* es similar a un array, pero no pone límite en el número máximo de objetos que se pueden guardar. Además, nos proporciona un conjunto de métodos para acceder, añadir o eliminar elementos.

La clase *StringTokenizer*, nos permite dividir un string dado en trozos o tokens. Su utilidad se pone de manifiesto en el tratamiento de los datos que se introducen en un control área de texto.

Archivos

Este capítulo no es esencial para el estudio de los applets ya que por razones de seguridad los applets no pueden acceder a los archivos del ordenador cliente, por lo que puede ser omitido en una primera lectura de este curso.

Sin embargo, el lenguaje Java no está diseñado exclusivamente para crear applets que corren en una ventana del navegador, sino también aplicaciones como cualquier otra que corremos en los entornos Windows, Unix, Macintosh o Linux. Muchas aplicaciones tienen acceso a los dispositivos estándar y a los archivos en disco, por lo que el lenguaje Java define un conjunto de clases agrupadas en jerarquías que describen los distintos flujos de datos.

Una de las facetas más potentes del lenguaje Java es la denominada serialización, que permite convertir los objetos en un flujo de bytes, marcando tan solo la clase que describe dichos objetos como *Serializable*. A su vez, esta característica del lenguaje nos permite reconstruir los objetos leyendo un flujo de datos que proviene de una fuente normalmente un archivo en disco o incluso de la propia Red.

Applets



[Introducción](#)

[El contexto gráfico](#)

[Gestores de diseño](#)

[Respuesta a las acciones del usuario sobre los controles](#)

[Subprocesos \(threads\)](#)

[Estudio de ejemplos completos](#)

[La tecnología de componentes: JavaBeans](#)

Introducción

Mediante una secuencia de imágenes se señala los pasos que han de seguirse para crear un applet mínimo con el Entorno Integrado de Desarrollo (IDE) JBuilder 2.0.

En el primer applet se redefine la función *paint*, para proporcionar alguna funcionalidad al applet, en este caso mostrar un mensaje. A continuación, se indican los pasos que hay que seguir para publicar un applet una vez que se ha completado el código fuente, y se ha depurado suficientemente:

- Se crea un archivo **.jar** con los archivos .class resultado de la compilación, proceso denominado deployment
- Se inserta la etiqueta APPLET en la correspondiente página web

Finalmente, se explica la forma en la que se comunican el applet y la página web, a través de los parámetros de la etiqueta APPLET.

El contexto gráfico

En este apartado, se estudia la clase *Graphics*, que describe el contexto gráfico de un componente. Los objetos de esta clase llaman a las funciones miembro para dibujar una línea, un rectángulo, una elipse, texto, etc, en el área de trabajo del componente. Se estudian también clases relacionadas como *Color*, *Font* y *FontMetrics*, que describen el color, las fuentes de texto y las características que las definen.

Gestores de diseño

En la superficie del applet se disponen los componentes: los paneles y sobre estos los controles. Para ello, se selecciona con el ratón un componente en la paleta correspondiente y se sitúa sobre el applet o sobre otro componente. Ahora bien, en Java no se pueden situar los componentes en lugares precisos, alinearlos, etc, como ocurre en Windows, ya que en Java existen los denominados gestores de diseño. El más simple de los gestores de diseño es *FlowLayout* y el más complicado es *GridBagLayout*.

Respuesta a las acciones del usuario sobre los controles.

La versión AWT 1.1 introduce un modelo denominado "Delegation Event Model" completamente distinto de codificar las acciones del usuario sobre los controles. El código de los applets de las versiones 1.0 y 1.1 presenta notables diferencias. Un programador que diseñe interfaces gráficos (GUI) precisa entender los mecanismos de interacción entre el usuario y el programa: cómo se codifica la respuesta a la acción de pulsar sobre un botón, a la de seleccionar un elemento de una lista; cómo se elabora una única respuesta a la acción sobre un conjunto de controles; cómo se verifica la introducción de datos en un control de edición, etc.

La acción más común es la pulsación sobre un botón. Esta situación nos servirá para introducir las bases del nuevo modelo, las definiciones de los distintos términos que intervienen: sucesos (events), componentes, los objetos (listeners) que manejan los sucesos, las funciones respuesta, etc. A continuación, estudiamos la respuesta a las acciones del usuario sobre controles individuales: el control lista, control selección (*Choice*), la barra de desplazamiento, y el control de edición. Pasaremos luego, a estudiar la respuesta a las acciones sobre un grupo de controles: un conjunto de tres botones, de tres casillas de verificación o de tres botones de radio.

El ratón es uno de los dispositivos estándares en un interfaz gráfico, que facilita la interacción del usuario con el programa, por lo que saber programar las acciones del ratón es fundamental. Comenzaremos por la acción más simple, la de hacer clic sobre un punto de la pantalla gráfica. A continuación, abordaremos

las acciones de pulsar el botón izquierdo del ratón, arrastrarlo y liberarlo, para dibujar "a mano alzada".

El control canvas es importante por que nos permite separar las distintas tareas. El applet controla la interacción usuario/ordenador por medio de los controles y en el canvas se lleva a cabo la representación gráfica, una animación, etc. El problema se presentará a la hora de comunicar los objetos applet y al canvas.

Terminamos este largo y a la vez fundamental capítulo con dos ejercicios que se plantean al lector para que conozca el grado de comprensión de los conceptos explicados hasta este momento.

Subprocesos (threads)

El lenguaje Java es muy apropiado para crear subprocesos que corren simultáneamente. Esta no es una característica añadida al lenguaje, sino que el lenguaje ha sido diseñado para que soporte esta característica que nos permite crear programas más simples y fáciles de entender.

Para crear un subproceso hay dos aproximaciones:

- crear una clase derivada de *Thread*.
- implementar el interface *Runnable*

En la primer página, estudiaremos qué es un subproceso, y cómo se crea mediante el primer procedimiento. También veremos cuál es el ciclo de vida de un subproceso, y cómo se establecen prioridades entre distintos subprocesos que corren a la vez.

Los subprocesos, en general, no corren uno independientemente del otro. El problema surge a la hora de coordinar las tareas que realizan dos o más subprocesos que acceden a los mismos datos. Por ejemplo, un subproceso se encarga de escribir datos en un buffer y otro proceso se encarga de leerlos. Un ejemplo basado en el modelo Productor/Consumidor intentará aclarar este aspecto complicado.

La creación de animaciones es una aplicación directa de los subprocesos (threads). Como veremos, Java es el lenguaje ideal para programar animaciones en la Web. Vamos a estudiar dos situaciones distintas:

- Una figura invariable que se mueve por el área de trabajo del applet.
- Una figura inmóvil cuyo aspecto cambia con el tiempo.

A la primera categoría pertenecen muchos de los programas del [Curso Interactivo de Física en Internet](http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/applets.htm). Respecto de la segunda, estudiaremos la clase *MediaTracker* que supervisa la carga de las imágenes y otros elementos multimedia a través de la red.

Estudio de ejemplos completos

Se estudia una serie de ejemplos, algunos de los cuales forman parte del Curso Interactivo de Física en Internet.

Error en las medidas directas

Se explica cómo se ha creado uno de los applets más simples de programar del Curso Interactivo de Física. Se describirán los objetivos del programa, la creación de clases para distintos propósitos: una clase que describe el interfaz y otra para el tratamiento de los datos. La disposición de los controles en la ventana del applet, y la definición de las funciones respuesta a las acciones del usuario sobre los controles.

Conversión de unidades

Varios son los aspectos que se tratarán en este ejemplo ilustrativo: cómo se crea un grupo de botones de radio. La disposición de los controles empleando el gestor de diseño *GridBagLayout*. La respuesta a las acciones del usuario sobre un botón, la verificación de los datos introducidos en los controles de edición. La respuesta a las acciones del usuario sobre un grupo de botones de radio.

Diagramas

Las diagramas de barras o en forma de tarta son empleados ampliamente para representar datos. Por ejemplo, los ingresos o gastos que ha tenido una compañía por distintos conceptos. El número de alumnos de una clase que ha suspendido, aprobado, que ha sacado notable o sobresaliente. La importancia de este ejemplo, estriba en la creación de una jerarquía de clases: una clase base abstracta en la cual se hace un tratamiento de los datos introducidos. De esta clase derivan otras dos que definen la función miembro *dibuja* para dibujar un diagrama de barras o un diagrama en forma de tarta.

Representación gráfica de una función

La representación gráfica tiene mucha importancia en las ciencias y en la ingeniería ya que nos permite comprender el comportamiento de un sistema de un solo vistazo. En este caso, se representa la ley de la [distribución de velocidades de Maxwell](#), que describe la proporción de moléculas de un gas ideal que tienen una velocidad determinada a una temperatura dada. Este ejemplo, nos permite examinar, el origen, los ejes de la gráfica, las escalas vertical y horizontal. Cómo se ponen divisiones en los ejes y se etiquetan convenientemente, cómo se superponen gráficas de distintos colores para que podamos compararlas, etc.

Un programa de dibujo simple

Se simula un programa de dibujo simple, con una caja de herramientas. En el programa se estudia el modo de dibujo XOR, que nos permite dibujar una figura extensible; la clase *Vector* para guardar objetos gráficos. Finalmente, podemos optar por definir una clase que describe todas las figuras, o bien crear una jerarquía de clases formada por una clase base abstracta, y por clases derivadas que definen la función que dibuja la figura concreta en un contexto gráfico.

El movimiento de los planetas

El movimiento de los planetas explica un típico applet del Curso Interactivo de Física. Se describen los fundamentos físicos, el procedimiento numérico de cálculo (el método de Runge-Kutta). Se aborda el problema de la animación, es decir, la representación en la ventana del applet de la posición del móvil en función del tiempo. La técnica conocida como double-buffer, para mostrar datos que cambian sin que se aprecie un molesto parpadeo. Finalmente, se explica el diseño del interfaz en base a paneles anidados.

La tecnología de componentes: JavaBeans

En la industria electrónica como en otras industrias se está acostumbrado a utilizar componentes para construir placas, tarjetas, etc. En el campo del software la idea es la misma. Se puede crear un interfaz de usuario en un programa Java en base a componentes: paneles, botones, etiquetas, caja de listas, barras de desplazamiento, diálogos, menús, etc.

En este capítulo, se enseña que es un javaBean y cómo se puede crear un javaBean con los asistentes de JBuilder 2.0. A continuación, aprenderemos a insertarlo en la paleta de componentes y usarlo en nuestras aplicaciones. Estudiaremos en detalle sus propiedades y su comportamiento: cómo emite sucesos (events) cuando cambia los valores de sus propiedades ligadas, y cómo se conecta la fuente de los sucesos con otros objetos interesados en la notificación del cambio en los valores de dichas propiedades. Dichos objetos (listeners) reciben información acerca de los sucesos y realizan una determinada tarea.

Finalizaremos este estudio, creando un bean a partir de dos componentes AWT, y que puede tener utilidad para los programadores en el ámbito científico.

Enlaces Java



[Gamelan](#)

[Jars](#)

[JavaWorld](#)

Introducción



La Máquina Virtual Java

- La Máquina Virtual Java
- El lenguaje Java

La primera aplicación

- Crear un nuevo proyecto
- La aplicación
- Ejecutar la aplicación
- Resumen

Elementos del lenguaje Java

- Identificadores
- Comentarios
- Sentencias
- Bloques de código
- Expresiones
- Variables
- Los tipos básicos de datos
- Las cadenas de caracteres o strings
- Palabras clave

Los operadores (aritméticos)

- Los operadores aritméticos
- La concatenación de strings
- La precedencia de operadores
- La conversión automática y promoción
- Los operadores unarios

Los operadores (relacionales)

- Los operadores relacionales
- Los operadores lógicos

El flujo de un programa (sentencias condicionales)

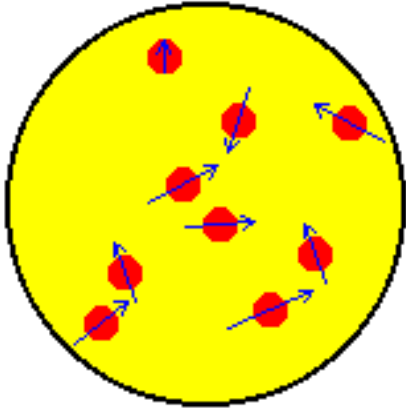
- La sentencia *if*
- La sentencia *if...else*
- La sentencia *switch*

El flujo de un programa (sentencias iterativas)

- La sentencia *for*
- La sentencia *while*
- La sentencia *do...while*
- La sentencia *break*
- La sentencia *continue*

Departamento de Física Aplicada I

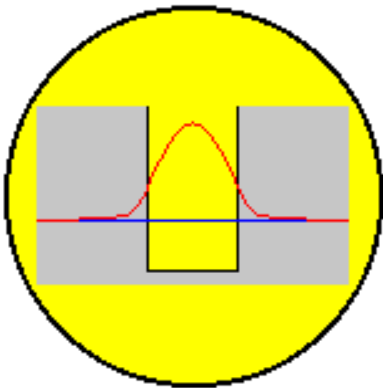
Escuela Universitaria de Ingeniería Técnica Industrial de Eibar



Curso Interactivo de Física en Internet



Programación en lenguaje Java

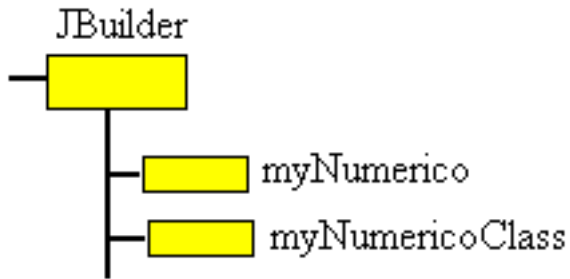


Procedimientos numéricos en lenguaje Java



Los programas ejemplo se han creado con JBuilder2 de [Borland International](http://www.borland.com/)

Para compilar los programas se ha de crear la estructura de directorios que se indica. Descargue aquí los programas de la parte del curso que le interese



- [Fundamentos del lenguaje Java](#)
- [Creación de applets](#)
- [Procedimientos numéricos en lenguaje Java](#)

Si lo desea puede [descargar el curso completo](#) (lenguaje Java y procedimientos numéricos) incluidos los ejemplos.

Advertencia: Aunque el curso se ha desarrollado con el IDE JBuilder 2.0 de Borland, el lector puede emplear cualquier otra herramienta de programación que sea de su preferencia.



[Angel Franco García](#)

Última actualización:

Curso de Lenguaje Java: Enero de 2000

Procedimientos Numéricos en Lenguaje Java: Diciembre de 2001

Clases y objetos



Conceptos básicos de la Programación Orientada a Objetos

- Introducción
- La clase
- Los objetos
- La vida de un objeto
- Identificadores

Composición

- La clase *Punto*
- La clase *Rectangulo*
- Objetos de la clase *Rectangulo*

La clase *String*

- La clase *String*
- Cómo se obtiene información acerca del string
- Comparación de strings
- Extraer un substring de un string
- Convertir un número a string
- Convertir un string en número
- La clase *StringBuffer*

Los arrays

- Declarar y crear un array
- Inicializar y usar el array
- Arrays multidimensionales
- El código fuente

Los paquetes

- El paquete (**package**)
- El comando **import**
- Paquetes estándar

Los números aleatorios

- La clase *Random*
- Comprobación de la uniformidad de los números aleatorios
- Secuencias de números aleatorios

Creación de clases

- Definición de la clase *Lista*
- Objetos de la clase *Lista*

Miembros estáticos

- Variables de instancia, variables de clase
- Miembros estáticos

La clase *Math*

- Miembros dato constantes
- Funciones miembro
- Cálculo del número irracional π

Una clase con funciones estáticas

- El valor absoluto de un número
- La potencia de exponente entero de un número entero
- El factorial de un número
- Determinar si un número es o no primo
- La clase *Matematicas*
- Llamada a las funciones miembro

La clase *Fraccion*

- Los miembros dato
- Las funciones miembro
- La clase *Fraccion*
- Uso de la clase *Fraccion*
- Modificadores de acceso
- Mejora de la clase *Lista*

La herencia y el polimorfismo



La clase base y la clase derivada

- Introducción
- La clase base
- La clase derivada
- Controles de acceso
- La clase base *Object*

La jerarquía de clases

- La jerarquía de clases que describen las figuras planas
- Uso de la jerarquía de clases
- Enlace dinámico
- El polimorfismo en acción

Ampliación de la jerarquía de clases

- Añadiendo nuevas clases a la jerarquía
- El polimorfismo en acción
- El operador **instanceof**
- Resumen

Clases y métodos finales

- Clases finales
- Métodos finales

Interfaces

- ¿Qué es un interface?
- Diferencias entre un interface y una clase abstracta

Las excepciones



Las excepciones estándar

- Las excepciones
- Captura de las excepciones
- Manejando varias excepciones

Las excepciones propias

- La clase que describe la excepción
- El método que puede lanzar una excepción
- Captura de las excepciones
- Una función que puede lanzar varias excepciones
- La cláusula **finally**

Pasando datos a una función



Paso por valor

- Pasando datos de tipo básico
- Pasando objetos

Duplicación de objetos

- El interface *Cloneable*
- Duplicación de un objeto simple
- Duplicación de un objeto compuesto
- Duplicación de arrays unidimensionales
- Duplicación de arrays bidimensionales

Las clases *Vector* y *StringTokenizer*



La clase Vector

- Crear un vector
- Añadir elementos al vector
- Acceso a los elementos de un vector

La clase StringTokenizer

- Los constructores
- Obtención de tokens

Entrada/salida



Archivos y directorios

- La clase *File*
- Creación de un filtro

Flujos de datos

- Las jearaquías de clases
- Lectura
- Escritura

Entrada/salida estándar

- Los objetos *System.in* y *System.out*
- La clase *Reader*

Entrada/salida a un archivo en disco

- Lectura de un archivo de texto
- Lectura/escritura

Leer y escribir datos de tipo primitivo

- Los flujos de datos *DataInputStream* y *DataOutputStream*
- Ejemplo: un pedido
- El final del archivo

Leer y escribir objetos

- El interface *Serializable*
- Lectura/escritura
- El modificador *transient*
- Objetos compuestos
- La herencia
- Serialización personalizada

Introducción



Un proyecto nuevo

- Un proyecto nuevo
- Asistente para la creación de un applet
- Correr el applet

El primer applet

- El primer applet
- Insertando el applet en una página web
- Comprensión de los archivos **.class** (deployment)

Los parámetros

- La anchura y altura del applet
- Otros parámetros

El contexto gráfico



Funciones gráficas

- El contexto gráfico
- Funciones gráficas

Las clases *Color*, *Font* y *FontMetrics*

- La clase *Color*
- La clase *Font*
- La clase *FontMetrics*

Gestores de diseño



Los gestores *FlowLayout*, *BorderLayout* y *GridLayout*

- La paleta de componentes
- Añadir componentes al applet
- El gestor *FlowLayout*
- El gestor *BorderLayout*
- El gestor *GridLayout*

El gestor de diseño *GridBagLayout*

- Las propiedades que gobiernan este gestor de diseño
- Ejemplo: diseño de una ficha

Respuesta a las acciones del usuario sobre los controles



Bases del nuevo modelo

- Sucesos, componentes, interfaces
- Respuesta a la acción de pulsar sobre un botón
- La clase que describe el applet implementa el interface *Listener*
- Una clase denominada *AccionBoton* implementa el interface *Listener*
- Cómo genera JBuilder el código
- Clases internas
- Clases internas anónimas
- Separando el código de inicialización de la respuesta.

Respuesta a las acciones del usuario sobre diversos controles

- Hacer doble clic sobre un elemento de una lista
- Seleccionar un elemento de una lista
- El control seleccion (*Choice*)
- Lista de elección múltiple
- El control barra de desplazamiento

Respuesta a las acciones del usuario sobre un grupo de controles

- La información acerca del suceso
- Respuesta a las acciones del usuario sobre un conjunto de tres botones
- Tres funciones respuesta
- Un único objeto maneja los sucesos de un grupo de botones

- Respuesta a las acciones sobre controles casilla de verificación (*Checkbox*)
- Respuesta a las acciones del usuario sobre un grupo de botones de radio

Programación del ratón

- Introducción
- *Listeners* y *adapters*
- Pulsando el botón izquierdo del ratón
- Dibujar los puntos y guardarlos en memoria
- Un programa simple de dibujo "a mano alzada"

Verificación de la información que introduce el usuario en un control de edición

- Introducción
- El control de edición
- Verificación de los datos mientras se introducen
- Verificación después de introducir los datos

El control canvas

- El control *Canvas*
- Seleccionar un color en una caja de selección (*Choice*)
- Dibujar una circunferencia de radio variable en un canvas
- Dibujar un punto pulsando el botón izquierdo del ratón
- Dibujar "a mano alzada"

Sucesos, componentes, interfaces y funciones respuesta

Ejercicios

- Crear una fuente de texto a partir de las selecciones efectuadas en tres listas (nombre de las fuentes, estilos y tamaños)
- Coordinar una barra de desplazamiento y un control de edición

Subprocesos (threads)



La clase derivada de *Thread*

- El método *run*
- Derivando de la clase *Thread*
- Prioridades
- Control de la ejecución de los subprocesos

Implementando el interface *Runnable*

- El interface *Runnable*
- La vida del subproceso
- Fechas y horas
- Dibujando el reloj
- Eliminando el parpadeo
- El double-buffer

Sincronización

- La palabra reservada **synchronized**
- El modelo Productor/Consumidor
- Cargando imágenes

Ejercicio

- Crear un conjunto de subprocesos que se ejecutan a la vez independientemente uno del otro

Moviendo una figura por el área de trabajo de una

ventana

- La primera aproximación al problema de la animación
- Eliminando el parpadeo
- Insertando un retardo en el bucle sin fin

Control de la animación

- El diseño del applet
- Los botones que controlan la animación
- La clase que describe el canvas

Una figura cuyo aspecto cambia con el tiempo

- Pasos para crear una animación
- Cargando y mostrando las imágenes
- La clase *MediaTracker*

Estudio de ejemplos completos



Error en las medidas directas

- Planteamiento del problema
- Diseño del applet
- Una clase para el tratamiento de los datos
- Conclusiones

La conversión de unidades.

- Introducción
- Un grupo de botones de radio
- Disposición de los controles empleando el gestor *GridBagLayout*
- Respuesta a la pulsación sobre un botón
- Verificación de los datos introducidos en controles de edición.
- Respuesta a las acciones sobre un grupo de controles botón de radio.

Diagramas

- Diseño del applet
- La jerarquía de clases
- La clase que describe el canvas

Representación gráfica de una función

- Diseño del applet
- Respuesta a las acciones del usuario sobre los controles
- El área de la representación gráfica
- Definición de la función
- La clase que describe el canvas

Un programa de dibujo simple

- Diseño del applet
- Manejo del ratón
- Guardar las figuras en memoria
- La clase que describe las figuras
- La jerarquía de clases
- Conclusiones

El movimiento de los planetas

- Introducción
- Fundamentos físicos y procedimientos numéricos
- Movimiento del planeta
- El estado del móvil
- La comunicación entre el usuario y el programa
- Relación entre los objetos de las distintas clases
- Conclusiones
- Bibliografía

Componentes JavaBeans



Introducción a los JavaBeans

- Definición de JavaBean
- Propiedades

Propiedades ligadas.

- Una clase con una propiedad ligada (bound)
- La clase que define un suceso
- El interface XXXListener
- La fuente de los sucesos (events)
- Los objetos (listeners) interesados
- Vinculación entre la fuente de sucesos y los objetos (listeners) interesados

Creación de JavaBeans con JBuilder.

- Un nuevo proyecto
- Añadir un bean al proyecto
- Definir una propiedad
- La clase cuyos objetos (listeners) están interesados en el cambio en el valor de la propiedad
- Vinculación entre la fuente de sucesos y los objetos (listeners) interesados

Un botón rudimentario como bean

- Crear un proyecto
- Añadir un bean al proyecto
- Propiedades simples
- Persistencia
- Aspecto del botón

- El tamaño del canvas
- Relación entre el aspecto y las propiedades del bean
- Respondiendo a las acciones del usuario
- El botón emite un suceso del tipo *ActionEvent*.
- El bean terminado
- La clase *BeanInfo*
- Deployment

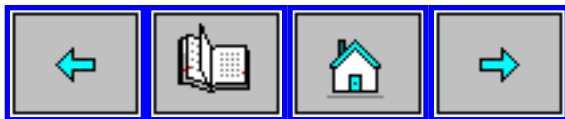
Uso del bean

- La paleta de componentes
- El uso del bean
- La respuesta a la acción del usuario sobre el botón

Un bean hecho con controles AWT

- Diseño del bean
- Propiedades y sucesos del bean
- Relación entre el aspecto y las propiedades del bean
- Preparación del bean
- El uso del bean

Un proyecto nuevo



[Introducción](#)

[Un proyecto nuevo](#)

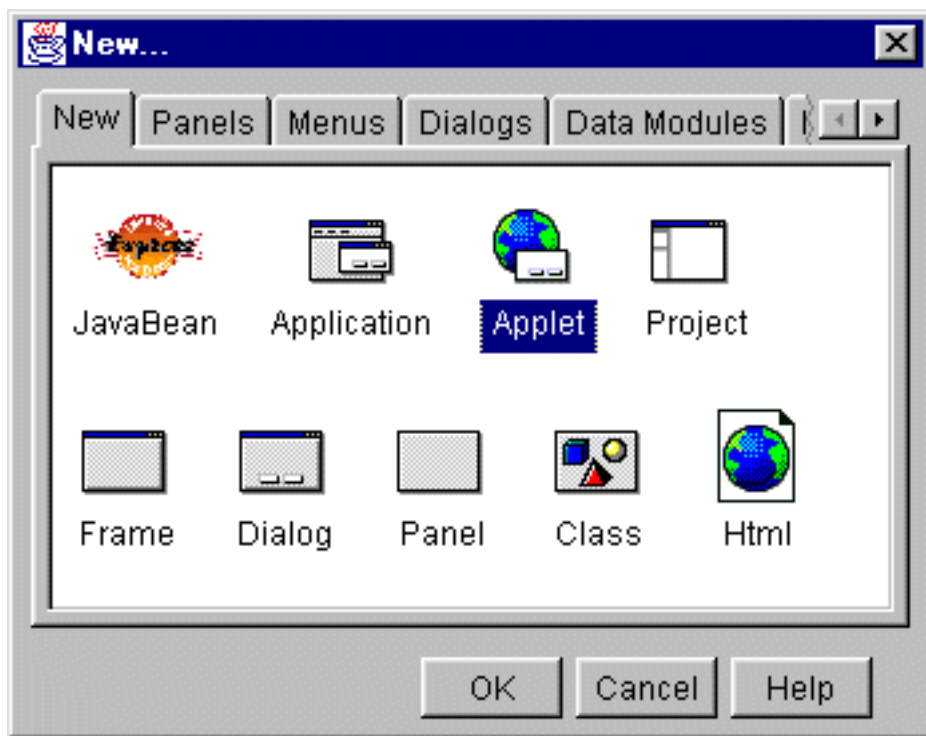
[Asistente para la creación de un applet](#)

[Correr el applet](#)

Como veremos a lo largo de esta parte del curso el Entorno Integrado de Desarrollo (IDE) JBuilder facilita enormemente la tarea de creación y publicación de los applets. En esta página, vamos a ver mediante una secuencia de imágenes cómo se crea un applet mínimo.

Un proyecto nuevo

Para crear un proyecto se selecciona **File/New** apareciendo el diálogo titulado **New**.



Se selecciona el icono **Applet** y a continuación se pulsa el botón **OK** para cerrar el diálogo.

Aparece el [asistente para la creación de proyectos](#), el mismo que ya vimos en el estudio de los fundamentos del lenguaje Java.



Vamos a crear un proyecto titulado **applet1** y que se va a guardar en un subdirectorío denominado **applet1**.

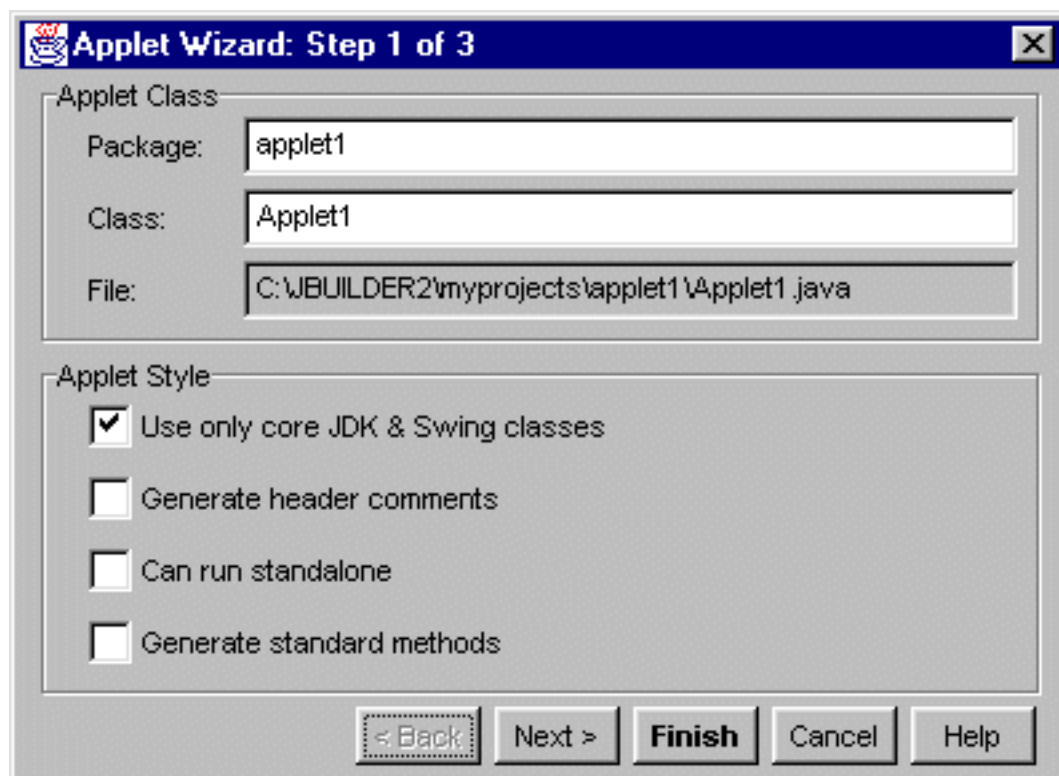
Por defecto, JBuilder sitúa los proyectos en el subdirectorio **myprojects**. Dentro de este subdirectorio se crea la carpeta titulada **applet1** (**untitled1** es el valor por defecto) y crea un archivo denominado **applet1.jpr** (**untitled1.jpr** por defecto). En todos los programas ejemplo, el nombre del proyecto será idéntico al nombre de la carpeta que lo contiene.

Opcionalmente, se puede rellenar los campos titulados Title, Author, Company y Description. JBuilder genera el archivo **applet1.html** que contiene la información que suministramos en dichos campos.

Asistente para la creación de un applet

Al cerrar el diálogo pulsando en el botón titulado **Finish** aparece el asistente para la creación del applet que consta de tres pasos

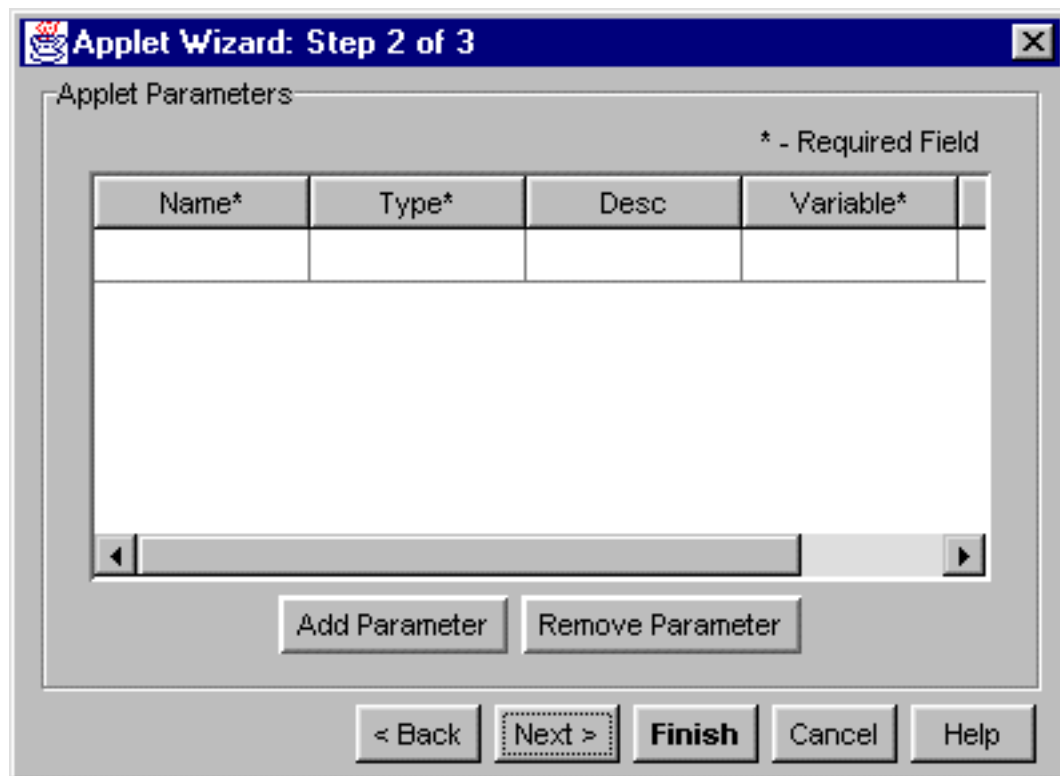
El primer diálogo se titula **Applet Wizard Step 1 of 3**, y nos permite introducir el nombre que asignamos a la clase que describe el applet.



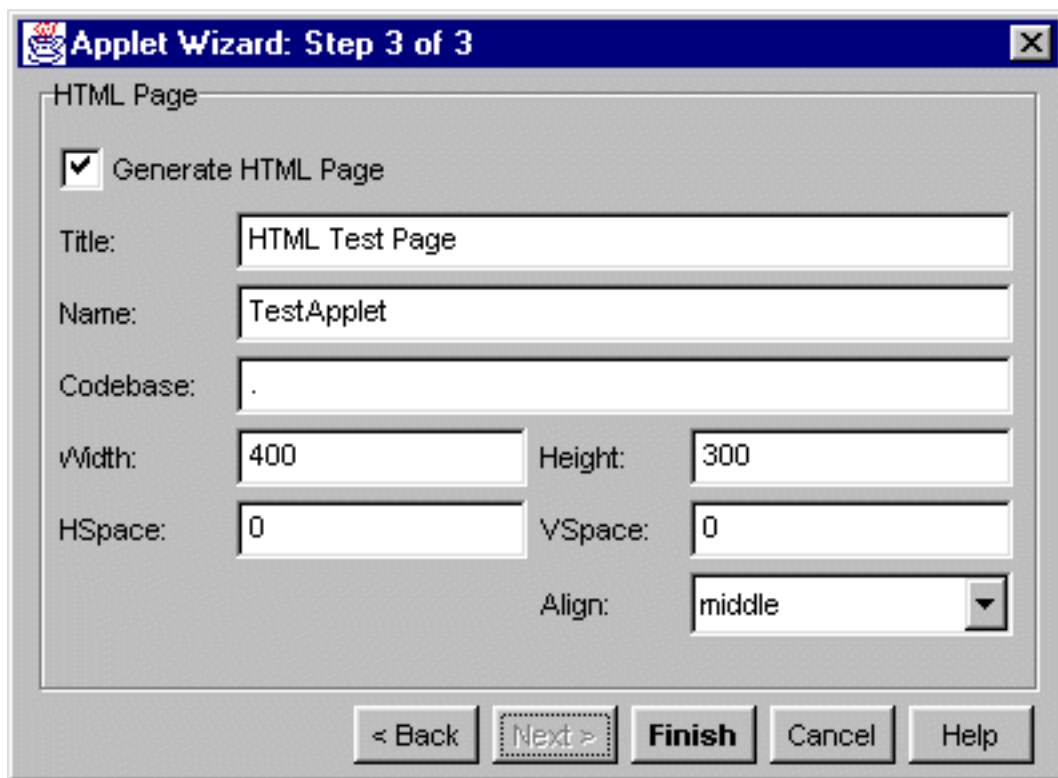
Modificamos el campo denominado **Class**, le ponemos un nombre a la clase que describe el applet, *Applet1*, en este caso dejamos el nombre que pone JBuilder por defecto. Ahora podemos, pulsar el botón **Finish** para cerrar el asistente, o bien continuar con el segundo paso.

Pulsando en el botón titulado **Next**, aparece el diálogo titulado **Applet Wizard Step 2 of 3**, y nos permite

comunicar la página HTML y el applet que está insertado en ella, suministrándole ciertos datos que acompañan a los denominados parámetros de la etiqueta APPLET. De momento, no usaremos este diálogo



Volviendo a pulsar el botón **Next**, aparece el diálogo titulado **Applet Wizard Step 3 of 3** que genera un archivo **.html** de una página que contiene la etiqueta APPLET, y que será explicado en la página siguiente.



The image shows a Java Applet Wizard dialog box, specifically Step 3 of 3, titled "HTML Page". The dialog has a standard Windows-style title bar with a close button. The main area contains a checkbox labeled "Generate HTML Page" which is checked. Below this are several input fields: "Title" with the text "HTML Test Page", "Name" with "TestApplet", "Codebase" with a single dot ".", "Width" with "400", "Height" with "300", "HSpace" with "0", "VSpace" with "0", and an "Align" dropdown menu set to "middle". At the bottom of the dialog are five buttons: "< Back", "Next >", "Finish", "Cancel", and "Help". The "Next >" button is disabled, while the others are active.

Applet Wizard: Step 3 of 3

HTML Page

☒ Generate HTML Page

Title: HTML Test Page

Name: TestApplet

Codebase: .

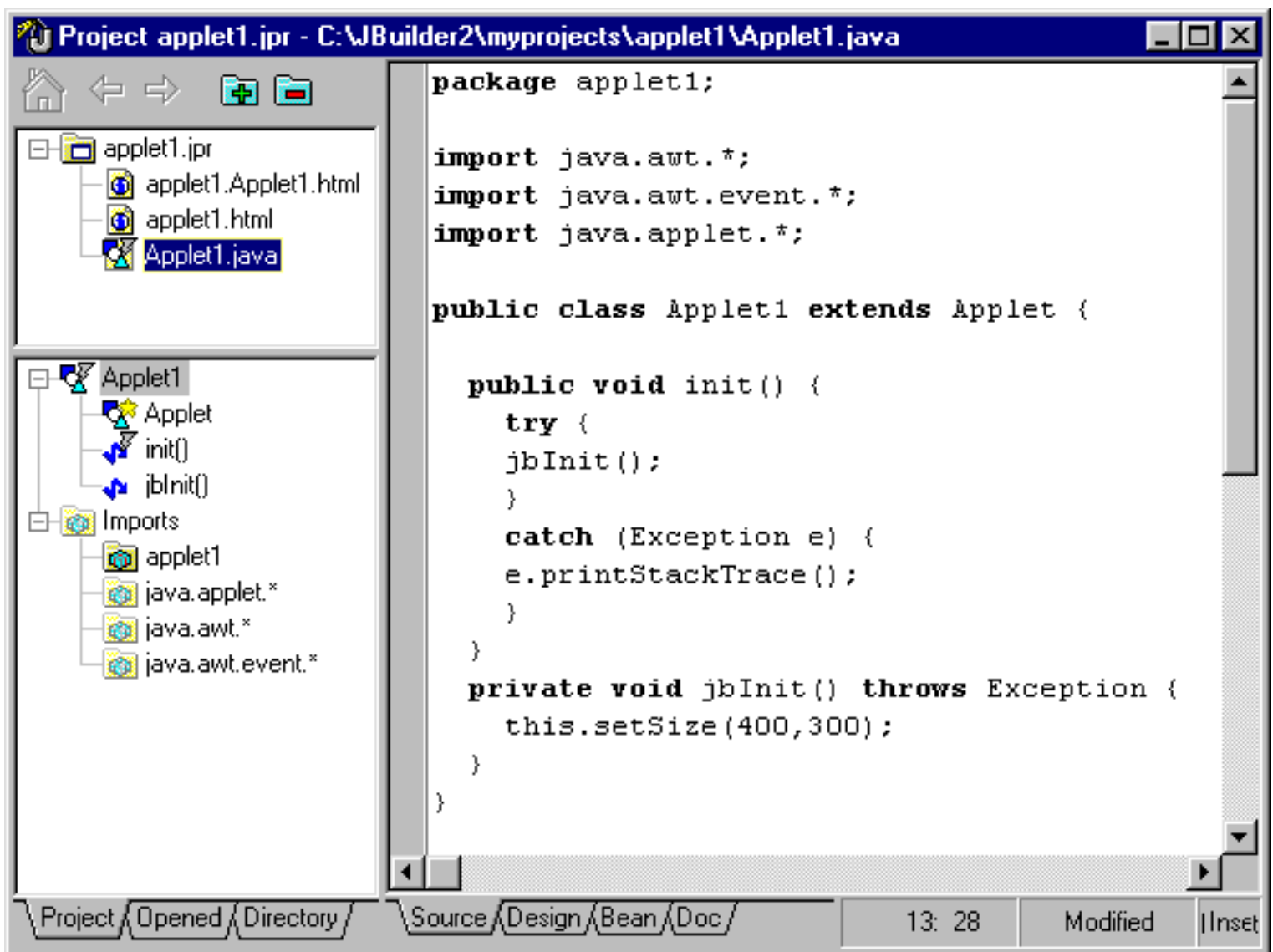
Width: 400 Height: 300

HSpace: 0 VSpace: 0

Align: middle

< Back Next > Finish Cancel Help

Habitualmente, pulsaremos el botón **Finish** en el primer paso, en vez de el tercer paso. En cualquier caso, JBuilder genera cuatro archivos y una buena porción de código.



El primer archivo **applet1.jpr** es el proyecto que está contenido en una carpeta que tiene el mismo nombre **applet1**, situada en el subdirectorio **myprojects** o en general, en el subdirectorio del código fuente (**Source root directory**) [definido en Tools/Default project properties](#). El proyecto consta como vemos en el panel de navegación (superior izquierda) de tres archivos.

applet1.html contiene, como ya se ha explicado, la documentación del proyecto, es decir, la información que introducimos en los campos del asistente para la creación de un proyecto nuevo.

applet1.Applet1.html es un archivo que está situado en el subdirectorio **myclasses** o en general, en el subdirectorio del código compilado (**Output root directory**) [definido en Tools/Default project properties](#). En este subdirectorio JBuilder sitúa los archivos compilados **.class**. Este archivo describe una página HTML que contiene la etiqueta **APPLET**. Recuerdese que un applet corre en una región rectangular de la ventana del navegador.

Por último, **Applet1.java** contiene el código fuente de la clase que describe el applet. Recuerdese que el nombre de la clase pública coincide con el nombre del archivo.

En el panel inferior izquierda, vemos la estructura de la clase, y en el panel de la derecha el código fuente que se ha generado. Previamente se ha eliminado la parte del código que no es estrictamente necesario.

El applet mínimo

Lo primero que apreciamos es que la nueva clase *Applet1* deriva de la clase base *Applet*

```
public class Applet1 extends Applet{
//...
}
```

Y en segundo lugar, que tiene un método denominado *init*, que se llama cuando se craga el applet en el navegador.

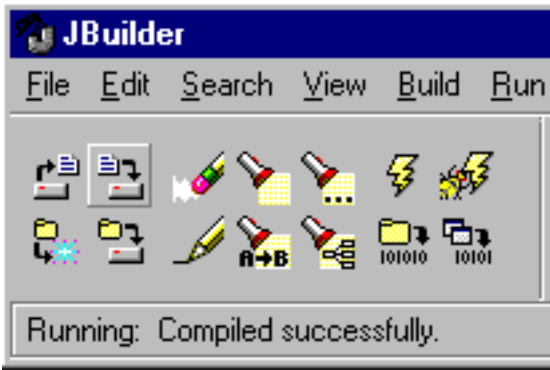
```
public class Applet1 extends Applet{
    public void init(){
        //...
    }
}
```

El código generado por JBuilder, que se observa en la imagen previa es un poco más complicado: la función *init* llama a una función privada *jbInit*. Aunque es equivalente a la porción de código situada más arriba, no debe ser cambiado ya que afecta al modo de diseño.

Ahora podemos seleccionar la pestaña **Design**, seleccionar un control de alguna de las paletas de componentes y situarlo sobre el applet. JBuilder genera el código correspondiente. Se trata de una herramienta de doble vía, los cambios en el diseño se reflejan en el código y los cambios en el código se reflejan en el diseño.

Correr el applet

Para correr el applet, apuntamos con el ratón el archivo **applet1.Applet1.html** en el panel de navegación, y a continuación, pulsamos sobre el icono en forma de rayo en la barra de herramientas.



El applet se compila generando un archivo cuya extensión es **.class** situado en la carpeta **applet1** en el subdirectorio **myclasses**. Podemos ver el applet en el **AppletViewer** de la misma forma que aparecerá en el navegador insertado en una página web.

En la mayor parte de los casos el comportamiento del applet en el **AppletViewer** es el mismo que en el navegador (Internet Explorer o Communicator). Una vez terminado el applet, es mejor asegurarse plenamente, probando su comportamiento en los navegadores compatibles con la versión JDK 1.1 antes de enviarlo al servidor para su publicación en Internet.

Funciones gráficas



[Contexto gráfico](#)

[El contexto gráfico](#)

[Funciones gráficas](#)

Hemos introducido la noción de contexto gráfico en las páginas anteriores, cuando se ha [creado el primer applet](#), que muestra un mensaje. En este capítulo se estudiarán algunas funciones gráficas definidas en la clase *Graphics*, y tres clases que sirven de apoyo para dibujar en el contexto gráfico: la clase *Color* que describe los colores, la clase *Font* que nos permite crear fuentes de texto y la clase *FontMetrics* que nos proporciona sus características.

El contexto gráfico

La función *paint* y *update* nos suministran el contexto gráfico del applet o del componente, en otros casos, hemos de obtener el contexto gráfico del componente mediante la función *getGraphics*. Una vez obtenido el contexto gráfico podemos llamar desde este objeto a las funciones gráficas definidas en la clase *Graphics*.

```
public void paint(Graphics g){
//usar el contexto gráfico g
}

public void update(Graphics g){
//usar el contexto gráfico g
}

void funcion(){
    Graphics g=getGraphics();
    //usar el contexto gráfico g
    g.dispose();
}
```

}

Como vemos en esta porción de código existe una sutil diferencia entre suministrar y obtener el contexto gráfico *g*. Solamente es necesario liberar los recursos asociados al contexto *g*, mediante la llamada a la función *dispose*, cuando se obtiene el contexto gráfico mediante *getGraphics*.

La clase *Graphics* es abstracta por lo que no se pueden crear mediante **new** objetos de esta clase, pero se pueden guardar en una referencia *g* de la clase *Graphics* los contextos gráficos concretos de los distintos componentes.

Un contexto gráfico es como la hoja en blanco situada en un trazador (plotter). Para dibujar en dicha hoja se toma una pluma, se dibuja, se toma otra pluma de distinto color o grosor, se dibuja otra porción del gráfico, y así sucesivamente. Cuando no se selecciona explícitamente, se dibuja con una pluma que se establece por defecto.

Las librerías gráficas como la de Windows, disponen de plumas de distinto grosor para dibujar líneas con distintos estilos, brochas para rellenar el interior de una figura cerrada con un color sólido, con una determinada trama o figura, y fuentes de texto, para dibujar texto con distintas fuentes y estilos. La librería gráfica que viene con la versión 1.1 de Java es muy limitada. No hay objetos pinceles, ni brochas. Las líneas tienen un único grosor y estilo, solamente se pueden cambiar de color, las figuras cerradas solamente se pueden rellenar con un color sólido, y las fuentes de texto disponibles son muy pocas.

La clase *Graphics* describe el contexto gráfico y proporciona un conjunto de funciones para dibujar las siguientes figuras

- Líneas
- Círculos y elipses
- Rectángulos y polígonos
- Imágenes
- Texto



El sistema de coordenadas que se usa en Java es similar a Windows. El área de trabajo del applet está compuesta por una matriz bidimensional de puntos o pixels. Decimos que un punto tiene de coordenadas (x, y) cuando está en la columna x medida desde la izquierda, y está en la fila y , medida desde arriba.

La esquina superior izquierda es el origen $(0, 0)$.

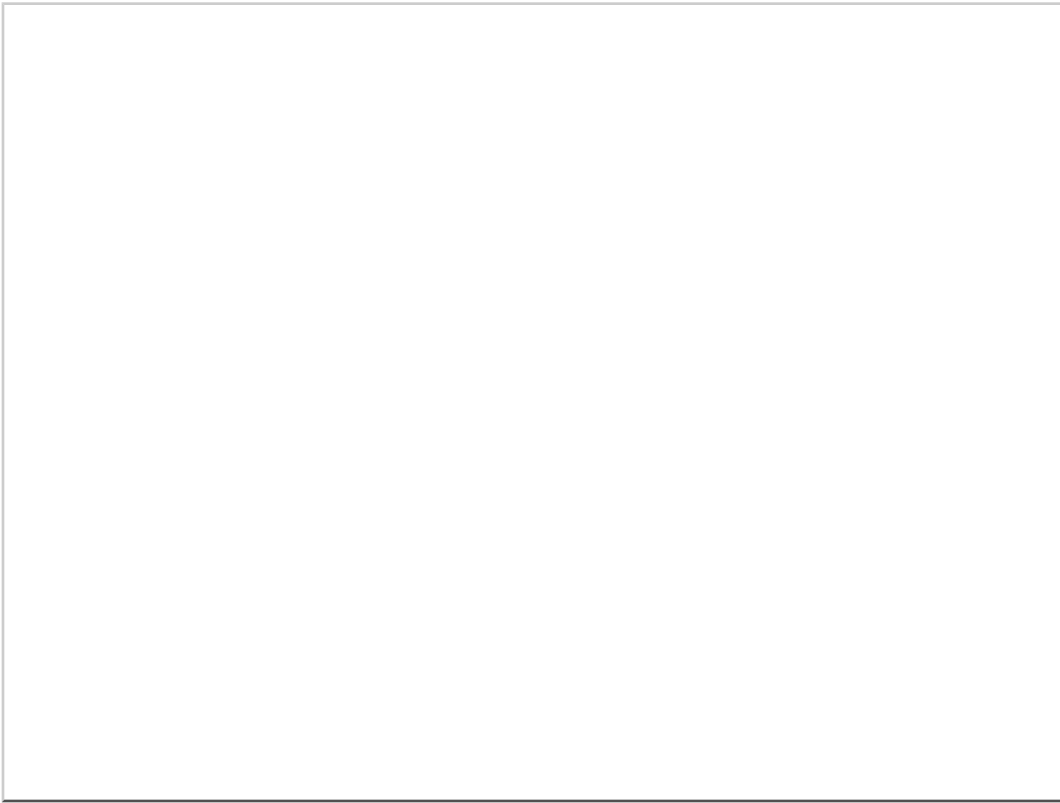
La esquina inferior derecha viene determinada por las dimensiones del componente. La función *getSize* nos devuelve un objeto de la clase *Dimension* cuyos miembros *width* y *height* nos suministran la anchura y altura del componenete.

```
int ancho=getSize().width;
int alto=getSize().height;
```

Funciones gráficas



funciones: [FuncionesApplet.java](#)



No vamos a examinar completamente la clase *Graphics*, pero si vamos a mostrar mediante un applet el uso de algunas funciones de esta clase. En el capítulo dedicado al estudio de los [ejemplos completos](#) se representarán diagramas en forma de barras o tarta, y funciones matemáticas.

La función *paint* nos va a proporcionar el objeto *g* de la clase *Graphics* que denominamos contexto gráfico del componente (applet). Desde dicho objeto llamaremos a las funciones miembro de la clase *Graphics*.

Establecer un color

El color negro es el color por defecto del contexto gráfico. Para [establecer otro color](#), como veremos en la página siguiente, se utiliza la función *setColor*, y se le pasa un color predefinido o definido por el usuario.

```
g.setColor(Color.cyan);
```

Dibujar una línea

Para dibujar una línea recta se llama a la función *drawLine*, le pasamos el punto inicial y el punto final. Para dibujar una línea diagonal desde el origen (0, 0) o esquina superior izquierda, hasta la esquina inferior derecha, obtenemos las dimensiones del applet mediante la función *getSize*, que devuelve un objeto de la clase *Dimension*. El miembro *width* nos proporciona la anchura y el miembro *height* la altura.

```
g.drawLine(0, 0, getSize().width-1, getSize().height-1);
```

Dibujar un rectángulo

Un rectángulo viene definido por un origen (esquina superior izquierda), su anchura y altura. La siguiente sentencia dibuja un rectángulo cuyo origen es el punto 50, 150, que tiene una anchura de 50, y una altura de 60. La función *drawRect* dibuja el contorno del color seleccionado, y *fillRect* dibuja el rectángulo pintando su interior del color seleccionado, en este caso de color rojo.

```
g.setColor(Color.red);
g.fillRect(50, 150, 50, 60);
```

Dibujar un arco

Los elipses (oval), arcos (arc), se dibujan en el interior del rectángulo circundante. Una elipse se dibuja mediante *drawOval* o *fillOval*, con los mismos parámetros que el rectángulo. Un arco requiere dos parámetros más el ángulo inicial y el ángulo final. Las sentencias que vienen a continuación, dibujan un arco en el interior del rectángulo cuyo origen es el punto 10, 10, cuya anchura es 150, y cuya altura es 100. El ángulo inicial es 0 y el ángulo final es 270, expresado en grados.

```
g.setColor(Color.cyan);
g.fillArc(10, 10, 150, 100, 0, 270);
g.setColor(Color.black);
g.drawArc(10, 10, 150, 100, 0, 270);
```

Dibujar un polígono

Para dibujar un polígono, se requieren un array de puntos. Un polígono y una polilínea son parecidos, el primero es una figura cerrada mientras que una polilínea es un conjunto de segmentos. Para formar un

polígono a partir de una pililínea se une el punto inicial y el punto final. El polígono precisa de un array de abscisas x , un array de ordenadas y , y la [dimensión del array](#).

```
int[] x={100, 150, 170, 190, 200};
int[] y={120, 280, 200, 250, 60};
g.setColor(Color.blue);
g.drawPolygon(x, y, x.length);
```

Alternativamente, se puede usar un objeto de la clase *Polygon*, al cual se le añaden puntos mediante la función miembro *addPoint*.

```
Polygon poligono=new Polygon();
poligono.addPoint(100, 120);
poligono.addPoint(150, 280);
poligono.addPoint(170, 200);
poligono.addPoint(190, 250);
poligono.addPoint(200, 60);
```

Para dibujar el polígono con su interior pintado del color seleccionado se llama a la función *fillPolygon* y se le pasa el objeto *poligono* de la clase *Polygon*.

```
g.setColor(Color.yellow);
g.fillPolygon(poligono);
```

Veremos en el applet un polígono cuyo contorno está dibujado en azul y su interior en amarillo.

Dibujar una imagen

Para dibujar una imagen se requieren dos pasos:

- Cargar la imagen y crear un objeto de la clase *Image*
- Dibujar dicho objeto en el contexto gráfico

Para crear una imagen u objeto *disco* de la clase *Image* a partir del archivo disco.gif se usa la función *getImage*. Le hemos de indicar la ubicación de dicho archivo relativa a la página web que contiene el applet o al código compilado. En nuestro caso, hemos situado la imagen en el mismo subdirectorio que la página web que contiene al applet. El lugar más adecuado para cargar la imagen es en la función *init*, ya que como se ha mencionado se llama una sola vez.

```
public void init(){
    disco=getImage(getDocumentBase(), "disco.gif");
}
```

Para dibujar la imagen en el contexto gráfico *g*, se llama a la función *drawImage*. Hay varias versiones de esta función, la más simple es aquella a la que se le proporciona el objeto *disco* de la clase *Image*, las coordenadas de su esquina superior izquierda (250, 50), y el observador, el propio applet o **this**.

```
g.drawImage(disco, 250, 50, this);
```

Si deseamos obtener las dimensiones del objeto *disco* de la clase *Image*, llamamos a dos funciones de esta clase *getWidth* y *getHeight*

```
int ancho=disco.getWidth(this);
int alto=disco.getHeight(this);
```

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

public class FuncionesApplet extends Applet {
    Image disco;
    public void init() {
        this.setSize(400,300);
        disco=getImage(getDocumentBase(), "disco.gif");
    }

    public void paint(Graphics g){
        g.drawLine(0, 0, getSize().width-1, getSize().height-1);

        g.setColor(Color.red);
        g.fillRect(50, 150, 50, 60);

        g.setColor(Color.cyan);
        g.fillArc(10, 10, 150, 100, 0, 270);
        g.setColor(Color.black);
        g.drawArc(10, 10, 150, 100, 0, 270);

        Polygon poligono=new Polygon();
        poligono.addPoint(100, 120);
        poligono.addPoint(150, 280);
        poligono.addPoint(170, 200);
        poligono.addPoint(190, 250);
```

```
    poligono.addPoint(200, 60);  
    g.setColor(Color.yellow);  
    g.fillPolygon(poligono);  
    int[] x={100, 150, 170, 190, 200};  
    int[] y={120, 280, 200, 250, 60};  
    g.setColor(Color.blue);  
    g.drawPolygon(x, y, x.length);  
  
    g.drawImage(disco, 250, 50, this);  
}  
}
```

Las clases *Color*, *Font* y *FontMetrics*



[Representación gráfica](#)

[La clase *Color*](#)

[La clase *Font*](#)

[La clase *FontMetrics*](#)

La clase *Color*

 paint1: [PaintApplet.java](#)

Los colores primarios son el rojo, el verde y el azul. Java utiliza un modelo de color denominado RGB, que significa que cualquier color se puede describir dando las cantidades de rojo (Red), verde (Green), y azul (Blue). Estas cantidades son números enteros comprendidos entre 0 y 255, o bien, números reales comprendidos entre 0.0 y 1.0. La siguiente tabla nos proporciona los colores más comunes y sus valores RGB.

| Nombre | Red (rojo) | Green (verde) | Blue (azul) |
|-----------|------------|---------------|-------------|
| white | 255 | 255 | 255 |
| lightGray | 192 | 192 | 192 |
| gray | 128 | 128 | 128 |
| drakGray | 64 | 64 | 64 |
| black | 0 | 0 | 0 |
| red | 255 | 0 | 0 |
| pink | 255 | 175 | 175 |
| orange | 255 | 200 | 0 |
| yellow | 255 | 255 | 0 |
| green | 0 | 255 | 0 |
| magenta | 255 | 0 | 255 |
| cyan | 0 | 255 | 255 |
| blue | 0 | 0 | 255 |

Para crear un objeto de la clase *Color*, se pasan tres números a su constructor que indican la cantidad de rojo, verde y azul.

```
Color colorRosa=new Color(255, 175, 175);
```

Mediante la función *setColor*, cambiamos color con el que dibujamos una línea, un texto o rellenamos una figura cerrada en el contexto gráfico *g*.

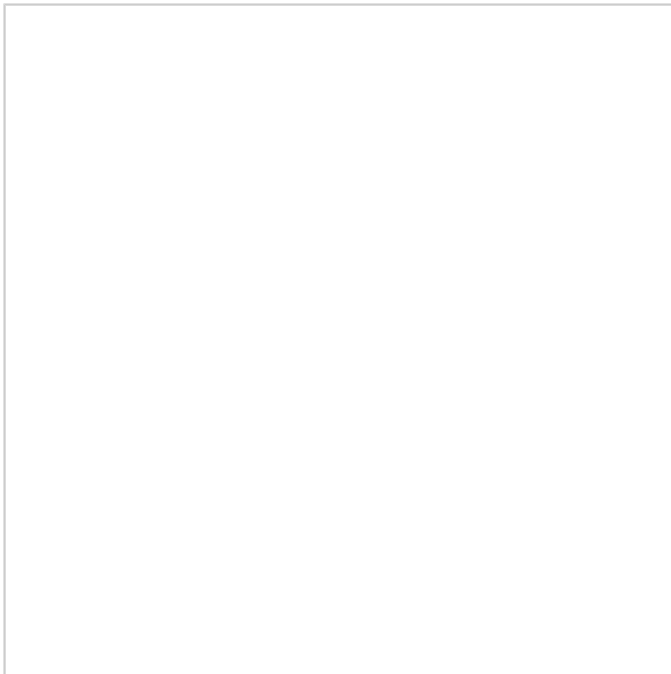
```
g.setColor(colorRosa);
```

No es necesario tener a mano la tabla de las componentes RGB de cada color. La clase *Color* nos proporciona un conjunto de colores predefinidos en forma de [miembros estáticos](#) de dicha clase. Podemos escribir alternativamente

```
g.setColor(Color.pink);
```

Los colores predefinidos son los siguientes

| | | |
|------------------------|---------------------|----------------------|
| <i>Color.white</i> | <i>Color.black</i> | <i>Color.yellow</i> |
| <i>Color.lightGray</i> | <i>Color.red</i> | <i>Color.green</i> |
| <i>Color.gray</i> | <i>Color.pink</i> | <i>Color.magenta</i> |
| <i>Color.darkGray</i> | <i>Color.orange</i> | <i>Color.cyan</i> |
| <i>Color.blue</i> | | |



El color de fondo del componente se establece con *setBackground* y se obtiene con *getBackground*. En el siguiente applet observamos cómo se utiliza esta segunda función para crear una diana. En la función *init* establecemos el color de fondo en blanco mediante *setBackground*. En la función miembro *paint* obtenemos el color de fondo mediante *getBackground*. Los círculos se pintan de mayor a menor radio. Se pinta un círculo de color rojo y se borra parte de su interior con el color de fondo, de este modo se crea un anillo, luego otro y así sucesivamente, hasta completar cuatro anillos de color rojo con la apariencia de una diana.

En el programa, también podemos apreciar que la función *paint* suministra el contexto gráfico *g* del componente (applet) en el cual podemos dibujar. El objeto *g* llama a *setColor* para establecer el color, y a *fillOval* para dibujar un círculo pintado de dicho color.

La función *getSize* nos devuelve el tamaño del componente (applet), de modo que los diámetros de la elipse mayor son respectivamente la anchura y altura del applet.

```
package paint1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class PaintApplet extends Applet {
    public void init(){
        setBackground(Color.white);
    }
    public void paint(Graphics g) {
        int x, y, ancho, alto;
        int appletAlto = getSize().height;
        int appletAncho = getSize().width;

        for (int i=8; i>=0; i--) {
            if ((i % 2)==0) g.setColor(Color.red);
            else g.setColor(getBackground());
            alto = appletAlto*i/8;
            ancho = appletAncho*i/8;
            x=appletAncho/2-i*appletAncho/16;
            y=appletAlto/2-i*appletAlto/16;
            g.fillOval(x, y, ancho, alto);
        }
    }
}
```

La clase *Font*



font2: [FontApplet2.java](#)

Para crear una fuente de texto u objeto de la clase *Font* llamamos a su constructor, y le pasamos el nombre de la fuente de texto, el estilo y el tamaño. Por ejemplo,

```
Font fuente=new Font("TimesRoman", Font.BOLD, 12);
```

Esta sentencia, crea una fuente de texto Times Roman, en letra negrita, de 12 puntos.

Los estilos vienen dados por constantes ([miembros estáticos](#) de la clase *Font*), *Font.BOLD* establece el estilo negrita, *Font.ITALIC*, el estilo cursiva, y *Font.PLAIN*, el estilo normal. Se pueden combinar las constantes *Font.BOLD*+*Font.ITALIC* para establecer el estilo negrita y cursiva a la vez.

La función *setFont* de la clase *Graphics* establece la fuente de texto en el contexto gráfico *g*.

```
g.setFont(fuente);
```

La función *getFont* obtiene la fuente de texto actual de dicho contexto gráfico. La función *drawString* dibuja el string guardado en el objeto *texto* de la clase *String*, y lo sitúa en la posición cuyas coordenadas vienen dadas por los dos números enteros que le siguen.

En la siguiente porción de código, establecemos una fuente de texto, dibujamos el texto, y reestablecemos la fuente de texto por defecto, una operación habitual que se realiza al programar un applet.

```
Font oldFont=getFont();
Font fuente=new Font("Monospaced", Font.BOLD, 36);
g.setFont(fuente);
g.drawString(texto, 100, 50);
g.setFont(oldFont);
g.drawString(otroTexto, 100, 70);
```

Para obtener el nombre de las fuentes de texto disponibles se escribe el siguiente código

```
String[] nombreFuentes=getToolkit().getFontList();
for(int i=0; i<nombreFuentes.length; i++){
    System.out.println(nombreFuentes[i]);
}
```

La clase *FontMetrics*

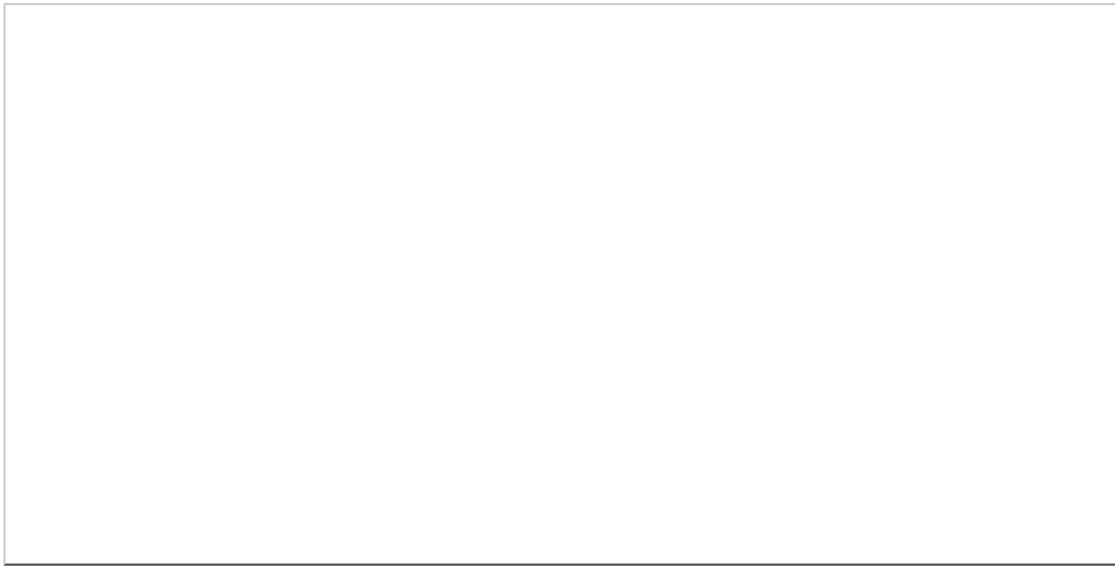
La clase *FontMetrics* nos permite conocer las características de una fuente de texto.

Desde el contexto gráfico *g*, llamamos a la función *getFontMetrics* para obtener un objeto *fm* de la clase *FontMetrics* que nos describe las características de una fuente determinada o de la fuente actualmente seleccionada. En el primer caso escribimos

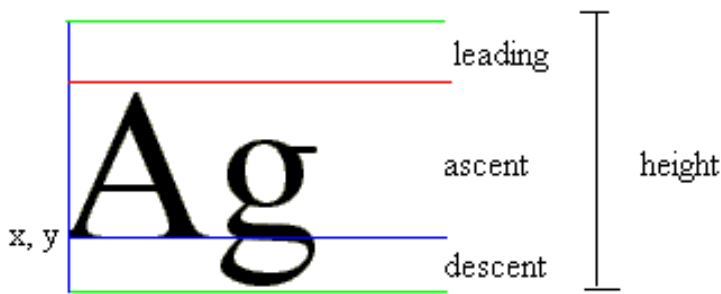
```
Font fuente=new Font("Dialog", Font.BOLD, 36);
FontMetrics fm=g.getFontMetrics(fuente);
```

En el segundo caso, escribimos

```
Font fuente=new Font("Courier", Font.BOLD, 36);
g.setFont(fuente);
FontMetrics fm=g.getFontMetrics();
```



En el applet mostramos las características de una fuente de texto: *Ascent* es la distancia entre línea horizontal de color azul (baseline) y la línea horizontal de color rojo. *Descent* es la distancia entre la línea horizontal de color azul (baseline) y la línea horizontal de color verde. *Leading* sería la distancia entre las línea de color verde (descent) y la línea roja (ascent) de la siguiente línea de texto..



Para obtener la altura de una fuente de texto, llamamos a la función *getHeight* miembro de *FontMetrics*. La altura de una fuente de texto, es la distancia entre dos líneas base (baseline) consecutivas, y es la suma de el *ascent*, *descent* y *leading*.

Tres funciones que comienzan por *get* devuelven los valores de estos tres atributos de una fuente de texto: *getAscent*, *getDescent*, y *getLeading*.

Para escribir dos líneas de texto, una debajo de otra escribimos

```
FontMetrics fm=g.getFontMetrics();
String texto="La cigüeña vendrá";
g.drawString(texto, 10, 50);
int hFont=fm.getHeight();
texto=new String("serÁ en primavera");
g.drawString(texto, 10, 50+hFont);
```

La primera línea de texto, se sitúa en el punto (10, 50), la ordenada 50, señala la posición vertical de la línea base de la fuente de texto, véase la figura. La segunda línea de texto tiene una línea base cuya posición vertical se obtiene sumando a 50 la altura *hFont* de la fuente de texto.

Otro valor interesante, es la anchura de un texto, que se obtiene mediante la función miembro *stringWidth*, y se le pasa el texto. Por ejemplo, para centrar horizontalmente un texto en el applet escribimos.

```
String texto="La cigüeña vendrá";
int ancho=fm.stringWidth(texto);
g.drawString(texto, (anchoApplet-ancho)/2, 50);
```

La función `getSize().width` obtiene la anchura del componente (applet), y la variable *ancho*, guarda la anchura del string *texto*.

Un poco más difícil es centrar un texto verticalmente, en una determinada posición. Teniendo en cuenta, que las coordenadas que se le pasan a la función *drawString* se refieren a la línea base del primer carácter, tal como se ve en la figura. La fórmula de centrado vertical en un punto de ordenada y sería: la ordenada y de la línea base menos *descent* más la mitad de la altura de los caracteres *hFont*. Se ha de tener en cuenta que la ordenada y aumenta de arriba hacia abajo.

```
g.drawLine(0, y, anchoApplet, y);
g.setColor(Color.red);
texto="Centrado: a, p, ñ, á, Á ";
g.drawString(texto, 10, y+hFont/2-descent);
```

Como los caracteres pueden estar o no acentuados, escritos en mayúsculas o minúsculas, etc, la fórmula para mostrar un texto centrado verticalmente no es única, se sugiere probar estas dos dadas por otros autores

```
g.drawString(texto, 10, y+ascent/4);
g.drawString(texto, 10, y-hFont/2+ascent);
```

El código completo de este ejemplo es, el siguiente

```
package fonts2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class FontApplet2 extends Applet {
    public void init() {
        setBackground(Color.white);
    }
    public void paint(Graphics g){
        int anchoApplet=getSize().width;
        Font oldFont=getFont();
        Font fuente=new Font("Monospaced", Font.BOLD, 36);
        g.setFont(fuente);
        FontMetrics fm=g.getFontMetrics();
        String texto="La cigüeña vendrá";
        int ancho=fm.stringWidth(texto);
        int y=50;
        g.drawString(texto, (anchoApplet-ancho)/2, y);
        texto=new String("serÁ en primavera");
        ancho=fm.stringWidth(texto);
        //características de las fuentes de texto
        int hFont=fm.getHeight();
        int ascent=fm.getAscent();
        int descent=fm.getDescent();
        int leading=fm.getLeading();
        g.drawString(texto, (anchoApplet-ancho)/2, y+hFont);
        //dibuja línea base
        g.setColor(Color.blue);
        g.drawLine(0, y, getSize().width, y);
```

```

        g.drawLine(0, y+hFont, anchoApplet, y+hFont);
//dibuja ascent
        g.setColor(Color.red);
        g.drawLine(0, y-ascent, anchoApplet, y-ascent);
        g.drawLine(getSize().width/2, y+hFont-ascent, anchoApplet, y+hFont-ascent);
//dibuja descent
        g.setColor(Color.green);
        g.drawLine(0, y+descent, anchoApplet/2, y+descent);
        g.drawLine(0, y+hFont+descent, anchoApplet, y+hFont+descent);

//texto centrado verticalmente en la posición y
        y+=2*hFont;
        g.setColor(Color.black);
        g.drawLine(0, y, anchoApplet, y);
        g.setColor(Color.red);
        texto="Centrado: a, p, ñ, 5, Á ";
        g.drawString(texto, 10, y+hFont/2-descent);

//Escribe tres líneas de texto en la fuente de texto por defecto.
        g.setFont(oldFont);
        fm=g.getFontMetrics();
        hFont=fm.getHeight();

        y+=3*hFont;
        g.setColor(Color.black);
        texto="leading =" +leading;
        g.drawString(texto, 10, y);

        texto="ascent =" +ascent;
        y+=hFont;
        g.drawString(texto, 10, y);

        texto="descent =" +descent;
        y+=hFont;
        g.drawString(texto, 10, y);

        texto="altura=ascent+descent+leading= " +(ascent+descent+leading);
        y+=hFont;
        g.drawString(texto, 10, y);
    }
}

```

La clase *Vector*

[Las clases *Vector* y *StringTokenizer*](#)



[Crear un vector](#)

[Añadir elementos al vector](#)

[Acceso a los elementos de un vector](#)

Los arrays en Java son suficientes para [guardar tipos básicos de datos](#), y [objetos de una determinada clase](#) cuyo número conocemos de antemano. Algunas veces deseamos guardar objetos en un array pero no sabemos cuantos objetos vamos a guardar. Una solución es la de crear un array cuya dimensión sea más grande que el número de elementos que necesitamos guardar. La clase *Vector* nos proporciona una solución alternativa a este problema. Un vector es similar a un array, la diferencia estriba en que un vector crece automáticamente cuando alcanza la dimensión inicial máxima. Además, proporciona métodos adicionales para añadir, eliminar elementos, e insertar elementos entre otros dos existentes.



vector: [VectorApp.java](#)

Crear un vector

Para usar la clase *Vector* hemos de poner al principio del archivo del código fuente la siguiente [sentencia import](#)

```
import java.util.*;
```

Cuando creamos un *vector* u objeto de la clase *Vector*, podemos especificar su dimensión inicial, y cuanto crecerá si rebasamos dicha dimensión.

```
Vector vector=new Vector(20, 5);
```

Tenemos un vector con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25.

Al segundo constructor, solamente se le pasa la dimensión inicial.

```
Vector vector=new Vector(20);
```

Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica. El programador ha de tener cuidado con este constructor, ya que si se pretende guardar un número grande de elementos se tiene que especificar el incremento de la capacidad del vector, si no se quiere desperdiciar inútilmente la memoria el ordenador.

Con el tercer constructor, se crea un vector cuya dimensión inicial es 10.

```
Vector vector=new Vector();
```

La dimensión del vector se duplica si se rebasa la dimensión inicial, por ejemplo, cuando se pretende guardar once elementos.

Añadir elementos al vector

Hay dos formas de añadir elementos a un vector. Podemos añadir un elemento a continuación del último elemento del vector, mediante la función miembro *addElement*.

```
v.addElement("uno");
```

Podemos también insertar un elemento en una determinada posición, mediante *insertElementAt*. El segundo parámetro o índice, indica el lugar que ocupará el nuevo objeto. Si tratamos de insertar un elemento en una posición que no existe todavía obtenemos una [excepción](#) del tipo *ArrayIndexOutOfBoundsException*. Por ejemplo, si tratamos de insertar un elemento en la posición 9 cuando el vector solamente tiene cinco elementos.

Para insertar el string "tres" en la tercera posición del vector *v*, escribimos

```
v.insertElementAt("tres", 2);
```

En la siguiente porción de código, se crea un vector con una capacidad inicial de 10 elementos, valor por

defecto, y se le añaden o insertan objetos de la clase *String*.

```
Vector v=new Vector();
v.addElement("uno");
v.addElement("dos");
v.addElement("cuatro");
v.addElement("cinco");
v.addElement("seis");
v.addElement("siete");
v.addElement("ocho");
v.addElement("nueve");
v.addElement("diez");
v.addElement("once");
v.addElement("doce");
v.insertElementAt("tres", 2);
```

Para saber cuantos elementos guarda un vector, se llama a la función miembro *size*. Para saber la dimensión actual de un vector se llama a la función miembro *capacity*. Por ejemplo, en la porción de código hemos guardado 12 elementos en el vector *v*. La dimensión de *v* es 20, ya que se ha superado la dimensión inicial de 10 establecida en la llamada al tercer constructor cuando se ha creado el vector *v*.

```
System.out.println("nº de elementos "+v.size());
System.out.println("dimensión "+v.capacity());
```

Podemos eliminar todos los elementos de un vector, llamando a la función miembro *removeAllElements*. O bien, podemos eliminar un elemento concreto, por ejemplo el que guarda el string "tres".

```
v.removeElement("tres");
```

Podemos eliminar dicho elemento, si especificamos su índice.

```
v.removeElementAt(2);
```

Acceso a los elementos de un vector

El acceso a los elementos de un vector no es tan sencillo como el acceso a los elementos de un array. En vez de dar un índice, usamos la función miembro *elementAt*. Por ejemplo, *v.elementAt(4)* sería equivalente a *v[4]*, si *v* fuese un array.

Para acceder a todos los elementos del vector, escribimos un código semejante al empleado para acceder a todos los elementos de un [array](#).

```
for(int i=0; i<v.size(); i++){
    System.out.print(v.elementAt(i)+"\t");
}
```

Existe otra alternativa, que es la de usar las funciones del [interface](#) *Enumeration*. Este interface declara dos funciones pero no implementa ninguna de ellas. Una *Enumeration* nos permite acceder a los elementos de una estructura de datos de forma secuencial.

```
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

La función miembro *elements* de la clase *Vector* devuelve un objeto de la clase *VectorEnumerator* que implementa el interface *Enumeration* y tiene que definir las dos funciones *hasMoreElements* y *nextElement*.

```
final class VectorEnumerator implements Enumeration {
    Vector vector;
    int count;
    VectorEnumerator(Vector v) {
        vector = v;
        count = 0;
    }
    public boolean hasMoreElements() {
        //...
    }
    public Object nextElement() {
        //...
    }
}
```

El objeto *enum* devuelto por la función miembro *elements* es de la clase *VectorEnumerator*, sin embargo no podemos escribir

```
VectorEnumerator enum=v.elements();
```

porque *VectorEnumerator* no es una clase pública. Como podemos ver en su definición, no tiene la palabra reservada **public** delante de **class**. Sin embargo, podemos guardar un objeto de la clase

VectorEnumerator en una variable *enum* del tipo *Enumeration*, por que la clase implementa dicho interface.

```
Enumeration enum=v.elements();
while(enum.hasMoreElements()){
    System.out.print(enum.nextElement()+"\t");
}
```

Desde el objeto *enum* devuelto por la función miembro *elements* de la clase *Vector* llamamos a las funciones miembro *hasMoreElements* y *nextElement* de la clase *VectorEnumerator*. La función *hasMoreElements* devuelve **true** mientras haya todavía más elementos que se puedan acceder en el vector *v*. Cuando se ha llegado al último elemento del vector, devuelve **false**. La función *nextElement* devuelve una referencia al próximo elemento en la estructura de datos. Esta función devuelve una referencia a un objeto de la clase base *Object*, que el programador precisará en ciertos casos, como veremos más abajo, promocionar (casting) a la clase adecuada.

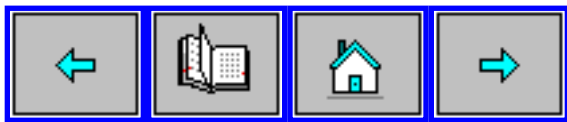
Para buscar objetos en un vector se puede usar una *Enumeration* y hacer una comparación elemento por elemento mediante [*equals*](#), tal como vemos en la siguiente porción de código

```
Enumeration enum=v.elements();
while(enum.hasMoreElements()){
    String elemento=(String)enum.nextElement();
    if(elemento.equals("tres")){
        System.out.println("Encontrado tres");
        break;
    }
}
```

Podemos usar alternativamente, la función miembro *contains* para este propósito.

```
if(v.contains("tres")){
    System.out.println("Encontrado tres");
}
```

La clase *StringTokenizer*



[Las clases Vector y StringTokenizer](#)

[Los constructores](#)

[Obtención de tokens](#)

La clase *StringTokenizer* nos ayuda a dividir un string en substrings o tokens, en base a otro string (normalmente un carácter) separador entre ellos denominado delimitador.

Supongamos un string consistente en el nombre, y los dos apellidos de una persona separados por espacios en blanco. La clase *StringTokenizer* nos ayuda a romper dicho string en tres substrings basado en que el carácter delimitador es un espacio en blanco.



tokens: [TokenApp.java](#)

Un control área de texto, permite varias líneas de texto, cada línea está separada de la siguiente mediante un carácter nueva línea '**\n**' que se obtiene pulsando la tecla Enter o Retorno. Mediante una función denominada *getText* obtenemos todo el texto que contiene dicho control. La clase *StringTokenizer* nos permite dividir el string obtenido en un número de substrings o tokens igual al número de líneas de texto, basado en que el carácter delimitador es '**\n**'.

Para usar la clase *StringTokenizer* tenemos que poner al principio del archivo del código fuente la siguiente [sentencia import](#).

```
import java.util.*;
```

o bien

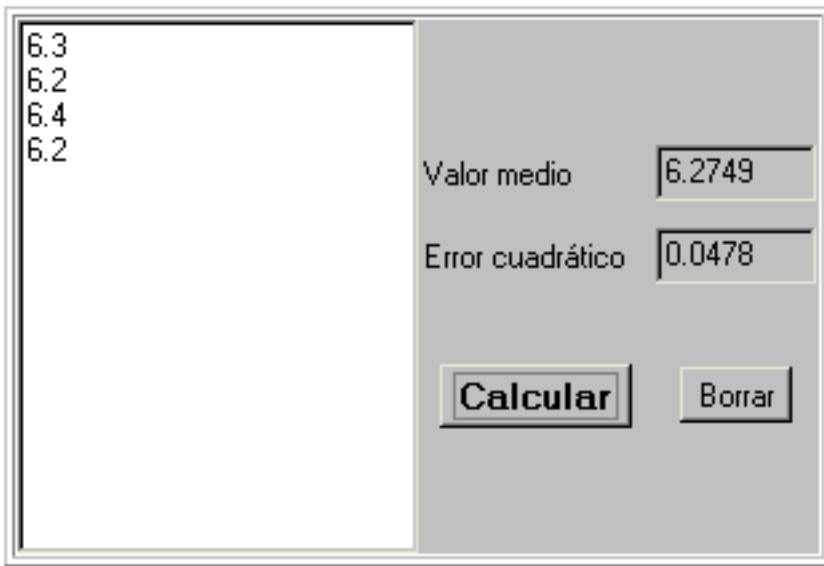
```
import java.util.StringTokenizer;
```

Los constructores

Creamos un objeto de la clase *StringTokenizer* llamando a uno de los tres constructores que tiene la clase. Al primer constructor, se le pasa el string *nombre* que va a ser dividido teniendo en cuenta que el espacio en blanco es el delimitador por defecto.

```
String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
```

Supongamos ahora que en [un control área de texto](#) introducimos los siguientes datos, resultado de ciertas medidas, tal como se ve a la izquierda en la figura.



Obtenemos el texto del control área de texto. Creamos un objeto *tokens* de la clase *StringTokenizer*, pasándole el string *strDatos* y el delimitador `"\n"`

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
```

Obtención de los tokens

La clase *StringTokenizer* implementa el [interface Enumeration](#), por tanto define las funciones *nextElement* y *hasMoreElements*.

```
public class StringTokenizer implements Enumeration {
```

```

        //...
    public boolean hasMoreElements() {
        //...
    }
    public Object nextElement() {
        //...
    }
}

```

Para el programador es más cómodo usar las funciones miembro equivalentes *nextToken* y *hasMoreTokens*. Para extraer el nombre, el primer apellido y el segundo apellido en el primer ejemplo, escribiremos

```

String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
while(tokens.hasMoreTokens()){
    System.out.println(tokens.nextToken());
}

```

El segundo ejemplo, requiere un poco más de trabajo, ya que además de extraer los tokens del string *strDatos*, hemos de convertir cada uno de los substrings en un valor numérico de tipo **double** y guardarlos en el array *datos* del mismo tipo. Véase la sección [convertir un string en un valor numérico](#).

```

String str=tokens.nextToken();
datos[i]=Double.valueOf(str).doubleValue();

```

El número de tokens o de datos *nDatos* que hay en un string *strDatos*, se obtiene mediante la función miembro *countTokens*. Con este dato establecemos la dimensión del array *datos*.

```

int nDatos=tokens.countTokens();
double[] datos=new double[nDatos];

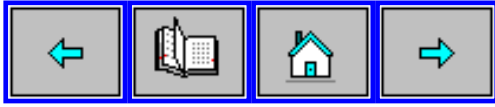
```

Una vez guardados los datos introducidos en el control área de texto en el array *datos*, podemos operar con ellos, obteniendo su valor medio, y el error cometido en las medidas efectuadas.

El código completo para extraer los tokens del string *strDatos* y guardarlos en un array *datos*, es el siguiente.

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
int nDatos=tokens.countTokens();
double[] datos=new double[nDatos];
int i=0;
while(tokens.hasMoreTokens()){
    String str=tokens.nextToken();
    datos[i]=Double.valueOf(str).doubleValue();
    System.out.println(datos[i]);
    i++;
}
```

Los gestores *FlowLayout*, *BorderLayout* y *GridLayout*



Gestores de diseño

[La paleta de componentes](#)

[Añadir componentes al applet](#)

[El gestor *FlowLayout*](#)

[El gestor *BorderLayout*](#)

[El gestor *GridLayout*](#)

El Entorno Integrado de Desarrollo (IDE) JBuilder dispone de un modo de diseño, similar a otras herramientas de programación de programas gráficos (GUI). En la superficie del applet se disponen los componentes, como paneles y controles. Para ello, se selecciona con el ratón un componente en la paleta correspondiente y se sitúa sobre el applet o sobre otro componente. Ahora bien, la herramienta de diseño no dispone de opciones para situar los componentes en lugares precisos, alinearlos, etc, como ocurre en Windows, ya que en Java existen los denominados gestores de diseño que no sitúan los componentes en posiciones absolutas, sino que la disposición se determina mediante un algoritmo. El más simple de los gestores de diseño es *FlowLayout* y el más complicado es *GridBagLayout*.

Para el programador acostumbrado a diseñar ventanas y diálogos en el entorno Windows le parecerá extraña esta forma de proceder y pensará que con este sistema será difícil elaborar un interfaz gráfico de usuario.

Como veremos, se puede crear un diseño complejo mediante el gestor *GridBagLayout* y también, mediante la aproximación de paneles anidados. En la parte del curso dedicada a [estudiar ejemplos completos](#) se describen ampliamente estas dos aproximaciones.

No obstante, el programador debe percibir la diferencia entre applets y aplicaciones. Los applets están insertados en una página web y se ejecutan en la ventana del navegador. El applet comparte espacio con texto, imágenes y otros elementos multimedia. El usuario tiene la libertad de moverse por la página, y por otras páginas a través de los enlaces. La percepción que tiene el usuario del applet es completamente distinta de la percepción que tiene de una aplicación que llama completamente su atención.

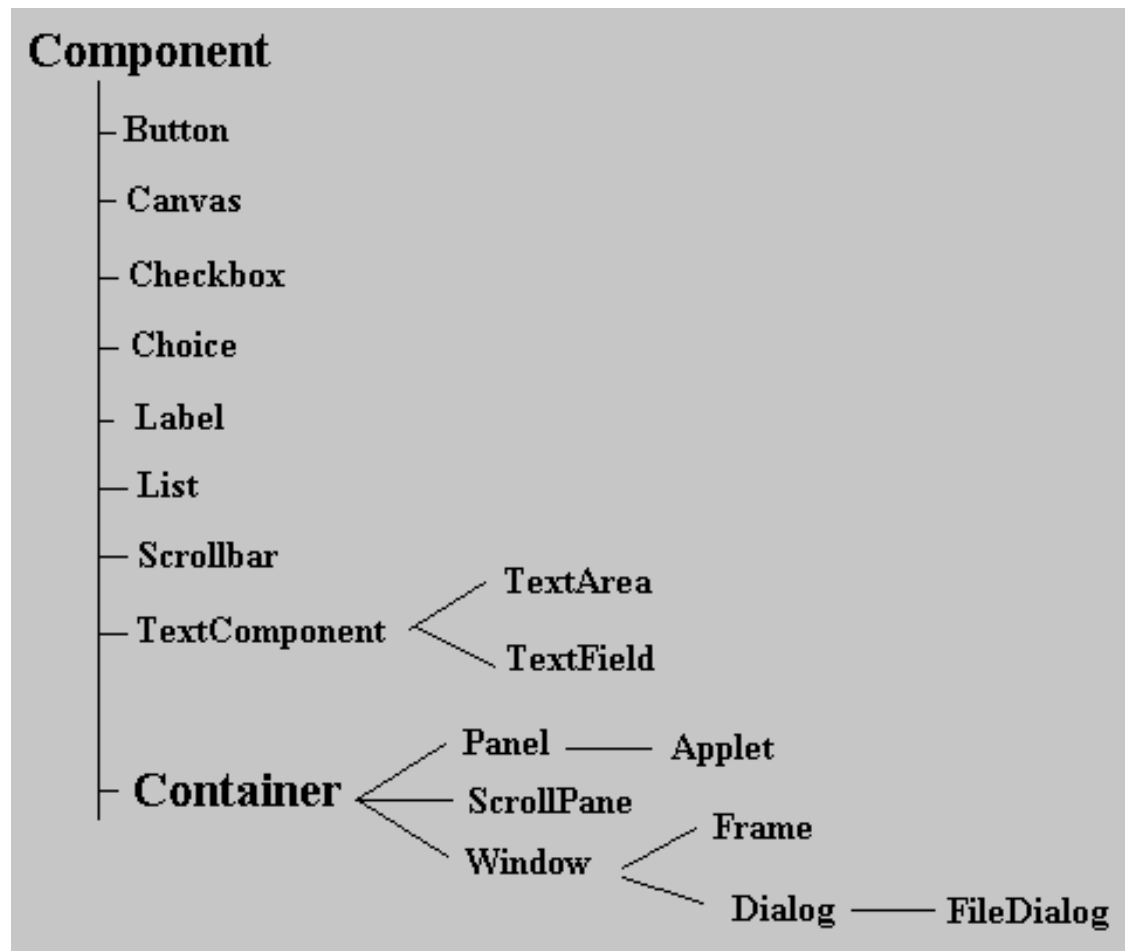
El programador debe tener en cuenta estas diferencias de percepción, y debe esforzarse en crear un diseño de modo que el usuario encuentre evidente el manejo del applet a primera vista.

La paleta de componentes

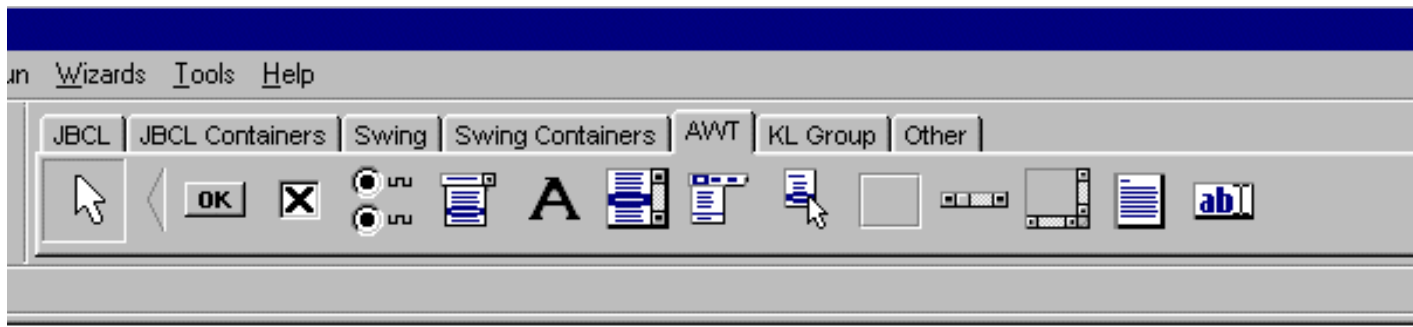
Las paletas de componentes están situados en la ventana superior del IDE. Observamos en la paleta que hay dos tipos de componentes:

- los controles
- los paneles

Los controles derivan de la clase *Component*, y los paneles derivan de la clase *Container*. Los paneles (un applet es un panel especializado) pueden contener otros componentes (paneles y controles). La jerarquía de clases se muestra en la figura



La librería AWT no es muy rica en componentes, tal como se ve en la figura, por lo que JBuilder viene acompañado por otras paletas de componentes. Sin embargo, hemos de tener cuidado en usar solamente componentes AWT para los applets que vayamos a publicar en Internet. Los navegadores no ejecutan applets que utilicen componentes no estándar.

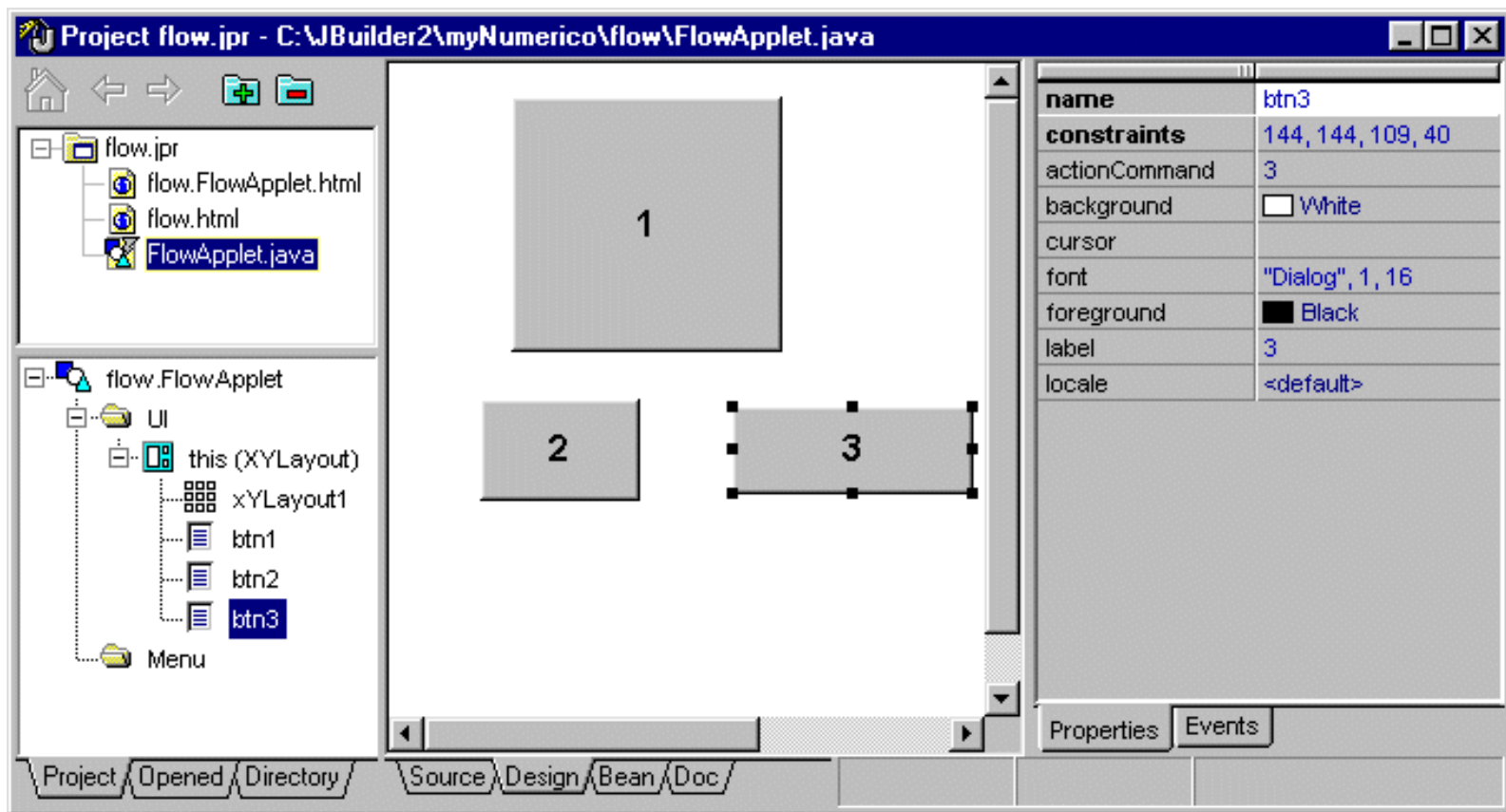


Por debajo de los controles AWT están los controles nativos, esto presenta algunas dificultades, por ejemplo, los controles son siempre rectangulares y opacos. La versión Java 2.0 sustituye la librería AWT por la denominada Swing en la que se elimina la dependencia de los controles nativos, el número y el tipo de componentes puede satisfacer con creces las aspiraciones de cualquier programador exigente. Además, podemos crear nuestros propios controles, comprarlos o encargarnos a medida, y situarlos en la paleta de componentes. En el capítulo dedicado a la tecnología de componentes o [javaBeans](#) nos ocuparemos de este asunto.

Añadir componentes al applet

Una vez que hemos [creado el applet](#) con el asistente, se pulsa con el ratón la pestaña **Design** situada en la parte inferior del panel de contenido. Al seleccionar esta pestaña, no cambia el panel de navegación, pero cambia el panel de estructura (inferior izquierda) y el panel de contenido (a la derecha). El panel de estructura nos muestra los componentes **this** (el applet) y los botones que se han situado sobre el applet.

Nos aseguraremos que el applet (**this**) tenga establecido el gestor de diseño *XYLayout*. Este gestor permite el posicionamiento absoluto de los componentes, sirviendo de gran ayuda en la etapa de diseño. Una vez terminado, hemos de cambiarlo por los gestores estándar que estudiamos en estas páginas.



Cuando se selecciona un componente, por ejemplo un botón, aparece su hoja de propiedades a la derecha. Dicha hoja contiene una lista de propiedades del componente y al lado un editor asociado. Por ejemplo, la propiedad **name** es el nombre con el que se conoce al botón en el código fuente. Podemos cambiar el valor por defecto y poner un nombre significativo. Para los controles se acostumbra poner un nombre que consta de dos partes:

- El prefijo indica el tipo de control. Por ejemplo, **btn** indica botón, **ch**, caja de selección, **list**, lista etc.
- El sufijo indica la funcionalidad o el título del control, *btnAceptar*, para el botón Aceptar, *btnCancelar*, etc.

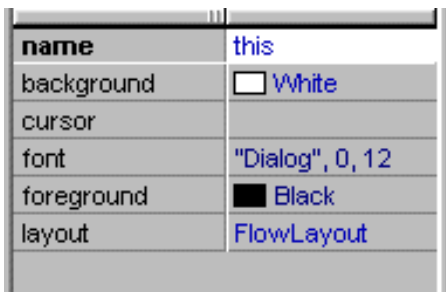
La propiedad **font** tiene un editor asociado que es un diálogo en el que podemos elegir el tipo de fuente, el tamaño, y el estilo. En este caso elegimos Dialog, 16 puntos, estilo negrita (bold).

La propiedad **label** tiene un editor asociado en el que podemos cambiar el título del botón.

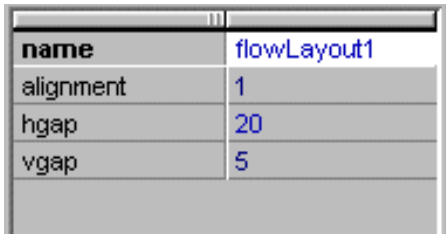
El gestor *FlowLayout*

flow: [FlowApplet.java](#)

Vamos al panel de estructura y situamos el cursor en **this** (el applet) y observamos su hoja de propiedades. La propiedad **layout** tiene un editor asociado que es una caja de selección, elegimos el elemento *FlowLayout*. Veremos como los botones se alinean en el centro y en la parte superior del applet.

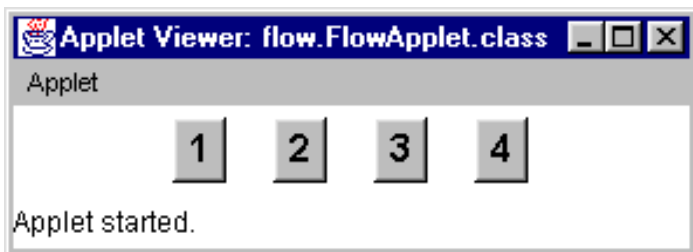


Volvemos al panel de estructura y situamos el cursor en *flowLayout1*, aparece su hoja de propiedades

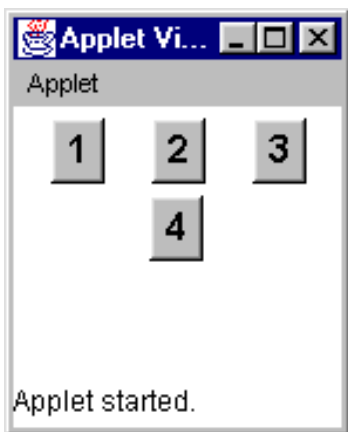


Podemos cambiar su nombre por defecto, en el editor asociado a la propiedad **name**, el alineamiento (0 es a la izquierda, 1 al centro y 2 a la derecha) y el espaciado horizontal y vertical entre controles, en los editores asociados a las propiedades **alignment**, **hgap** y **vgap**. En este caso hemos elegido un alineamiento en el centro (por defecto) y un espaciado horizontal de 20 y un espaciado vertical de 5 (por defecto).

FlowLayout es un gestor que pone los controles en una línea, como puede verse en la figura



Si se cambia el tamaño del applet y los controles no caben en una línea, pasan a la línea siguiente, como puede verse en la figura.



El código fuente

Si pulsamos con el ratón en la pestaña titulada **Source**, vemos el código fuente que ha generado el IDE.

Los controles son objetos de la clase *Button*, y el gestor de diseño es un objeto de la clase *FlowLayout*. Una vez inicializados los miembros dato, en la función miembro *init* se establecen sus propiedades y se añaden al applet mediante la función *add*, una vez establecido el gestor de diseño mediante *setLayout*. Los pasos son los siguientes

1. Crear los botones (objetos de la clase *Button*) y el gestor de diseño (objeto de la clase *FlowLayout*)

```
Button btn1 = new Button();
FlowLayout flowLayout1 = new FlowLayout();
```

2. Establecer sus propiedades en *init*

```
btn1.setFont(new Font("Dialog", 1, 16));
btn1.setLabel("1");
flowLayout1.setHgap(20);
```

3. Establecer el gestor de diseño del applet (o de un panel) mediante *setLayout*

```
this.setLayout(flowLayout1);
```

4. Añadir los controles al applet (o a un panel) mediante *add*

```
this.add(btn1, null);
```

Lo que se ha dicho para un applet vale para cualquier panel, ya que un applet no es otra cosa que un panel especializado.

```
public class FlowApplet extends Applet {
    Button btn1 = new Button();
    Button btn2 = new Button();
    Button btn3 = new Button();
    Button btn4 = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init(){
        setBackground(Color.white);
        btn1.setFont(new Font("Dialog", 1, 16));
        btn1.setLabel("1");
        btn2.setFont(new Font("Dialog", 1, 16));
        btn2.setLabel("2");
        btn3.setFont(new Font("Dialog", 1, 16));
        btn3.setLabel("3");
        btn4.setFont(new Font("Dialog", 1, 16));
        btn4.setLabel("4");
        flowLayout1.setHgap(20);
    }
}
```

```
        this.setLayout(flowLayout1);  
        this.add(btn1, null);  
        this.add(btn2, null);  
        this.add(btn3, null);  
        this.add(btn4, null);  
    }  
}
```

El gestor *BorderLayout*



border: [BorderApplet.java](#)

Los pasos para establecer el gestor *BorderLayout* son distintos a los empleados para el gestor *FlowLayout*.

1. Crear los botones (objetos de la clase *Button*) y el gestor de diseño (objeto de la clase *BorderLayout*)

```
Button btnOeste = new Button();  
BorderLayout borderLayout1 = new BorderLayout();
```

2. Establecer sus propiedades en *init*

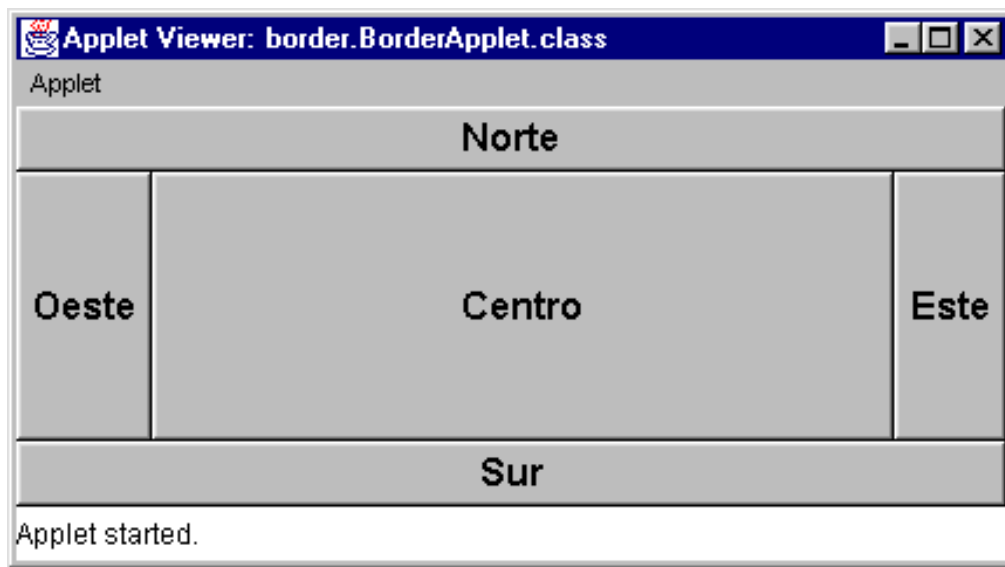
```
btnOeste.setFont(new Font("Dialog", 1, 16));  
btn1.setLabel("Oeste");
```

3. Añadir los controles al applet (o a un panel) mediante *add*, indicando en el segundo argumento la posición que ocupará cada control en el panel mediante miembros estáticos de la clase *BorderLayout*.

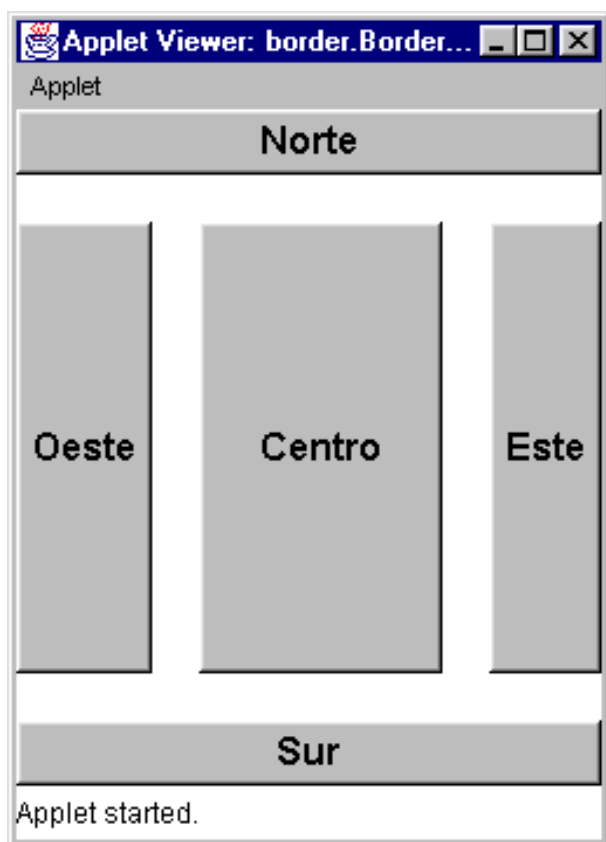
```
this.add(btnOeste, BorderLayout.WEST);
```

En el código fuente se ha marcado en letra negrita las diferencias y semejanzas entre los distintos gestores.

En este ejemplo, se han creado cinco botones cuyos títulos son *Oeste*, *Norte*, *Sur*, *Este* y *Centro*. Cuando se aplica el gestor *BorderLayout* al applet los cinco botones se disponen como se muestra en la figura. Si alguno de los botones no está en la posición correcta se puede arrastrar con el ratón a la posición adecuada.



Vamos al panel de estructura y situamos el cursor sobre el gestor *borderLayout1*. En su hoja de propiedades podemos establecer un espaciado entre los botones tal como se muestra en la figura. Con este gestor de diseño los botones ocupan completamente el panel. Cuando se cambia la dimensiones del applet los botones cambian su tamaño para adaptarse a las nuevas dimensiones del panel, tal como puede verse en la figura inferior.



```

public class BorderApplet extends Applet {
    Button btnOeste = new Button();
    Button btnEste = new Button();
    Button btnNorte = new Button();
    Button btnSur = new Button();
    Button btnCentro = new Button();
    BorderLayout BorderLayout1 = new BorderLayout();

    public void init() {
        setBackground(Color.white);
        this.setSize(new Dimension(336, 253));
        this.setLayout(BorderLayout1);
        btnOeste.setFont(new Font("Dialog", 1, 16));
        btnOeste.setLabel("Oeste");
        btnEste.setFont(new Font("Dialog", 1, 16));
        btnEste.setLabel("Este");
        btnNorte.setFont(new Font("Dialog", 1, 16));
        btnNorte.setLabel("Norte");
        btnSur.setFont(new Font("Dialog", 1, 16));
        btnSur.setLabel("Sur");
        btnCentro.setFont(new Font("Dialog", 1, 16));
        btnCentro.setLabel("Centro");
        BorderLayout1.setVgap(20);
        BorderLayout1.setHgap(20);
        this.add(btnOeste, BorderLayout.WEST);
        this.add(btnEste, BorderLayout.EAST);
        this.add(btnNorte, BorderLayout.NORTH);
        this.add(btnSur, BorderLayout.SOUTH);
        this.add(btnCentro, BorderLayout.CENTER);
    }
}

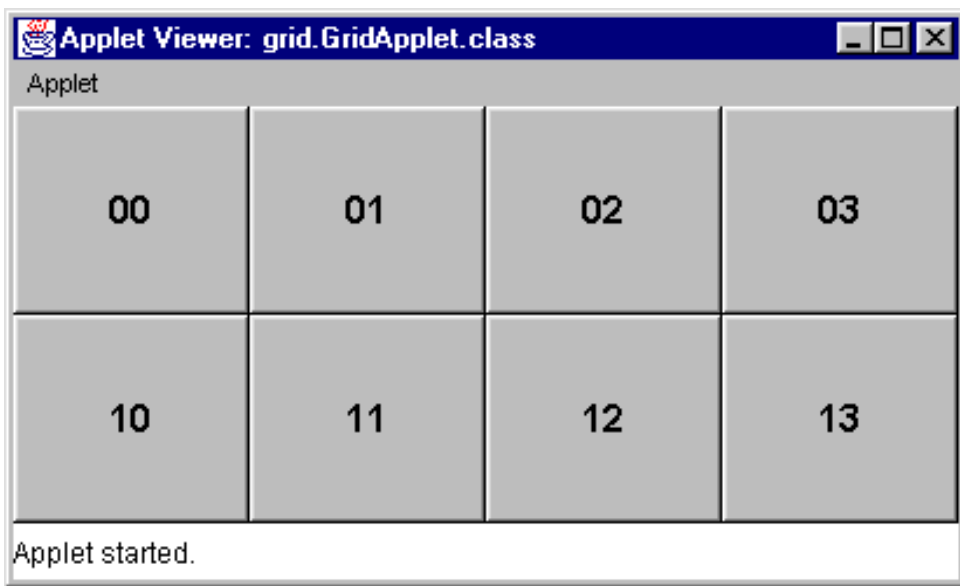
```

El gestor *GridLayout*

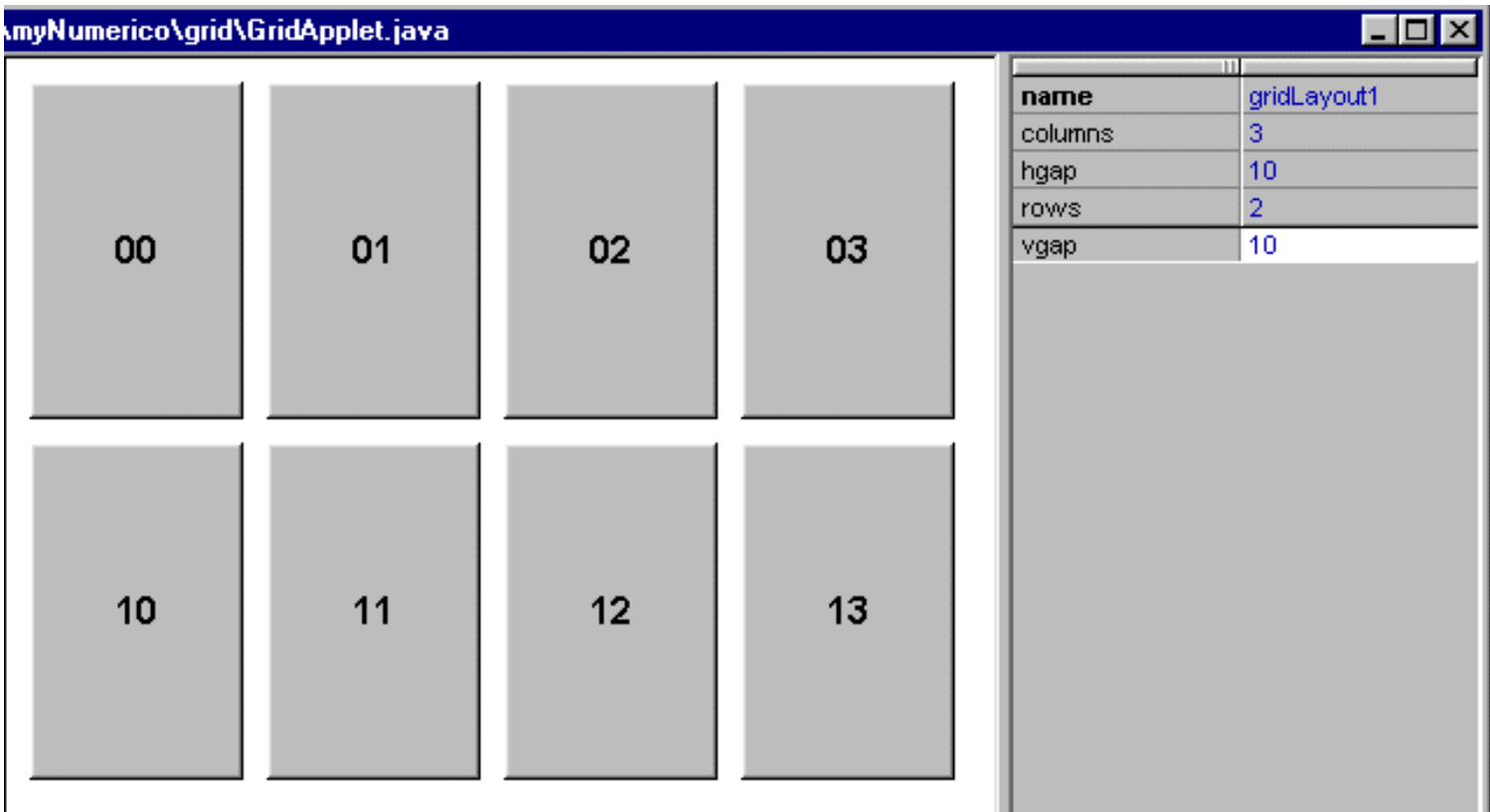


grid: [GridApplet.java](#)

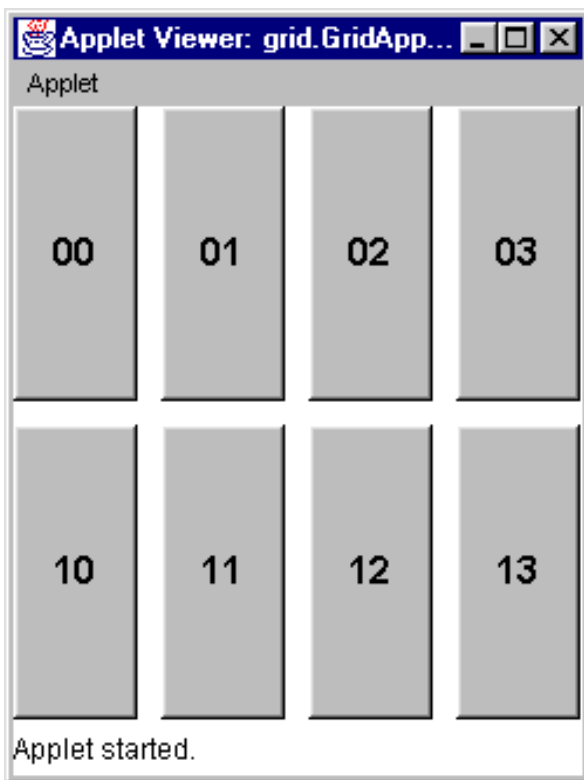
Los pasos para establecer el gestor *GridLayout* son idénticos a los que hemos seguido para establecer el gestor *FlowLayout*. Este gestor dispone los controles en forma de una matriz tal como puede verse en la figura. Tenemos ocho botones dispuestos en dos filas y en cuatro columnas.



Para disponer los controles de esta manera, hemos de seleccionar el objeto *gridLayout1* en el panel de estructura y cambiar las propiedades **columns** y **rows** tal como se ve en la figura inferior. Opcionalmente podemos establecer un espaciado vertical y horizontal entre los controles, introduciendo nuevos valores en los editores asociados a las propiedades **hgap** y **vgap**.



Los controles ocupan todo el panel, de modo que cuando se cambian las dimensiones del applet los controles cambian de tamaño para ajustarse a sus nuevas dimensiones, tal como se ve en la figura inferior.



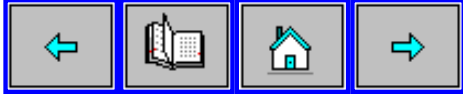
```
public class GridApplet extends Applet {
    Button btn00 = new Button();
    Button btn01 = new Button();
    Button btn02 = new Button();
    Button btn03 = new Button();
    Button btn10 = new Button();
    Button btn11 = new Button();
    Button btn12 = new Button();
    Button btn13 = new Button();
    GridLayout gridLayout1 = new GridLayout();

    public void init() {
        setBackground(Color.white);
        btn00.setFont(new Font("Dialog", 1, 16));
        btn00.setLabel("00");
        btn01.setFont(new Font("Dialog", 1, 16));
        btn01.setLabel("01");
        btn02.setFont(new Font("Dialog", 1, 16));
        btn02.setLabel("02");
        btn03.setFont(new Font("Dialog", 1, 16));
        btn03.setLabel("03");
        btn10.setFont(new Font("Dialog", 1, 16));
        btn10.setLabel("10");
        btn11.setFont(new Font("Dialog", 1, 16));
        btn11.setLabel("11");
        btn12.setFont(new Font("Dialog", 1, 16));
        btn12.setLabel("12");
        btn13.setFont(new Font("Dialog", 1, 16));
    }
}
```



```
        btn13.setLabel("13");  
        GridLayout1.setRows(2);  
        GridLayout1.setHgap(10);  
        GridLayout1.setColumns(3);  
        GridLayout1.setVgap(10);  
        this.setLayout(gridLayout1);  
        this.add(btn00, null);  
        this.add(btn01, null);  
        this.add(btn02, null);  
        this.add(btn03, null);  
        this.add(btn10, null);  
        this.add(btn11, null);  
        this.add(btn12, null);  
        this.add(btn13, null);  
    }  
}
```

El gestor de diseño *GridBagLayout*



[Gestores de diseño](#)

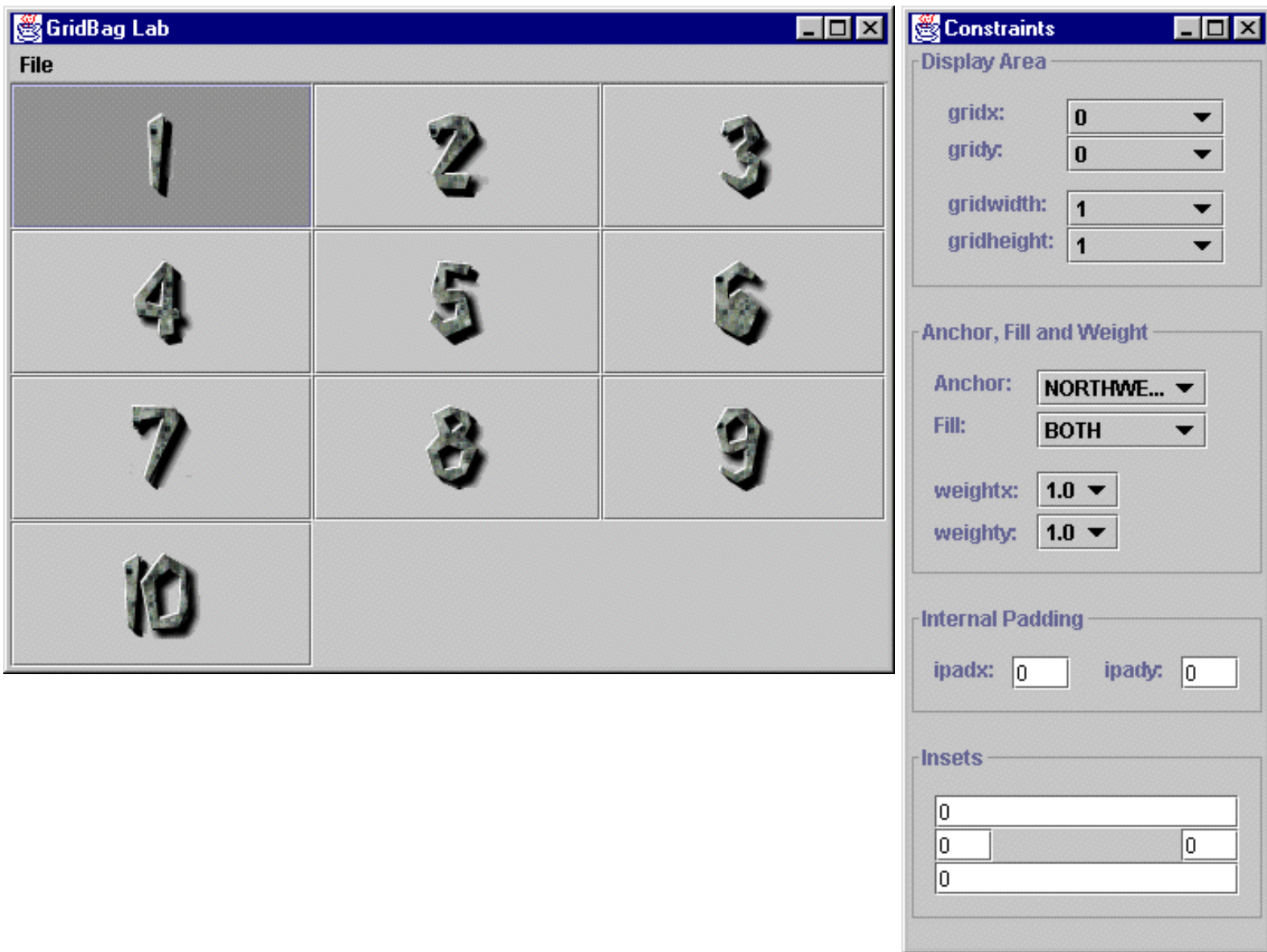
[Las propiedades que gobiernan este gestor de diseño](#)

[Ejemplo: diseño de una ficha](#)

En esta página estudiamos un gestor de diseño bastante complicado, y que necesita de bastante práctica. Quizá la mejor descripción de este gestor se encuentre en el libro de David M. Geary. *Graphic JAVA 1.1. Mastering the JFC (Volume I)*. Prentice Hall (1998). En el libro viene incluida una aplicación denominada GridBagLab, con la que podemos experimentar los cambios en las propiedades de este gestor de diseño, y como afectan al aspecto de una matriz de botones. Varias imágenes tomadas de esta aplicación se han incluido en esta página.

Las propiedades que gobiernan este gestor de diseño

El gestor de diseño *GridBagLayout* es muy complicado, ya que está gobernado por un conjunto de propiedades que están interrelacionados entre sí. Los componentes se disponen en una matriz tal como vimos al estudiar el gestor [GridLayout](#). En la figura vemos la disposición de los controles y a la derecha los valores de los propiedades de dicho gestor para el primer control (en color gris oscuro)



gridx y **gridy** señalan la posición del control en la matriz, el control seleccionado (titulado 1) esta en la fila cero y en la columna 0. Cada uno de los controles tiene una unidad de anchura y una unidad de altura tal como especifica las propiedades **gridwidth** y **gridheight**.

Vamos a cambiar la propiedad **fill** de su valor BOTH a NONE. Vemos que el control 1 está anclado (**Anchor**) en la parte superior izquierda (NORTHWEST) y tiene un tamaño menor que el área en el que se muestra. La clave de la comprensión del este difícil gestor está en esta figura, en la distinción entre el control (el 1 en gris oscuro) y el área en la que se puede mostrar.



Cambiamos la propiedad **Anchor** de NORTHWEST (esquina superior izquierda) a CENTER (centro). El control 1 aparecerá de la forma en la que se indica en la figura.



Cambiamos ahora la propiedad **fill** de NONE a HORIZONTAL. El control incrementa su tamaño horizontalmente hasta ocupar toda el área disponible.



Partimos de la situación inicial restaurando el valor de la propiedad **fill** a BOTH (el control incrementa su tamaño hasta ocupar toda el área disponible tanto horizontalmente como verticalmente). Cambiemos la propiedad **Insets** de su valor por defecto (0, 0, 0, 0), estableciendo un margen de 10 unidades a la izquierda, a la derecha, arriba y abajo. El efecto se ve en la figura inferior.



Restablezcamos todas las propiedades a su valor por defecto y examinemos el efecto del cambio en la propiedad **gridwidth**.

Cambiamos ahora la propiedad **gridwidth** del control 2, de su valor por defecto 1, a la constante REMAINDER. Veremos en la figura que este control se convierte en el último de la primera fila. El control ocupa toda el área disponible por que su propiedad **fill** tiene el valor de BOTH.

Como veremos en el ejemplo, REMAINDER es el valor de la propiedad **gridwidth** del último elemento de una fila.

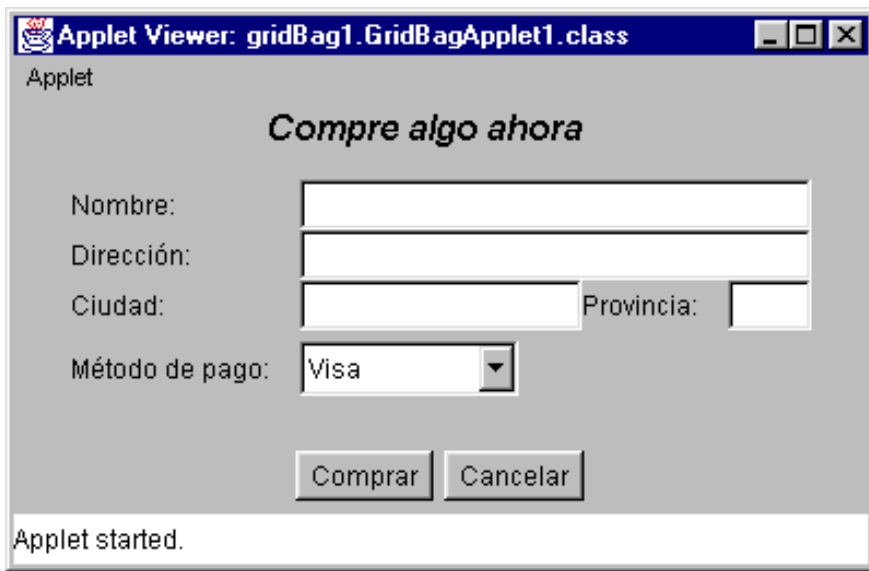


Una vez que hemos visto las propiedades fundamentales del gestor *GridBagLayout*, vamos a usarlas para diseñar una ficha.

Ejemplo: diseño de una ficha

 gridBag1: [GridbagApplet.java](#)

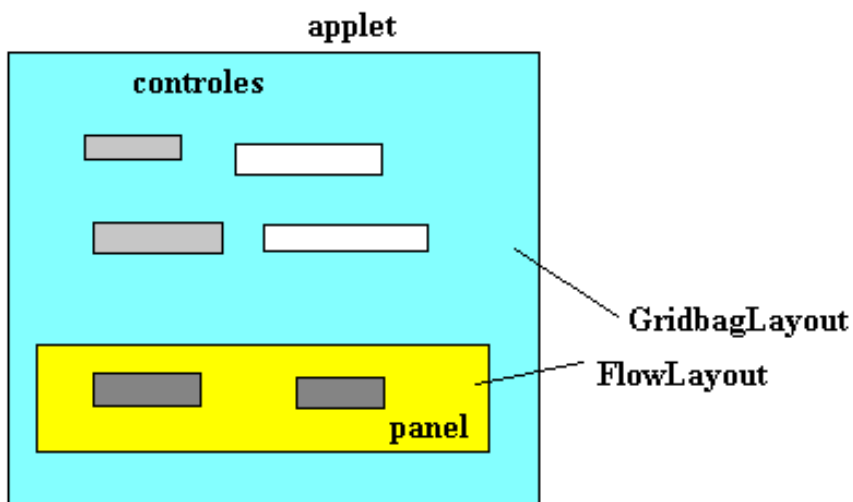
Vamos a estudiar los pasos necesarios para crear una ficha como la que muestra la figura empleando el gestor de diseño *GridBagLayout*



Se crea un applet con el [asistente de creación de applets](#).

Vamos ahora al modo diseño, pulsando con el ratón en la pestaña **Design**. Nos aseguramos de que **this** (el applet) tiene establecido el gestor de diseño *XYLayout*. En el caso de que no lo tenga establecido, seleccionar **this** en el panel de estructura y cambiar su propiedad **layout** a *XYLayout* en su editor asociado.

Añadimos un panel en la parte inferior del applet y varios controles en la parte superior. Sobre el panel situamos dos botones.



El panel

Poner un panel en la parte inferior y cambiar su propiedad **name**

```
Panel panelBotones = new Panel();
```

Seleccionar en el panel de estructura *panelBotones* y cambiar su propiedad **layout** a *XYLayout*

Poner dos botones sobre el panel. Cambiar sus propiedades **name** y **label**

```
Button btnPago=new Button();
Button btnCancelar=new Button();
```

```
btnPago.setLabel("Comprar");  
btnCancelar.setLabel("Cancelar");
```

Seleccionar de nuevo en el panel de estructura *panelBotones* y cambiar su propiedad **layout** a [*FlowLayout*](#). Los botones se sitúan en el centro del panel. Opcionalmente, establecer un espaciado horizontal entre los botones.

El applet

Situar los siguientes controles sobre el applet cambiando su propiedad **name**.

```
Label titulo=new Label();  
Label nombre=new Label();  
Label direccion=new Label();  
Label pago=new Label();  
Label telefono=new Label();  
Label ciudad=new Label();  
Label provincia=new Label();  
  
TextField textNombre=new TextField();  
TextField textDireccion=new TextField();  
TextField textCiudad=new TextField();  
TextField textProvincia=new TextField();  
  
Choice chPago=new Choice();
```

Cambiar las propiedades de los controles a los valores que se indican en el código de la función miembro *init*.

```
titulo.setText("Compre algo ahora");  
titulo.setFont(new Font("Times-Roman", Font.BOLD + Font.ITALIC, 16));  
nombre.setText("Nombre:");  
direccion.setText("Dirección:");  
pago.setText("Método de pago:");  
telefono.setText("Teléfono:");  
ciudad.setText("Ciudad:");  
provincia.setText("Provincia:");  
  
textNombre.setColumns(25);  
textDireccion.setColumns(25);  
textCiudad.setColumns(15);  
textProvincia.setColumns(2);  
  
btnPago.setLabel("Comprar");  
btnCancelar.setLabel("Cancelar");
```

Volver al modo código fuente (pulsar sobre la pestaña **Source**). Añadir elementos al control selección (*Choice*) denominado *chPago*.

```
chPago.add("Visa");  
chPago.add("MasterCard");  
chPago.add("Caja Ahorros");
```


El gestor de diseño *GridBagLayout*

Eliminar en el código fuente todas las líneas de código que hacen referencia al gestor *XYLayout*.

Crear dos objetos uno de la clase *GridBagLayout* y otro de la clase *GridBagConstraints*. El segundo objeto establece los *constraints*. El objeto *gbc* tiene los valores por defecto que hemos mencionado en el primer apartado. Si se cambia un valor de una propiedad (**gridwidth**, **anchor**, **fill**, etc) el cambio permanece hasta que se vuelve a establecer otro valor.

```
GridBagLayout gbl=new GridBagLayout();
GridBagConstraints gbc=new GridBagConstraints();
```

Establecer el gestor de diseño *GridBagLayout* mediante *setLayout*.

```
setLayout(gbl);
```

Añadir los componentes al applet

Se añade un componente al applet (o a un panel) en el que se ha establecido el gestor de diseño *GridBagLayout*, y se han definido los constraints del siguiente modo.

```
add(componente, constraints);
```

La primera fila está formada por un único (REMAINDER) componente el *titulo*, centrado en la parte superior (NORTH)

```
gbc.anchor=GridBagConstraints.NORTH;
gbc.gridwidth=GridBagConstraints.REMAINDER;
add(titulo, gbc);
```

La segunda fila, está formada por dos controles, *nombre* y a continuación *textNombre*. El primero está anclado (**anchor**) al este (WEST), antes estaba en NORTH, **gridwidth** toma el valor 1 (por defecto) antes estaba en REMAINDER. Como *textNombre* es el último control de la fila **gridwidth** vuelve a tomar el valor REMAINDER. Los controles ocupan todo el espacio horizontal disponible, la propiedad **fill** toma el valor HORIZONTAL

```
gbc.fill=GridBagConstraints.HORIZONTAL;
gbc.anchor=GridBagConstraints.WEST;
gbc.gridwidth=1;
add(nombre, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
add(textNombre, gbc);
```

La tercera fila, está formada por dos controles: *direccion* y *textDireccion*. Las propiedades *anchor* y *fill* ya tienen fijados sus valores a WEST y HORIZONTAL, por lo que no hay que volver a establecerlo. Como *textDireccion* es el último control de la fila su propiedad **gridwidth** toma el valor REMAINDER.

```
gbc.gridwidth = 1;
add(direccion, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
add(textDireccion, gbc);
```

La cuarta fila, está formada por cuatro controles: *ciudad*, *textCiudad*, *provincia* y *textProvincia*. Cuando se coloca el último control *textProvincia*, su propiedad **gridwidth** toma el valor REMAINDER.

```
gbc.gridwidth = 1;
add(ciudad, gbc);
add(textCiudad, gbc);
add(provincia, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
add(textProvincia, gbc);
```

La quinta fila, está formada por dos controles: *pago* y *chPago*. El código es semejante, salvo la propiedad **fill** que toma el valor NONE antes estaba en HORIZONTAL. Esto hace que el control *chPago* no ocupe toda el área disponible, extendiéndose horizontalmente hasta alinearse con los otros controles por la parte derecha.

```
gbc.gridwidth = 1;
add(pago, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.fill=GridBagConstraints.NONE;
add(chPago, gbc);
```

Finalmente, añadimos el panel *panelBotones* centrado. Cambiamos la propiedad **anchor** de WEST a SOUTH.

```
gbc.anchor=GridBagConstraints.SOUTH;
add(panelBotones, gbc);
```

Terminamos el diseño de la ficha separando convenientemente las filas de controles con objetos de la clase **Insets**, tal como se muestra en el código.

```
public class GridBagApplet1 extends Applet {
    Panel panelBotones = new Panel();

    Label titulo=new Label();
    Label nombre=new Label();
    Label direccion=new Label();
    Label pago=new Label();
    Label telefono=new Label();
    Label ciudad=new Label();
    Label provincia=new Label();

    TextField textNombre=new TextField();
    TextField textDireccion=new TextField();
    TextField textCiudad=new TextField();
    TextField textProvincia=new TextField();

    Choice chPago=new Choice();

    Button btnPago=new Button();
    Button btnCancelar=new Button();

    GridBagLayout gbl=new GridBagLayout();
    GridBagConstraints gbc=new GridBagConstraints();
    FlowLayout flowLayout1=new FlowLayout();
```

```

    public void init() {
        setBackground(Color.lightGray);
//propiedades de los controles
        titulo.setText("Compre algo ahora");
        titulo.setFont(new Font("Times-Roman", Font.BOLD + Font.ITALIC, 16));
        nombre.setText("Nombre:");
        direccion.setText("Dirección:");
        pago.setText("Método de pago:");
        telefono.setText("Teléfono:");
        ciudad.setText("Ciudad:");
        provincia.setText("Provincia:");

        textNombre.setColumns(25);
        textDireccion.setColumns(25);
        textCiudad.setColumns(15);
        textProvincia.setColumns(2);

        btnPago.setLabel("Comprar");
        btnCancelar.setLabel("Cancelar");

        chPago.add("Visa");
        chPago.add("MasterCard");
        chPago.add("Caja Ahorros");

//gestor gridBaglayout
        setLayout(gbl);

//primera fila - título
        gbc.anchor=GridBagConstraints.NORTH;
        gbc.insets=new Insets(0,0,10,0);
        gbc.gridwidth=GridBagConstraints.REMAINDER;
        add(titulo, gbc);

//segunda fila - nombre
        gbc.fill=GridBagConstraints.HORIZONTAL;
        gbc.anchor=GridBagConstraints.WEST;
        gbc.gridwidth=1;
        gbc.insets=new Insets(0,0,0,0);
        add(nombre, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(textNombre, gbc);

//tercera fila - dirección
        gbc.gridwidth = 1;
        add(direccion, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(textDireccion, gbc);

//cuarta fila - ciudad - provincia
        gbc.gridwidth = 1;
        add(ciudad, gbc);
        add(textCiudad, gbc);
        add(provincia, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        add(textProvincia, gbc);

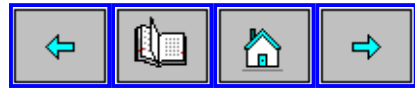
//quinta fila - pago

```

```
        gbc.gridwidth = 1;
        add(pago, gbc);
        gbc.insets=new Insets(5,0,5,0);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.fill=GridBagConstraints.NONE;
        add(chPago, gbc);
//panel de los botones
        panelBotones.setLayout(flowLayout1);
        panelBotones.add(btnPago);
        panelBotones.add(btnCancelar);

        gbc.anchor=GridBagConstraints.SOUTH;
        gbc.insets=new Insets(15,0,0,0);
        add(panelBotones, gbc);
    }
}
```

Bases del nuevo modelo



Sucesos (events)

[Sucesos, componentes, interfaces](#)

[Respuesta a la acción de pulsar sobre un botón](#)

[La clase que describe el applet implementa el interface *Listener*](#)

[Una clase denominada *AccionBoton* implementa el interface *Listener*](#)

[Cómo genera JBuilder el código](#)

[Clases internas](#)

[Clases internas anónimas](#)

[Separando el código de inicialización de la respuesta.](#)

Sucesos, componentes, interfaces

En el modelo AWT 1.1 los sucesos (events) son generados por fuentes (sources). Uno o más objetos interesados (listeners) son notificados que tal o cual suceso se ha producido en una fuente particular.

Los objetos interesados en los sucesos, también llamados manejadores de sucesos (event handlers) pueden ser objetos de cualquier clase, siempre que dicha clase implemente un determinado interface. Son necesarios tres pasos para gestionar los sucesos.

1. Una clase que [implemente un interface](#), por ejemplo *ActionListener*

```
class MiClase implements ActionListener{
    //...
}
```

2. La implementación del método del interface

```
public void actionPerformed(ActionEvent ev){
    //código respuesta a la acción del usuario sobre el componente
}
```

3. La relación entre la fuente de los sucesos (el componente o componentes) y el *objetodeMiClase* que está interesado o que maneja el suceso.

```
componente.addActionListener(objetoDeMiClase);
```

4. La información acerca del suceso viene encapsulada en un objeto de una clase específica derivada de *Event*.

Respuesta a la acción de pulsar sobre un botón

Veamos un ejemplo sencillo que consiste en un botón que al ser pulsado muestra un mensaje en un control etiqueta (label).



El control *Button*

El botón o control *Button* es uno de más simples y utilizados en un interfaz gráfico de usuario.

```
Button btnAceptar=new Button();
```

Para establecer el título del botón se llama a *setLabel*

```
btnAceptar.setLabel("Aceptar");
```

El control *Label*

La etiqueta o control label sirve para mostrar un mensaje que habitualmente no cambia. Normalmente, acompaña a los controles de edición, para indicar al usuario el tipo de información que tiene que introducir. Para establecer el texto en la etiqueta se llama a *setText*

```
lMensaje.setText("Se ha pulsado el botón");
```

Propósito

Un control etiqueta muestra el texto "Pulsar el botón". Al pulsar en el botón Aceptar se cambia el texto que aparece en dicho control a "Se ha pulsado el botón"

Diseño

Crear un applet y situar sobre el applet en el modo diseño (pestaña **Design**) un control etiqueta (Label) y un botón (Button).

Cambiar sus propiedades en sus respectivas hojas de propiedades

Establecer [*FlowLayout*](#) como gestor de diseño del applet, separando horizontalmente los controles, cambiando la propiedad **hgap**.

Respuesta a las acciones del usuario

Para responder a la acción de pulsar un botón, hay que implementar el interface *ActionListener*. Dicho interface tiene una única función miembro denominada *actionPerformed*. La clase que implementa este interface obligatoriamente ha de definir esta función.

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent e);  
}
```

A lo largo de esta página, vamos a describir las distintas aproximaciones que existen para responder a las acción sobre el botón, que nos servirán de modelo para los otros controles.

La clase que describe el applet implementa el interface *Listener*



boton1: [BotonApplet1.java](#)

El código consta como hemos descrito en el primer apartado, de tres partes

1. La clase que describe el applet *BotonApplet1* implementa el interface *ActionListener* y define la función *actionPerformed*

```
public class BotonApplet1 extends Applet implements ActionListener{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    //...
    public void actionPerformed(ActionEvent ev){
        lMensaje.setText("Se ha pulsado el botón");
    }
}
```

En esta función definimos lo que se quiere hacer cuando se pulsa el botón. En este caso, al pulsar el botón *btnAceptar* se muestra un mensaje en un control etiqueta *lMensaje*.

2. El control *btnAceptar* es la fuente de los sucesos (events), y el objeto (listener) que los maneja es el objeto applet o **this**. La función que asocia ambos objetos se denomina *addActionListener*. En la función miembro *init*, el control botón llama a *addActionListener* y le pasa el objeto (el applet) **this** que maneja los sucesos.

```
public class BotonApplet1 extends Applet implements ActionListener{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    public void init(){
        //...
        btnAceptar.addActionListener(this);
    }
    //...
}
```

De este modo, cada vez que se pulsa el botón se llama a la función respuesta *actionPerformed*, donde se programan las tareas específicas que deseamos realizar cuando se pulsa el botón.

El código completo de este ejemplo, es el siguiente

```
package boton1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonApplet1 extends Applet implements ActionListener{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init(){
        lMensaje.setText("Pulsar el botón          ");
        btnAceptar.setLabel("Aceptar");
        btnAceptar.addActionListener(this);
        flowLayout1.setHgap(25);
        this.setLayout(flowLayout1);
        this.add(lMensaje, null);
        this.add(btnAceptar, null);
    }

    public void actionPerformed(ActionEvent ev){
        lMensaje.setText("Se ha pulsado el botón");
    }
}
```

```
}
}
```

Una clase denominada *AccionBoton* implementa el interface *Listener*



boton2: [BotonApplet2.java](#)

Vamos a ver una segunda aproximación. Definimos una clase denominada *AccionBoton* que implementa el interface *ActionListener*. En *init* o *jbInit* (si se genera el código con JBuilder) se asocia el control que es la fuente de los sucesos con un objeto (listener) de dicha clase que maneja los sucesos.

Se define una clase *AccionBoton* que implementa el interface *ActionListener* y define la función *actionPerformed*

```
class AccionBoton implements ActionListener{
    private Label label;
    public AccionBoton(Label label){
        this.label=label;
    }
    public void actionPerformed(ActionEvent ev){
        label.setText("Se ha pulsado el botón");
    }
}
```

Se asocia la fuente que produce los sucesos, el botón *btnAceptar*, con el objeto *accion* de la clase *AccionBoton* que maneja o que está interesado en dichos sucesos.

```
AccionBoton accion=new AccionBoton(lMensaje);
btnAceptar.addActionListener(accion);
```

La clase *AccionBoton* es independiente de la clase que describe el applet, *BotonApplet2*. La función miembro *actionPerformed* de la clase *AccionBoton* necesita tener acceso al control *lMensaje* para que muestre un texto en dicho control cuando se pulsa el botón. Para ello, ponemos como miembro dato de dicha clase un control etiqueta (*Label*) que inicializamos en el constructor al pasarle *lMensaje*.

El código completo de este ejemplo, es el siguiente

```
package boton2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonApplet2 extends Applet{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init() {
        lMensaje.setText("Pulsar el botón          ");
        btnAceptar.setLabel("Aceptar");
        AccionBoton accion=new AccionBoton(lMensaje);
        btnAceptar.addActionListener(accion);
        flowLayout1.setHgap(25);
        this.setLayout(flowLayout1);
        this.add(lMensaje, null);
    }
}
```



```

        this.add(btnAceptar, null);
    }
}
//*****
class AccionBoton implements ActionListener{
    private Label label;
    public AccionBoton(Label label){
        this.label=label;
    }
    public void actionPerformed(ActionEvent ev){
        label.setText("Se ha pulsado el botón");
    }
}

```

Cómo genera JBuilder el código (Standard adapter)



boton3: [BotonApplet3.java](#)

Para tener acceso desde la clase *AccionBoton* a los miembros datos que no sean privados (**private**) de la clase que describe el applet, en vez de pasarle en el constructor una lista de dichos miembros, le podemos pasar el objeto **this** que hace referencia el applet. El miembro dato *applet* de la clase *AccionBoton*, se inicializa con **this** en el constructor. Desde el objeto *applet* se accede al miembro dato *lMensaje* para mostrar un texto en dicho control. La clase *AccionBoton* quedaría como sigue.

```

class AccionBoton implements ActionListener{
    private BotonApplet3 applet;
    public AccionBoton(BotonApplet3 applet){
        this.applet=applet;
    }
    public void actionPerformed(ActionEvent ev){
        applet.lMensaje.setText("Se ha pulsado el botón");
    }
}

```

En *init* creamos un objeto de la clase *AccionBoton* y le pasamos el applet (**this**). Luego, asociamos la fuente que produce los sucesos, el botón *btnAceptar*, con el objeto *accion* de la clase *AccionBoton* que los maneja.

```

public void init(){
    //...
    AccionBoton accion=new AccionBoton(this);
    btnAceptar.addActionListener(accion);
}

```

JBuilder tiene la posibilidad de generar el código de la declaración de la función respuesta, en el modo diseño, seleccionando el control y eligiendo el panel Events la función respuesta (*actionPerformed*) que queremos definir. El código generado tiene una forma similar a la siguiente.

```

package boton3;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonApplet3 extends Applet{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init() {
        lMensaje.setText("Pulsar el botón          ");
        btnAceptar.setLabel("Aceptar");
        AccionBoton accion=new AccionBoton(this);
        btnAceptar.addActionListener(accion);
        flowLayout1.setHgap(25);
        this.setLayout(flowLayout1);
        this.add(lMensaje, null);
        this.add(btnAceptar, null);
    }
    void btnAceptar_actionPerformed(ActionEvent ev) {
        lMensaje.setText("Se ha pulsado el botón");
    }
}

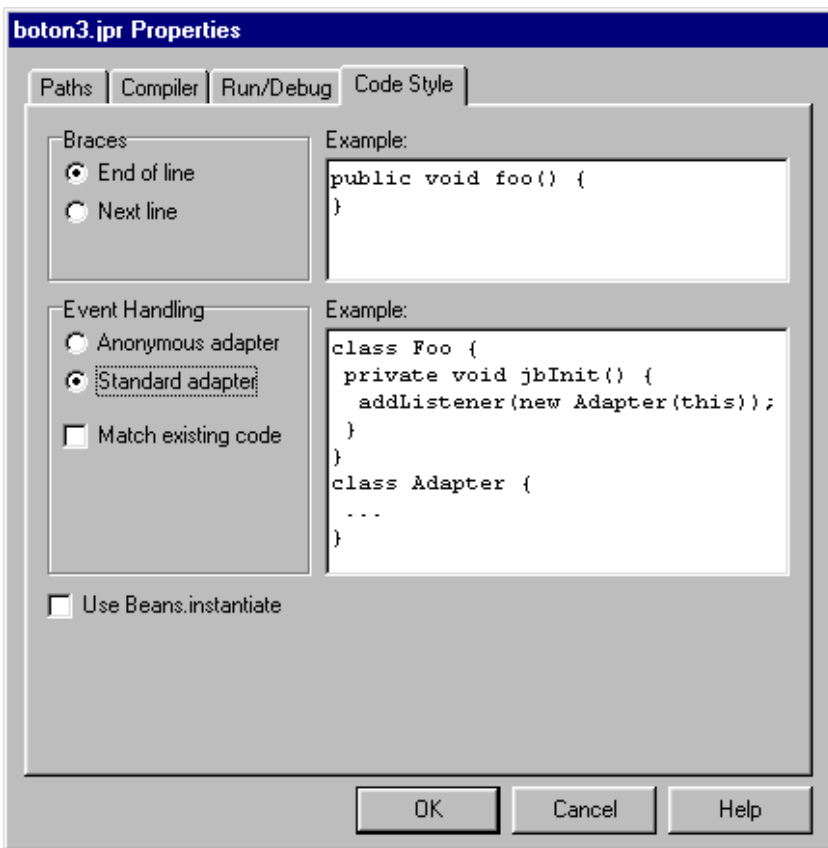
//*****
class AccionBoton implements ActionListener{
    private BotonApplet3 applet;
    public AccionBoton(BotonApplet3 applet){
        this.applet=applet;
    }
    public void actionPerformed(ActionEvent ev){
        applet.btnAceptar_actionPerformed(ev);
    }
}

```

Vemos que en la clase que describe el applet *BotonApplet3*, se define una función respuesta *btnAceptar_actionPerformed*. Desde la función miembro *actionPerformed* de la clase *AccionBoton* se llama a dicha función. De este modo al ser *btnAceptar_actionPerformed* miembro de la clase que describe el applet tiene acceso a todos sus miembros sean públicos o privados.

Otra ventaja que presenta esta aproximación es que en la clase que define al applet, *BotonApplet3*, tenemos por una parte el código de inicialización del applet, en *init* (*jbInit*), y por otra el código de las funciones respuesta. El mantenimiento del código es más fácil con esta aproximación, ya que una sola clase controla la inicialización y las funciones respuesta a las acciones del usuario, en vez de estar en clases separadas.

Para que JBuilder genere el código de esta forma es necesario seleccionar en el elemento del menú **Run/Parameters**. En el cuadro de diálogo que aparece se selecciona la pestaña **Code style**. Se activa el botón del radio titulado **Standard adapter** del grupo titulado **Event Handling**.



Se selecciona el control en modo de diseño, y se hace doble-clic en el panel **Events**, en el editor asociado al nombre de la función *actionPerformed* tal como se ve en la figura. En dicho editor podemos cambiar el nombre de la función respuesta *btnAceptar_actionPerformed* generado por JBuilder. Podemos también generar dicho código haciendo doble-clic sobre el propio botón en el modo diseño



Clases internas

 boton4: [BotonApplet4.java](#)

En esta aproximación vamos a mover la clase *AccionBoton* al interior de la clase que describe el applet. La clase *AccionBoton* se denomina ahora interna. Una clase interna tiene acceso a los miembros de la clase que la encierra. El uso de las clases internas como veremos simplifica mucho el

código. Para mover la clase *AccionBoton* al interior de la clase que describe el applet simplemente trasladamos la llave de cierre del applet después de la definición de la clase *AccionBoton*.

```
package boton4;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonApplet4 extends Applet{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init() {
        lMensaje.setText("Pulsar el botón          ");
        btnAceptar.setLabel("Aceptar");
        btnAceptar.addActionListener(new AccionBoton());
        flowLayout1.setHgap(25);
        this.setLayout(flowLayout1);
        this.add(lMensaje, null);
        this.add(btnAceptar, null);
    }

    class AccionBoton implements ActionListener{
        public void actionPerformed(ActionEvent ev){
            lMensaje.setText("Se ha pulsado el botón");
        }
    }
}
```

Como vemos en el código, la clase interna *AccionBoton* tiene acceso al miembro dato *lMensaje* de la clase que describe el applet.

Clases internas anónimas

Las clase internas anónimas nos permiten simplificar aún más el código que define la respuesta a la acción del usuario sobre un control. En *init* o en *jbInit* (si el código es generado por JBuilder) se escribe

```
public void init(){
    //...
    btnAceptar.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent ev){
            lMensaje.setText("Se ha pulsado el botón");
        }
    });
}
```

En unas pocas líneas de código, se crea con **new** un objeto de una clase anónima que implementa el interface *ActionListener*

```
new ActionListener(){
    //...
}
```

Dentro de dicha clase se ha de definir la función *actionPerformed*

```

new ActionListener(){
    public void actionPerformed(ActionEvent ev){
        //definir la función respuesta
    }
}

```

Se asocia mediante *addActionListener* el control botón *btnAceptar* con el objeto de la clase anónima que maneja las acciones del usuario sobre dicho botón.

```

btnAceptar.addActionListener(new ActionListener(){//...});

```

Separando el código de inicialización de la respuesta (Anonymous adapter)



boton5: [BotonApplet5.java](#)

Para una mejor lectura del código, es mejor separar el código de inicialización del applet, la definición de *init* (o *jbInit*), del código que describe la respuesta a las acciones del usuario sobre los controles. Sin modificar la funcionalidad del applet mejoramos la legibilidad del código tal como se muestra en las líneas marcadas en letra negrita.

```

package boton5;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonApplet5 extends Applet{
    Label lMensaje = new Label();
    Button btnAceptar = new Button();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init() {
        lMensaje.setText("Pulsar el botón          ");
        btnAceptar.setLabel("Aceptar");
        btnAceptar.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                btnAceptar_actionPerformed(e);
            }
        });
        flowLayout1.setHgap(25);
        this.setLayout(flowLayout1);
        this.add(lMensaje, null);
        this.add(btnAceptar, null);
    }

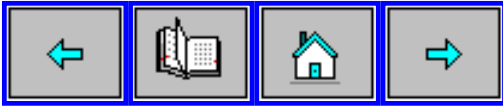
    void btnAceptar_actionPerformed(ActionEvent ev) {
        lMensaje.setText("Se ha pulsado el botón");
    }
}

```

Esta es la otra forma que compilador JBuilder genera el código de la respuesta a las acción de pulsar sobre el botón, cuando se selecciona el control en modo de diseño, y se hace doble-clic en el panel **Events**, en el editor asociado al nombre de la función *actionPerformed* tal como se ve en la figura más arriba. En dicho editor podemos cambiar el nombre de la función respuesta *btnAceptar_actionPerformed* generado por JBuilder. Podemos también generar dicho código haciendo doble-clic sobre el propio botón en el modo diseño.

JBuilder genera este código por defecto. Podemos comprobarlo, al seleccionar en el elemento del menú **Run/Parameters**. En el diálogo [Properties](#) que vimos anteriormente, se selecciona la pestaña **Code style**, veremos activado el botón del radio titulado **Anonymous adapter** del grupo titulado **Event Handling**, si no lo hemos cambiado previamente.

Respuesta a las acciones del usuario sobre diversos controles



[Sucesos\(events\)](#)

[Hacer doble clic sobre un elemento de una lista](#)

[Seleccionar un elemento de una lista](#)

[El control seleccion \(*Choice*\)](#)

[Lista de elección múltiple](#)

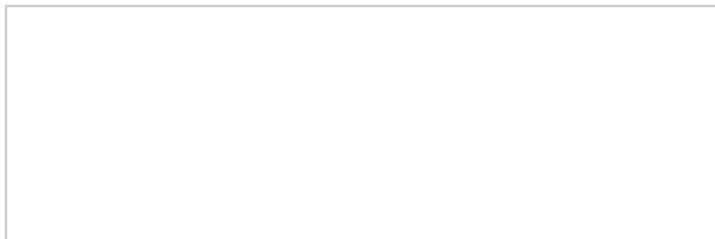
[El control barra de desplazamiento](#)

A continuación vamos a ver la respuesta a las acciones del usuario sobre varios controles, y dada su importancia dedicaremos una página al estudio de los controles de edición, aquellos que nos permiten introducir datos al programa para su procesamiento.

Hacer doble-clic sobre un elemento de una lista

 lista1: [ListaApplet1.java](#)

En una lista existen dos posibles acciones, seleccionar un elemento de la lista cuando se sitúa el cursor sobre dicho elemento, o hacer doble-clic sobre un elemento de la lista.



Propósito

Un control lista tiene como elementos los nombres de los tres colores básicos: rojo, verde y azul. Al hacer doble-clic sobre un elemento de la lista se pinta un rectángulo del color seleccionado.

El control *List*

Para crear un control *lista* de la clase *List* y añadirle elementos, se procede del siguiente modo

```
List lista=new List();
lista.add("Rojo");
lista.add("Verde");
lista.add("Azul");
lista.select(0);
```

La última línea de código, indica que el primer elemento de la lista, "Rojo", aparecerá inicialmente seleccionado.

Para saber qué elemento de la lista ha sido seleccionado empleamos dos funciones *getSelectedIndex* y *getSelectedItem*, el primero devuelve un entero que indica la posición del elemento en la lista, y la segunda devuelve un string que guarda el elemento de la lista.

```
int indice=lista.getSelectedIndex();
string texto=lista.getSelectedItem();
```

Diseño

Crear un applet y situar sobre el applet en el modo diseño (pestaña **Design**) un control lista (*List*)

Establecer [FlowLayout](#) como gestor de diseño del applet, con alineamiento a la derecha (la propiedad **alignement** vale 2).

Crear un array de objetos de la clase [Color](#) cuyos elementos son los tres colores básicos

```
Color[] colores={Color.red, Color.green, Color.blue};
```

Redefinir la función *paint* para pintar un rectángulo (a la izquierda del applet) del color seleccionado

Respuesta a la acción del usuario

La respuesta a las acción de hacer doble-clic sobre un elemento de la lista es similar a la acción de pulsar sobre un botón, y la resumiremos en los siguientes puntos:

- Se crea una clase que implemente el interface *ActionListener*, y que ha de definir *actionPerformed*
- Mediante *addActionListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso viene encapsulada en un objeto de la clase *ActionEvent*.

Para responder a las acción de hacer doble-clic sobre un elemento de la lista, vamos a elegir la alternativa más cómoda, la que nos proporciona el IDE de JBuilder2. En el modo diseño, seleccionamos el control *lista* y en el panel Events hacemos doble-clic sobre el editor asociado a *actionPerformed*. En [el código que genera JBuilder](#), se asocia el

control *lista* con un objeto de una [clase anónima](#) mediante la función miembro *addActionListener* (Anonymous adapter).

```
lista.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        lista_actionPerformed(e);
    }
});
```

Queda ahora el trabajo de definir la función respuesta *lista_actionPerformed*, la tarea a realizar por esta función miembro, que es la de dibujar un rectángulo pintado del color seleccionado.

Para pintar un rectángulo de un determinado color, redefinimos la función miembro *paint* que nos proporciona el contexto gráfico *g* en el cual podemos dibujar figuras llamando a distintas funciones.

```
public void paint(Graphics g){
    g.setColor(colores[indice]);
    g.fillRect(2, 2, 100, 50);
}
```

La primera sentencia selecciona el color del lápiz con el que se va a pintar el contorno de una figura, o la brocha con la que se va a pintar su interior, donde *colores* es un array cuyos elementos son los tres colores básicos, y la variable *indice* guarda el índice del color seleccionado.

La tarea de la función respuesta *lista_actionPerformed* será la de obtener el índice del color seleccionado y a continuación llamar a la función *paint* para que pinte un rectángulo de dicho color.

```
void lista_actionPerformed(ActionEvent e) {
    indice=lista.getSelectedIndex();
    repaint();
}
```

La función miembro *getSelectedIndex* de la clase *List* obtiene el índice del color seleccionado y lo guarda en el miembro dato *indice*. Luego, llama a *paint*.

El código completo de este ejemplo, es el siguiente

```
package lista1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListaApplet1 extends Applet {
    List lista=new List();
    FlowLayout flowLayout1 = new FlowLayout();
    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice;

    public void init() {
        flowLayout1.setAlignment(2);
        lista.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                lista_actionPerformed(e);
            }
        });
        lista.add("Rojo");
        lista.add("Verde");
        lista.add("Azul");
        lista.select(0);
        this.setLayout(flowLayout1);
        this.add(lista, null);
    }

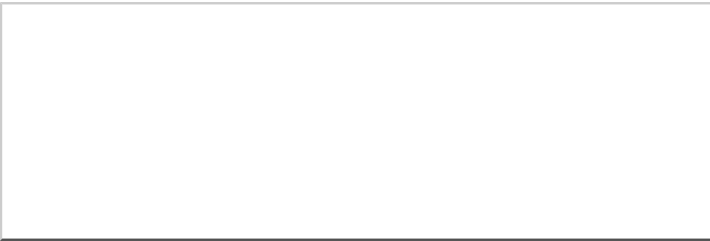
    void lista_actionPerformed(ActionEvent e) {
        indice=lista.getSelectedIndex();
        repaint();
    }

    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(2, 2, 100, 50);
    }
}
```

Seleccionar un elemento de la lista



lista2: [ListaApplet2.java](#)



Propósito

Un control lista tiene como elementos los nombres de los tres colores básicos: rojo, verde y azul. Al seleccionar un elemento de la lista se pinta un rectángulo del color seleccionado.

Diseño

El mismo que en el ejemplo anterior

Respuesta a las acciones del usuario

La respuesta a las acción de seleccionar un elemento de la lista la podemos resumir en los siguientes puntos:

- Se crea una clase que implemente el interface *ItemListener*, y que ha de definir *itemStateChanged*
- Mediante *addItemListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso viene encapsulada en un objeto de la clase *ItemEvent*.

Para responder a las acción de seleccionar un elemento de la lista, vamos a elegir la alternativa más cómoda, la que nos proporciona el IDE de JBuilder. En el modo diseño, seleccionamos el control *lista* y en el panel Events hacemos doble-clic sobre el editor asociado a *itemStateChanged*. En [el código que se genera](#) se asocia el control *lista* con un objeto de una [clase anónima](#) mediante la función miembro *addItemListener* (Anonymous adapter)

```
lista.addItemListener(new java.awt.event.ItemListener() {  
    public void itemStateChanged(ItemEvent e) {  
        lista_itemStateChanged(e);  
    }  
});
```

La definición de la función respuesta *lista_itemStateChanged* es la misma que *lista_actionPerformed*. El código completo de este ejemplo, es el siguiente.

```

package lista2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListaApplet2 extends Applet {
    List lista=new List();
    FlowLayout flowLayout1 = new FlowLayout();

    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice;

    public void init() {
        lista.addItemListener(new java.awt.event.ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                lista_itemStateChanged(e);
            }
        });

        flowLayout1.setAlignment(2);
        lista.add("Rojo");
        lista.add("Verde");
        lista.add("Azul");
        lista.select(0);
        this.setLayout(flowLayout1);
        this.add(lista, null);
    }

    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(2, 2, 100, 50);
    }

    void lista_itemStateChanged(ItemEvent e) {
        indice=lista.getSelectedIndex();
        repaint();
    }
}

```

El control selección (*Choice*)

El uso del control selección (*Choice*) es similar al uso del [control lista](#) (*List*), la diferencia está en el espacio que ocupa en el applet. El control *Choice* se puede desplegar pulsando en la flecha situada en la parte derecha para mostrar los elementos que contiene y seleccionar uno de ellos, que aparece a la izquierda de la flecha de la caja combinada. El

control *Choice* implementa el interface *ItemListener* de la misma manera que el control *List*.

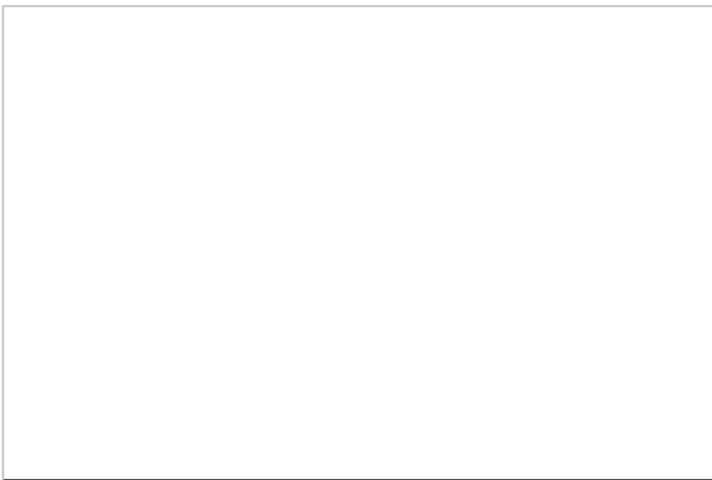
Para crear un control *lista* de la clase *Choice* y añadirle elementos, se procede del siguiente modo

```
Choice lista=new Choice();
lista.add("Rojo");
lista.add("Verde");
lista.add("Azul");
lista.select(1);
```

La última línea de código, indica que el segundo elemento de la lista, "Verde", aparecerá inicialmente seleccionado a la izquierda de la flecha en la caja combinada.

Lista de elección múltiple

 lista3: [ListaApplet3.java](#)



El control *List*

Podemos llenar una lista o un control selección a partir de un array de strings y en un bucle **for**.

```
String[] comidas={"Desayuno", "Comida", "Cena"};
for(int i=0; i<comidas.length; i++){
    chComida.addItem(comidas[i]);
}
```

Para eliminar un elemento de una lista, se llama a la función miembro *remove* y se le pasa el índice del elemento que queremos eliminar o el nombre de dicho elemento.

```
listElegir.remove(indice);
listElegir.remove(item);
```

Para eliminar todos los elementos de una lista, se llama a la función miembro *removeAll*.

```
listElegir.removeAll();
```

En una lista podemos elegir varios elementos, siempre que su propiedad *multipleMode* tenga el valor **true**, (por defecto, toma el valor **false**). Para cambiar esta propiedad llamamos a la función *setMultipleMode*

```
listElegir.setMultipleMode(true);
```

Para obtener los elementos seleccionados disponemos de dos funciones *getSelectedIndexes* que devuelve un array de enteros *int[]* (los índices de los elementos seleccionados), y *getSelectedItems* que devuelve un array de strings, *String[]* (los nombres de los elementos seleccionados).

```
String[] items=listElegir.getSelectedItems();
int[] indices=listElegir.getSelectedIndexes();
```

El control *TextField*

Más adelante estudiaremos con más detalle el [control de edición](#). Por ahora, mencionaremos que para cambiar el texto del control de edición se llama a la función miembro *setText*

```
tMenu.setText("Nuevo texto");
```

Propósito

En este ejemplo, combinamos un control selección (*Choice*), una lista (*List*) que va cambiando según sea el elemento elegido en el control selección, y un control de edición (*TextField*) que muestra los elementos seleccionados.

Diseño

Crear el applet. En el modo diseño (pestaña **Design**) situar sobre el applet un control etiqueta (*Label*), un control selección (*Choice*), un control lista (*List*), y un control de edición (*TextField*). Cambiar sus propiedades en sus correspondientes hojas de propiedades

Establecer [BorderLayout](#) como gestor de diseño del applet, de modo que la etiqueta se sitúa al norte (NORTH), el control selección al oeste (WEST), la lista al este (EAST) y el control de edición al sur (SOUTH).

Crear un array de strings para llenar el control selección

```
String[] comidas={"Desayuno", "Comida", "Cena"};
```

Crear un array bidimensional de strings para llenar la lista con grupos de elementos distintos, según sea la opción elegida en el control selección.

```
String[][] menus={{ "zumos", "huevos", "jamón", "mantequilla", "cereales"},
```

```
{ "pizza", "hamburguesa", "ensalada", "patatas", "filete", "café"},
{ "sopa", "pollo", "tortilla" } };
```

Establecer el estado inicial de los distintos controles

Respuesta a las acciones del usuario

Para responder a las acción de seleccionar un elemento de la lista, vamos a elegir la alternativa más cómoda, la que nos proporciona el IDE de JBuilder. En el modo diseño, seleccionamos el control *listElegir* y en el panel Events hacemos doble-clic sobre el editor asociado a *itemStateChanged*. Hacemos lo mismo para el control selección *chComida*.

Para cambiar dinámicamente los elementos de una lista, primero se eliminan todos mediante *removeAll*, y luego, se añaden los nuevos elementos de la lista mediante *addItem*. En la función respuesta a la selección de un elemento del control selección (*Choice*), se obtiene el índice del elemento que ha sido seleccionado mediante *getSelectedIndex*..

```
void chComida_itemStateChanged(ItemEvent e) {
    int indice=chComida.getSelectedIndex();
    listElegir.removeAll();
    for(int i=0; i<menus[indice].length; i++){
        listElegir.addItem(menus[indice][i]);
    }
}
```

En la función respuesta a la acción de seleccionar elementos en la lista, obtenemos los elementos de la lista que han sido seleccionados mediante *getSelectedItems*, que devuelve un array de strings.

Para poner en el control de edición los nombres de los elementos seleccionados en el control lista, se llama a la función *setText*.

```
void listElegir_itemStateChanged(ItemEvent e) {
    String[] items=listElegir.getSelectedItems();
    String menu=chComida.getSelectedItem()+" : ";
    for(int i=0; i<items.length; i++){
        menu+=items[i]+" ";
    }
    tMenu.setText(menu);
}
```

El código completo de este ejemplo, es el siguiente

```
package lista3;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListaApplet3 extends Applet {
    Label labell = new Label();
    Choice chComida = new Choice();
    List listElegir = new List();
    TextField tMenu = new TextField();
    BorderLayout borderLayout1 = new BorderLayout();
    String[][] menus={"zumo", "huevos", "jamón", "mantequilla", "cereales"},
        {"pizza", "hamburguesa", "ensalada", "patatas", "filete", "café"},
        {"sopa", "pollo", "tortilla"}};
    String[] comidas={"Desayuno", "Comida", "Cena"};

    public void init() {
        labell.setText("Realice su pedido");
        listElegir.addItemListener(new java.awt.event.ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                listElegir_itemStateChanged(e);
            }
        });
        chComida.addItemListener(new java.awt.event.ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                chComida_itemStateChanged(e);
            }
        });
        borderLayout1.setVgap(10);
        borderLayout1.setHgap(10);
        tMenu.setText("");
        listElegir.setMultipleMode(true);
        this.setLayout(borderLayout1);
        this.add(labell, BorderLayout.NORTH);
        this.add(chComida, BorderLayout.WEST);
        this.add(listElegir, BorderLayout.EAST);
        this.add(tMenu, BorderLayout.SOUTH);
        for(int i=0; i<comidas.length; i++){
            chComida.addItem(comidas[i]);
        }
        chComida.select(0);
        //por defecto pone el desayuno
        for(int i=0; i<menus[0].length; i++){
            listElegir.addItem(menus[0][i]);
        }

    }

    void chComida_itemStateChanged(ItemEvent e) {
```



```

        int indice=chComida.getSelectedIndex();
        listElegir.removeAll();
        for(int i=0; i<menus[indice].length; i++){
            listElegir.addItem(menus[indice][i]);
        }
    }

    void listElegir_itemStateChanged(ItemEvent e) {
        String[] items=listElegir.getSelectedItems();
        String menu=chComida.getSelectedItem()+" : ";
        for(int i=0; i<items.length; i++){
            menu+=items[i]+"  ";
        }
        tMenu.setText(menu);
    }
}

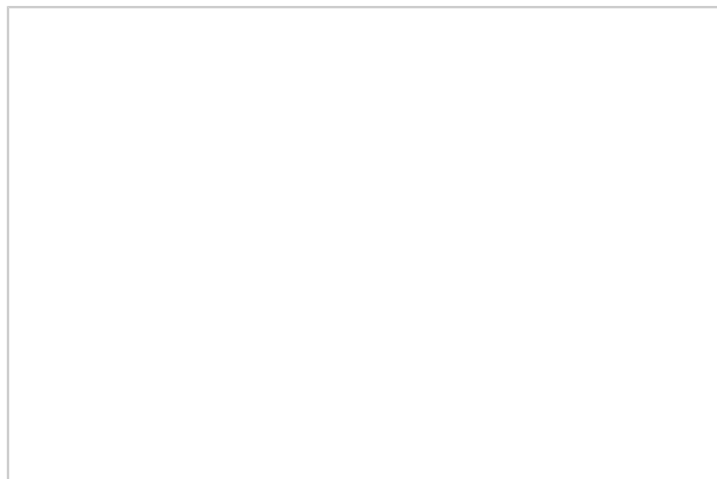
```

El control barra de desplazamiento



barra: [BarraApplet.java](#)

El control barra de desplazamiento se emplea frecuentemente en los interfaces gráficos de usuario, cuando la figura o el texto que queremos ver es más grande que las dimensiones de la ventana. Dicho control es mucho más manejable que un control de edición, ya que basta arrastrar el dedo de la barra de desplazamiento, en vez de introducir un dato en un control de edición y pulsar el botón apropiado. En una barra de desplazamiento no es necesario verificar el dato que se ha introducido, ya que se trata de números enteros en un determinado intervalo.



El control *Scrollbar*

Un control barra de desplazamiento es un objeto de la clase *Scrollbar*. Creamos un control *sbRadio* llamando al constructor por defecto

```
Scrollbar sbRadio = new Scrollbar();
```

Establecemos su orientación mediante la función miembro *setOrientation* pasándole un cero o un uno. La orientación horizontal es cero, o la constante *Scrollbar.HORIZONTAL*, y la orientación vertical es uno o la constante *Scrollbar.VERTICAL*

```
sbRadio.setOrientation(0);
```

El valor máximo y mínimo lo establecemos mediante las funciones miembro *setMaximum* y *setMinimum*, respectivamente.

```
sbRadio.setMaximum(110);
sbRadio.setMinimum(10);
```

Cuando actuamos con el ratón sobre las flechas situadas en los extremos de la barra, el dedo se desplaza una determinada cantidad denominada unidad y que se establece mediante la función miembro *setUnitIncrement*.

```
sbRadio.setUnitIncrement(5);
```

Cuando se actúa con el ratón en las dos regiones de la barra situadas entre las flechas y el dedo, éste se desplaza una determinada cantidad denominada página o bloque, y se establece mediante llamada a la función miembro *setBlockIncrement*.

```
sbRadio.setBlockIncrement(25);
```

La posición del dedo en la barra de desplazamiento se establece en un valor entre el mínimo y el máximo

```
sbRadio.setValue(10);
```

Para obtener la posición del dedo en la barra de desplazamiento se llama a la función miembro *getValue* que devuelve un entero

```
int pos=sbRadio.getValue();
```

Propósito

En este applet hemos situado una barra de desplazamiento en la parte superior del applet. A medida que movemos el dedo de la barra de desplazamiento, el radio de la circunferencia que se dibuja aumenta.

Diseño

Crear el applet y situar un control barra de desplazamiento (*Scrollbar*) en la parte superior

Establecer las propiedades del control en su hoja de propiedades

Establecer [BorderLayout](#) como gestor de diseño del applet de modo que la barra de desplazamiento quede al norte (NORTH).

Redefinir la función *paint* para dibujar una circunferencia (oval) centrada en al applet

Se recuerda al lector, que cada vez que se pretenda dibujar una circunferencia se debe llamar a la función *paint* mediante *repaint*.

Respuesta a las acciones del usuario

La respuesta a las acción del usuario sobre una barra de desplazamiento la podemos resumir en los siguientes puntos:

- Se crea una clase que implemente el interface *AdjustmentListener*, y que ha de definir *adjustmentValueChanged*
- Mediante *addAdjustmentListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso viene encapsulada en un objeto de la clase *AdjustmentEvent*.

Para responder a las acciones de mover el dedo en la barra de desplazamiento, actuar con el ratón sobre las flechas situadas en los extremos, o en entre los extremos de la barra y el dedo, vamos a elegir la alternativa más cómoda, la que nos proporciona el IDE de JBuilder. En el modo diseño, seleccionamos el control *sbRadio* y en el panel Events hacemos doble-clic sobre el editor asociado a *adjustmentValueChanged*. En [el código que se genera](#) se asocia el control *sbRadio* con un objeto de una [clase anónima](#) mediante la función miembro *addAdjustmentListener* (Anonymous adapter)

```
sbRadio.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent ev) {
        sbRadio_adjustmentValueChanged(ev);
    }
});
```

En la definición de la función respuesta *sbRadio_adjustmentValueChanged*, obtenemos el valor del dedo en la barra de desplazamiento mediante la función miembro *getValue*, y la guardamos en el miembro dato *radio*.

```
void sbRadio_adjustmentValueChanged(AdjustmentEvent ev) {
    radio=sbRadio.getValue();
    repaint();
}
```

Para dibujar una circunferencia cuyo centro está situado en el centro del applet, y cuyo radio sea el valor que indica el dedo en la barra de desplazamiento redefinimos la función miembro *paint*.

```
public void paint(Graphics g){
    int x1=getSize().width/2;
    int y1=getSize().height/2;
    g.setColor(Color.black);
```

```
g.drawOval(x1-radio, y1-radio, 2*radio, 2*radio);
}
```

Las dimensiones del applet se obtienen mediante la función miembro *getSize*, que devuelve un objeto de la clase *Dimension*, que tiene dos miembros dato *width*, que proporciona la anchura del applet y *height* que proporciona la altura del applet.

El código completo de este ejemplo, es el siguiente

```
package barra;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BarraApplet extends Applet {
    Scrollbar sbRadio = new Scrollbar();
    BorderLayout borderLayout1 = new BorderLayout();
    int radio=10;

    public void init() {
        sbRadio.setOrientation(0);
        sbRadio.setValue(10);
        sbRadio.setMaximum(110);
        sbRadio.setUnitIncrement(5);
        sbRadio.setBlockIncrement(25);
        sbRadio.setMinimum(10);

        sbRadio.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent ev) {
                sbRadio_adjustmentValueChanged(ev);
            }
        });

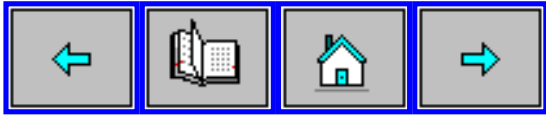
        this.setLayout(borderLayout1);
        this.add(sbRadio, BorderLayout.NORTH);
    }

    void sbRadio_adjustmentValueChanged(AdjustmentEvent ev) {
        radio=sbRadio.getValue();
        repaint();
    }

    public void paint(Graphics g){
        int x1=getSize().width/2;
        int y1=getSize().height/2;
        g.setColor(Color.black);
        g.drawOval(x1-radio, y1-radio, 2*radio, 2*radio);
    }
}
```

```
}
```

Respuesta a las acciones del usuario sobre un grupo de controles (botones)



Sucesos (events)

[La información acerca del suceso](#)

[Respuesta a las acciones del usuario sobre un conjunto de tres botones](#)

[Tres funciones respuesta](#)

[Un único objeto maneja los sucesos de un grupo de botones](#)

La información acerca del suceso

En el estudio de la función respuesta a la acción de [pulsar sobre un botón](#), o de hacer [doble-clic sobre un elemento de una lista](#), el objeto *ev* de la clase *ActionEvent*, nos proporciona información acerca del suceso que se ha producido, nos dice qué control ha generado el suceso. Por ejemplo, podemos saber mediante la función miembro *getSource* si el suceso procede de la acción sobre un botón, de un control lista, o de un control de edición, etc.

En el código de la función respuesta *actionPerformed*, podemos saber si el control sobre el que se ha actuado es una instancia de la clase *Button* mediante el operador [instanceof](#)

```
Object control=ev.getSource();
if(control instanceof Button){
    System.out.println("Se ha pulsado un botón");
}
```

Si hay varias botones, podemos saber cual de ellos ha sido pulsado mediante [equals](#).

```
Object control=ev.getSource();
if(control.equals(btnAceptar){
    System.out.println("Se ha pulsado el botón Aceptar");
}
```

Mediante *getActionCommand* obtenemos el nombre (etiqueta) del botón. Podemos saber si se ha pulsado sobre un botón titulado "Rojo".

```
String nombre=ev.getActionCommand();
if(nombre.equals("Rojo")){
    System.out.println("Se ha pulsado el botón Rojo");
}
```

Como veremos más adelante, dependiendo del tipo de suceso, la información está encapsulada en distintas clases, que proporcionan la información que interesa al usuario. Así, la clase *AdjustmentEvent* encapsula toda la información referente a las acciones sobre la [barra de desplazamiento](#). Del objeto *ev* podemos extraer mediante la función miembro *getValue*, el valor al que equivale la posición del dedo en la barra de desplazamiento.

```
void sbRadio_adjustmentValueChanged(AdjustmentEvent ev) {
    radio=ev.getValue();
    repaint();
}
```

La clase *ItemEvent* encapsula la información referente a las acciones sobre [un control lista, o un control selección](#). La función miembro *getItem* obtiene de un objeto *ev* de dicha clase el elemento que ha sido seleccionado. *getItem* devuelve un objeto de la clase base *Object*, que puede ser necesario promocionar (casting) a un objeto de la clase adecuada.

En el caso de un control selección (*Choice*) comparamos mediante *equals*, el objeto devuelto por *getItem* (el elemento seleccionado) con cada uno de los nombres de los elementos de dicho control.

```
void chComida_itemStateChanged(ItemEvent ev) {
    if((ev.getItem()).equals("Desayuno")){
        //...
    }
}
```

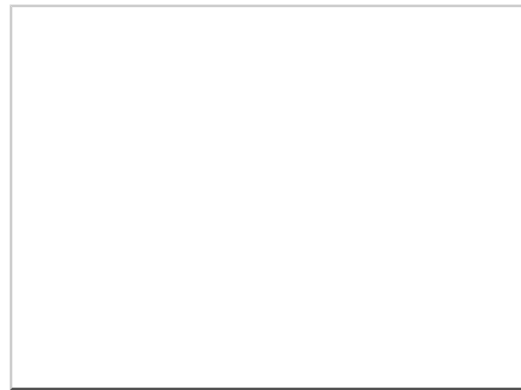
En el caso de un control lista (*List*) el objeto de la clase *Object* devuelto por *getItem*, es promocionado a *Integer*. El objeto de la clase *Integer* es convertido en un número entero (el índice del elemento seleccionado) mediante la función *intValue*.

```
void lista_itemStateChanged(ItemEvent e) {
    indice=((Integer)e.getItem()).intValue();
    //...
}
```

Respuesta a las acciones del usuario sobre un conjunto de tres botones



botones2: [BotonesApplet2.java](#)



Propósito

Ya estudiamos la respuesta a la [acción de pulsar sobre un botón](#). Diseñamos un applet contiene un conjunto de tres botones situados en la parte superior del applet. El nombre de los botones es el de los colores primarios: azul, verde y rojo. Al pulsar un botón se pinta un rectángulo de dicho color.

Diseño

Crear el applet, situar tres botones en la parte superior.

Cambiar sus propiedades **name** y **label** en sus respectivas hojas de propiedades

Establecer [FlowLayout](#), como gestor de diseño del applet. Cambiar la propiedad **hgap** para separar horizontalmente los botones.

Crear un array *colores* de objetos de la clase [Color](#) con los tres colores primarios: rojo, verde y azul.

```
final Color[] colores={Color.red, Color.green, Color.blue};
```

Redefinir la función *paint* para pintar un rectángulo

Respuesta a las acciones del usuario

Nuestra primera intención será la de seleccionar en el panel del diseño cada uno de los botones y en sus correspondientes paneles Events se hace doble-clic sobre el editor asociado a *actionPerformed*. A continuación, veremos cómo se puede elaborar una respuesta única a las acciones sobre un conjunto de

controles que responden de forma semejante.

Tres funciones respuesta

El IDE de JBuilder [genera el código](#) que crea tres objetos de otras tantas clases anónimas y asocia cada uno de ellos con el correspondiente botón mediante *addActionListener* (Anonymous adapter)

```
btnRojo.setLabel("Rojo");
    btnRojo.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            btnRojo_actionPerformed(e);
        }
    });
btnVerde.setLabel("Verde");
    btnVerde.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            btnVerde_actionPerformed(e);
        }
    });
btnAzul.setLabel("Azul");
    btnAzul.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            btnAzul_actionPerformed(e);
        }
    });
```

Ahora, solamente nos queda definir cada una de las funciones respuesta.

```
void btnRojo_actionPerformed(ActionEvent e) {
    indice=0;
    repaint();
}
void btnVerde_actionPerformed(ActionEvent e) {
    indice=1;
    repaint();
}
void btnAzul_actionPerformed(ActionEvent e) {
    indice=2;
    repaint();
}
```

A continuación, redefinimos la función *paint*, para dibujar un rectángulo del color seleccionado al pulsar el botón correspondiente.

```
public void paint(Graphics g){
    g.setColor(colores[indice]);
    g.fillRect(20, 50, 100, 50);
}
```

Ahora bien, se puede observar que en las funciones respuesta, el código está repetido. Para escribir un código más compacto, definimos una única función respuesta para todos los botones denominada *botones_actionPerformed*. Obtenemos el botón que ha sido pulsado, un objeto de la clase base *Object* mediante *getSource*. Alternativamente, se puede obtener el botón que ha sido pulsado mediante *getActionCommand*, la comparación se establece entonces con el título o etiqueta de los botones. Véase el primer apartado, [información acerca del suceso](#).

```
void botones_actionPerformed(ActionEvent e) {
    Object boton=e.getSource();
    if(boton.equals(btnRojo)){
        indice=0;
    }else if(boton.equals(btnVerde)){
        indice=1;
    }else{
        indice=2;
    }
    repaint();
}
```

El código fuente completo de este ejemplo, es el siguiente

```
package botones2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonesApplet2 extends Applet {
    Button btnRojo = new Button();
    Button btnVerde = new Button();
    Button btnAzul = new Button();
    FlowLayout flowLayout1 = new FlowLayout();
    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice;

    public void init() {
```

```

        btnRojo.setLabel("Rojo");
        btnRojo.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                botones_actionPerformed(e);
            }
        });
        btnVerde.setLabel("Verde");
        btnVerde.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                botones_actionPerformed(e);
            }
        });
        btnAzul.setLabel("Azul");
        btnAzul.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                botones_actionPerformed(e);
            }
        });
        flowLayout1.setHgap(10);
        this.setLayout(flowLayout1);
        this.add(btnRojo, null);
        this.add(btnVerde, null);
        this.add(btnAzul, null);
    }

    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(20, 50, 100, 50);
    }

    void botones_actionPerformed(ActionEvent e) {
        Object boton=e.getSource();
        if(boton.equals(btnRojo)){
            indice=0;
        }else if(boton.equals(btnVerde)){
            indice=1;
        }else{
            indice=2;
        }
        repaint();
    }
}

```

Un único objeto maneja los sucesos de un grupo de botones



botones1: [BotonesApplet1.java](#)

Podemos simplificar aún más el código, podemos pensar en un único objeto *accion* de una clase que implemente el interface *ActionListener* esté interesado en las acciones sobre los tres botones. Emplearemos el otro modo que tiene [JBuilder de generar el código](#) (Standard adapter) que maneja los sucesos.

Definimos una clase que implemente el interface *ActionListener* y defina la función *actionPerformed*.

```
class AccionBotones implements ActionListener{
    private BotonesApplet1 applet;
    public AccionBotones(BotonesApplet1 applet){
        this.applet=applet;
    }
    public void actionPerformed(ActionEvent ev){
        applet.botones_actionPerformed(ev);
    }
}
```

Creamos un objeto *accion* de la clase *AccionBotones* y le pasamos el applet. **this** inicializa el miembro dato *applet* de la clase *AccionBotones*. Desde dicho objeto llamamos a la función respuesta denominada *botones_actionPerformed*, en la definición de *actionPerformed*.

```
AccionBotones accion=new AccionBotones(this);
```

Asociamos mediante la función *addActionListener* cada uno de los botones con el objeto *accion* que registra las acciones sobre dichos botones.

```
btnRojo.addActionListener(accion);
btnVerde.addActionListener(accion);
btnAzul.addActionListener(accion);
```

Definimos la función respuesta *botones_actionPerformed*, la tarea a realizar, cuando se pulsa sobre alguno de los botones de la misma forma que en el ejemplo anterior.

El código completo de este ejemplo, es el siguiente

```

package botones1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class BotonesApplet1 extends Applet {
    Button btnRojo = new Button();
    Button btnVerde = new Button();
    Button btnAzul = new Button();
    FlowLayout flowLayout1 = new FlowLayout();
    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice;

    public void init() {
        btnRojo.setLabel("Rojo");
        btnVerde.setLabel("Verde");
        btnAzul.setLabel("Azul");
        AccionBotones accion=new AccionBotones(this);
        btnRojo.addActionListener(accion);
        btnVerde.addActionListener(accion);
        btnAzul.addActionListener(accion);
        flowLayout1.setHgap(10);
        this.setLayout(flowLayout1);
        this.add(btnRojo, null);
        this.add(btnVerde, null);
        this.add(btnAzul, null);
    }

    public void botones_actionPerformed(ActionEvent ev){
        Object boton=ev.getSource();
        if(boton.equals(btnRojo)){
            indice=0;
        }else if(boton.equals(btnVerde)){
            indice=1;
        }else{
            indice=2;
        }
        repaint();
    }

    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(20, 50, 100, 50);
    }
}

```

```
}  
  
class AccionBotones implements ActionListener{  
    private BotonesApplet1 applet;  
    public AccionBotones(BotonesApplet1 applet){  
        this.applet=applet;  
    }  
    public void actionPerformed(ActionEvent ev){  
        applet.botones_actionPerformed(ev);  
    }  
}
```

Archivos y directorios



[Entrada/salida](#)

[La clase File](#)

[Creación de un filtro](#)

La clase *File*



archivo1: [Filtro.java](#), [ArchivoApp1.java](#)

Antes de proceder al estudio de las clases que describen la entrada/salida vamos a estudiar la clase *File*, que nos proporciona información acerca de los archivos, de sus atributos, de los directorios, etc. También explicaremos como se crea un filtro mediante el [interface](#) *FilenameFilter* para obtener la lista de los archivos que tengan por ejemplo, la extensión **.java**.

La clase *File* tiene tres constructores

- *File(String path)*
- *File(String path, String name)*
- *File(File dir, String name)*

El parámetro *path* indica el camino hacia el directorio donde se encuentra el archivo, y *name* indica el nombre del archivo. Los métodos más importantes que describe esta clase son los siguientes:

- *String getName()*
- *String getPath()*
- *String getAbsolutePath()*
- *boolean exists()*
- *boolean canWrite()*
- *boolean canRead*
- *boolean isFile()*
- *boolean isDirectory()*
- *boolean isAbsolute()*
- *long lastModified()*
- *long length()*
- *boolean mkdir()*

- *boolean mkdirs()*
- *boolean renameTo(File dest);*
- *boolean delete()*
- *String[] list()*
- *String[] list(FilenameFilter filter)*

Mediante un ejemplo explicaremos algunos de los métodos de la clase *File*.

Creamos un objeto *fichero* de la clase *File*, pasándole el nombre del archivo, en este caso, el nombre del archivo código fuente *ArchivoApp1.java*.

```
File fichero=new File("ArchivoApp1.java");
```

Si este archivo existe, es decir, si la función *exists* devuelve **true**, entonces se obtiene información acerca del archivo:

- *getName* devuelve el nombre del archivo
- *getPath* devuelve el camino relativo
- *getAbsolutePath* devuelve el camino absoluto.
- *canRead* nos indica si el archivo se puede leer.
- *canWrite* nos indica si el archivo se puede escribir
- *length* nos devuelve el tamaño del archivo, si dividimos la cantidad devuelta entre 1024 obtenemos el tamaño del archivo en KB.

```
if(fichero.exists()){
    System.out.println("Nombre del archivo "+fichero.getName());
    System.out.println("Camino "+fichero.getPath());
    System.out.println("Camino absoluto "+fichero.getAbsolutePath());
    System.out.println("Se puede escribir "+fichero.canRead());
    System.out.println("Se puede leer "+fichero.canWrite());
    System.out.println("Tamaño "+fichero.length());
}
```

La salida del programa es la siguiente:

```
Nombre del arachivo ArchivoApp1.java
Camino ArchivoApp1.java
Camino absoluto c:\JBuilder2\myNumerico\archivo1\ArchivoApp1.java
Se puede escribir true
Se puede leer true
Tamaño 1366
```

Recuérdese que en Windows 95/98 se puede obtener las propiedades de un archivo, seleccionado dicho archivo y eligiendo Propiedades en el menú flotante que aparece al pulsar el botón derecho del ratón .



Para obtener la lista de los archivos del directorio actual se crea un nuevo objeto de la clase *File*

```
fichero=new File(".");
```

Para obtener la lista de los archivos que contiene este directorio se llama a la función miembro *list*, la cual nos devuelve un array de strings.

```
String[] listaArchivos=fichero.list();
for(int i=0; i<listaArchivos.length; i++){
    System.out.println(listaArchivos[i]);
}
```

La salida es la siguiente

```

archivol.jpr
archivol.html
ArchivoApp1.java
ArchivoApp1.~jav
Filtro.java
Filtro.~jav

```

Creación de un filtro

Un filtro es un objeto de una clase que implemente el interface *FilenameFilter*, y tiene que redefinir la única función del [interface](#) denominada *accept*. Esta función devuelve un dato de tipo **boolean**. En este caso, la hemos definido de forma que si el nombre del archivo termina con una determinada extensión devuelve **true** en caso contrario devuelve **false**. La función *endsWith* de [la clase String](#) realiza esta tarea tal como se ve en la porción de código que viene a continuación. La extensión se le pasa al constructor de la clase *Filtro* para inicializar el miembro dato *extension*.

```

import java.io.*;

public class Filtro implements FilenameFilter{
    String extension;
    Filtro(String extension){
        this.extension=extension;
    }
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }
}

```

Para obtener la lista de archivos con extensión .java en el directorio actual, creamos un objeto de la clase *Filtro* y se lo pasamos a la función *list* miembro de la clase *File*.

```

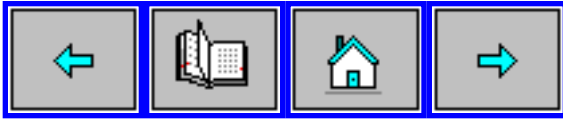
listaArchivos=fichero.list(new Filtro(".java"));
for(int i=0; i<listaArchivos.length; i++){
    System.out.println(listaArchivos[i]);
}

```

La salida es ahora la siguiente

```
ArchivoApp1.java  
Filtro.java
```

Flujos de datos



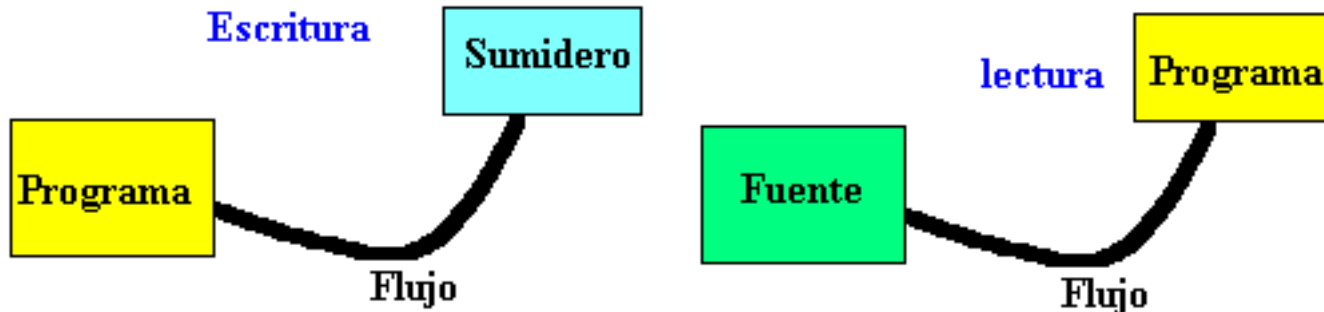
[Entrada/salida](#)

[Las jerarquías de clases](#)

[Lectura](#)

[Escritura](#)

Todos los datos fluyen a través del ordenador desde una entrada hacia una salida. Este flujo de datos se denomina también stream. Hay un flujo de entrada (input stream) que manda los datos desde el exterior (normalmente el teclado) del ordenador, y un flujo de salida (output stream) que dirige los datos hacia los dispositivos de salida (la pantalla o un archivo).



El proceso para leer o escribir datos consta de tres pasos

- Abrir el flujo de datos
- Mientras exista más información (leer o escribir) los datos
- Cerrar el flujo de datos

Las jerarquías de clases

En el lenguaje Java los flujos de datos se describen mediante clases que forman jerarquías según sea el

[tipo de dato](#) **char** Unicode de 16 bits o **byte** de 8 bits. A su vez, las clases se agrupan en jerarquías según sea su función de lectura o de escritura.

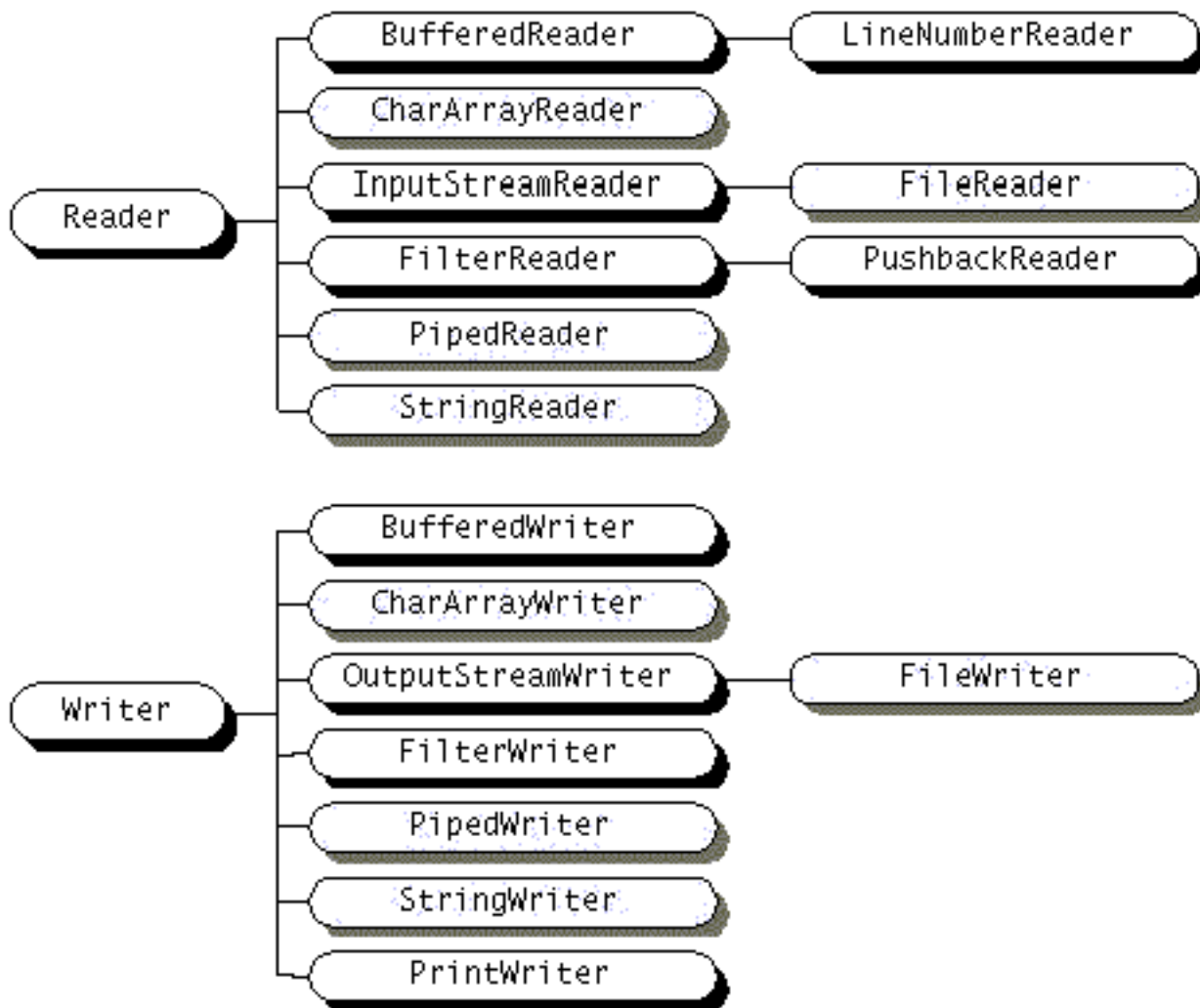
La característica de internacionalización del lenguaje Java es la razón por la que existe una jerarquía separada de clases para la lectura y escritura de caracteres.

Todas estas clases se encuentran en el [paquete](#) **java.io**, por lo que al principio del código fuente tendremos que escribir la sentencia

```
import java.io.*;
```

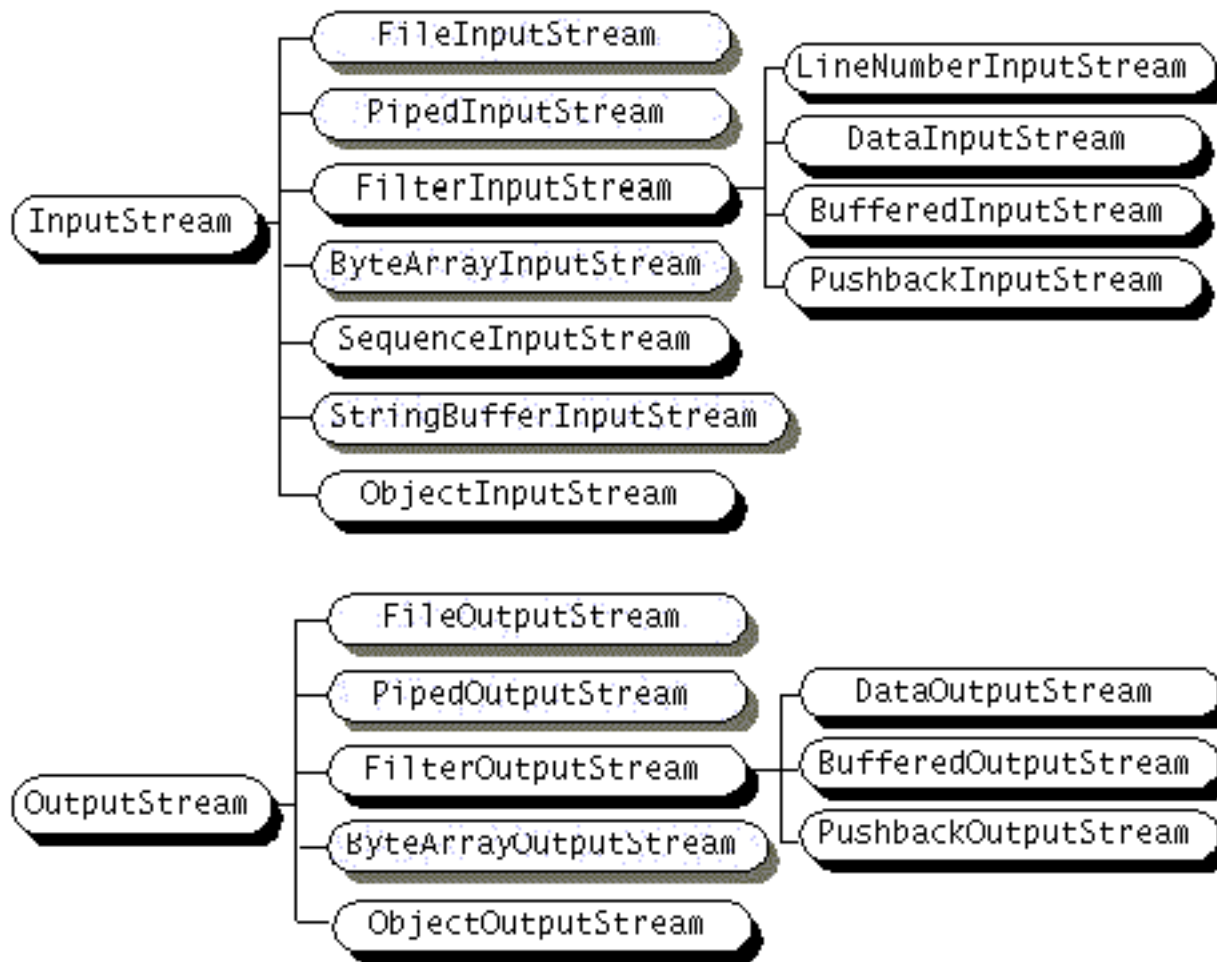
char

Unicode, 16 bits



byte

, 8 bits.



Reader y *Writer* son las clases bases de la jerarquía para los flujos de caracteres. Para leer o escribir datos binarios tales como imágenes o sonidos, se emplea otra jerarquía de clases cuyas clases base son *InputStream* y *OutputStream*.

Lectura

Las clases *Reader* e *InputStream* son similares aunque se refieren a distintos tipos de datos, lo mismo ocurre con *Writer* y *OutputStream*.

Por ejemplo, *Reader* proporciona tres métodos para leer un carácter **char** o un array de caracteres

```
int read()
int read(char buf[])
```

```
int read(char buf[], int offset, int len)
```

InputStream proporciona métodos similares para leer un **byte** o un array de bytes.

```
int read()  
int read(byte buf[])  
int read(byte buf[], int offset, int len)
```

La primera versión lee un **byte** como entero del flujo de entrada, devuelve -1 si no hay más datos que leer. La segunda versión, lee un array de bytes devolviendo el número de bytes leídos. La tercera versión, lee también, un array de bytes, pero nos permite especificar, la posición de comienzo del array en la que se empiezan a guardar los bytes, y el máximo número de bytes que se van a leer.

Dichas clases definen otras funciones miembro que no estudiaremos de momento.

Escritura

La clase *Writer* proporciona tres métodos para escribir un carácter **char** o un array de caracteres

```
int write(int c)  
int write(char buf[])  
int write(char buf[], int offset, int len)
```

La clase *OutputStream* proporciona métodos similares

```
int write(int c)  
int write(byte buf[])  
int write(byte buf[], int offset, int len)
```

Entrada/salida estándar



[Entrada/salida](#)

[Los objetos *System.in* y *System.out*](#)

[La clase *Reader*](#)

Los objetos *System.in* y *System.out*

La entrada/salida estándar (normalmente el teclado y la pantalla, respectivamente) se definen mediante dos objetos que puede usar el programador sin tener que crear flujos específicos.

La clase *System* tiene un miembro dato denominado *in* que es una instancia de la clase *InputStream* que representa al teclado o flujo de entrada estándar. Sin embargo, el miembro *out* de la clase *System* es un objeto de la clase *PrintStream*, que imprime texto en la pantalla (la salida estándar).

Para leer un carácter solamente tenemos que llamar a la función *read* desde *System.in*.

```
try{  
    System.in.read();  
}catch (IOException ex) { }
```

Obligatoriamente, el proceso de lectura ha de estar en un [bloque *try..catch*](#).

Esta porción de código es la que se ha empleado en muchas aplicaciones para detener la ejecución de una aplicación hasta que se pulse la tecla RETORNO.

Para leer un conjunto de caracteres hasta que se pulse la tecla RETORNO escribimos

```
StringBuffer str=new StringBuffer();  
char c;  
try{  
    while ((c=(char)System.in.read())!='\n'){  
        str.append(c);  
    }
```



```
    }
} catch (IOException ex) {}
```

La clase *StringBuffer* es una clase que nos permite crear strings. Contiene métodos para añadir nuevos caracteres a un buffer y convertir el resultado final en un string. Las principales funciones miembro son *insert* y *append*. Usamos una versión de esta última función para añadir un carácter al final de un objeto de la clase *StringBuffer*.

Para convertir un objeto *str* de la clase *StringBuffer* a *String* se usa la función miembro [*toString*](#). Esta llamada se hace de forma implícita cuando dicho objeto se le pasa a *System.out.println*.

```
System.out.println(str);
```

Finalmente, se ha de hacer notar, que la [*función read*](#) miembro de *InputStream* devuelve un **int** que es promocionado a **char**.

La clase *Reader*



teclado2: [TecladoApp2.java](#)

Existe la posibilidad de conectar el objeto *System.in* con un objeto de la clase *InputStreamReader* para leer los caracteres tecleados por el usuario.

Esta conexión se realiza mediante la sentencia

```
Reader entrada=new InputStreamReader(System.in);
```

Para leer una sucesión de caracteres se emplea un código similar

```
StringBuffer str=new StringBuffer();
char c;
try{
    Reader entrada=new InputStreamReader(System.in);
    while ((c=(char)entrada.read())!='\n'){
        str.append(c);
    }
}catch(IOException ex){}
```

Para imprimir los caracteres leídos se escribe como en la sección anterior

```
System.out.println(str);
```



teclado1: [TecladoApp1.java](#)

Podemos usar la segunda versión de la [función read](#) para leer el conjunto de caracteres tecleados por el usuario.

```
char[] buffer=new char[255];
try{
    Reader entrada=new InputStreamReader(System.in);
    int numBytes=entrada.read(buffer);
    System.out.println("Número de bytes leídos "+numBytes);
}catch(IOException ex){}
```

En esta segunda porción de código, se lee un conjunto de caracteres hasta que se pulsa la tecla RETORNO, los caracteres se guardan en el array *buffer*. La función *read* devuelve el número de caracteres leídos.

Para imprimir los caracteres leídos se crea un objeto *str* de la clase *String* a partir de un array de caracteres *buffer*, empleando uno de los constructores de dicha clase. A continuación, se imprime el string *str*.

```
String str=new String(buffer);
System.out.println(str);
```

Entrada/salida a un archivo en disco



[Entrada/salida](#)

[Lectura de un archivo de texto](#)

[Lectura/escritura](#)

Lectura de un archivo



archivo2: [ArchivoApp2.java](#)

El proceso de lectura de un archivo de texto es similar a la lectura desde el [dispositivo estándar](#). Creamos un objeto *entrada* de la clase *FileReader* en vez de *InputStreamReader*. El final del archivo viene dado cuando la función *read* devuelve -1. El resto del código es similar.

```
FileReader entrada=null;
StringBuffer str=new StringBuffer();
try {
    entrada=new FileReader("ArchivoApp2.java");
    int c;
    while((c=entrada.read())!=-1){
        str.append((char)c);
    }
}catch (IOException ex) {}
```

Para mostrar el archivo de texto en la pantalla del monitor, se imprime el contenido del objeto *str* de la clase *StringBuffer*.

```
System.out.println(str);
```

Una vez concluído el proceso de lectura, es conveniente cerrar el flujo de datos, esto se realiza en una [cláusula finally](#) que siempre se llama independientemente de que se produzcan o no errores en el proceso de lectura/escritura.

```
}finally{
    if(entrada!=null){
```

```

        try{
            entrada.close();
        }catch(IOException ex){}
    }
}

```

El código completo de este ejemplo es el siguiente:

```

public class ArchivoApp2 {
    public static void main(String[] args) {
        FileReader entrada=null;
        StringBuffer str=new StringBuffer();
        try {
            entrada=new FileReader("ArchivoApp2.java");
            int c;
            while((c=entrada.read())!=-1){
                str.append((char)c);
            }
            System.out.println(str);
            System.out.println("-----");
        }catch (IOException ex) {
            System.out.println(ex);
        }finally{
            //cerrar los flujos de datos
            if(entrada!=null){
                try{
                    entrada.close();
                }catch(IOException ex){}
            }
            System.out.println("el bloque finally siempre se ejecuta");
        }
    }
}

```

Lectura/escritura



archivo3: [ArchivoApp3.java](#)

Los pasos para leer y escribir en disco son los siguientes:

1. Se crean dos objetos de las clases *FileReader* y *FileWriter*, llamando a los respectivos constructores a los que se les pasa los nombres de los archivos o bien, objetos de la [clase File](#), respectivamente

```
entrada=new FileReader("ArchivoApp3.java");
salida=new FileWriter("copia.java");
```

2. Se lee mediante *read* los caracteres del flujo de entrada, hasta llegar al final (la función *read* devuelve entonces -1), y se escribe dichos caracteres en el flujo de salida mediante *write*.

```
while((c=entrada.read())!=-1){
    salida.write(c);
}
```

3. Finalmente, se cierran ambos flujos llamando a sus respectivas funciones *close* en bloques **try..catch**

```
entrada.close();
salida.close();
```

El código completo de este ejemplo que crea un archivo copia del original, es el siguiente

```
import java.io.*;

public class ArchivoApp3 {
    public static void main(String[] args) {
        FileReader entrada=null;
        FileWriter salida=null;

        try {
            entrada=new FileReader("ArchivoApp3.java");
            salida=new FileWriter("copia.java");
            int c;
            while((c=entrada.read())!=-1){
                salida.write(c);
            }
        }catch (IOException ex) {
            System.out.println(ex);
        }finally{
            //cerrar los flujos de datos
            if(entrada!=null){
                try{
                    entrada.close();
                }catch(IOException ex){}
            }
            if(salida!=null){
```

```
        try{
            salida.close();
        }catch(IOException ex){}
    }
    System.out.println("el bloque finally siempre se ejecuta");
}
}
```

Cuando se trate de leer y escribir datos binarios se sustituye *FileReader* por *FileInputStream* y *FileWriter* por *FileOutputStream*. De hecho, si se realiza esta sustitución en el código fuente de este ejemplo, los resultados no cambian.

Leer y escribir datos primitivos



[Entrada/salida](#)

[Los flujos de datos *DataInputStream* y *DataOutputStream*](#)

[Ejemplo: un pedido](#)

[El final del archivo](#)

Los flujos de datos *DataInputStream* y *DataOutputStream*

La clase *DataInputStream* es útil para leer datos del tipo primitivo de una forma portable. Esta clase tiene un sólo constructor que toma un objeto de la clase *InputStream* o sus derivadas como parámetro.

Se crea un objeto de la clase *DataInputStream* vinculándolo a un un objeto *FileInputStream* para leer desde un archivo en disco denominado pedido.txt..

```
FileInputStream fileIn=new FileInputStream("pedido.txt");
DataInputStream entrada=new DataInputStream(fileIn);
```

o en una sola línea

```
DataInputStream entrada=new DataInputStream(new FileInputStream("pedido.txt"));
```

La clase *DataInputStream* define diversos métodos *readXXX* que son variaciones del método *read* de la [clase base](#) para leer datos de tipo primitivo

```
boolean readBoolean();
byte readByte();
int readUnsignedByte();
short readShort();
int readUnsignedShort();
char readChar();
int readInt();
String readLine();
long readLong();
float readFloat();
double readDouble();
```

La clase *DataOutputStream* es útil para escribir datos del tipo primitivo de una forma portable. Esta clase tiene un sólo constructor que toma un objeto de la clase *OutputStream* o sus derivadas como parámetro.

Se crea un objeto de la clase *DataOutputStream* vinculándolo a un un objeto *FileOutputStream* para escribir en un archivo en disco denominado pedido.txt..

```
FileOutputStream fileOut=new FileOutputStream("pedido.txt");
DataOutputStream salida=new DataOutputStream(fileOut);
```

o en una sola línea

```
DataOutputStream salida=new DataOutputStream(new FileOutputStream("pedido.txt"));
```

La clase *DataOutputStream* define diversos métodos *writeXXX* que son variaciones del método *write* [de la clase base](#) para escribir datos de tipo primitivo

```
void writeBoolean(boolean v);
void writeByte(int v);
void writeBytes(String s);
void writeShort(int v);
void writeChars(String s);
void writeChar(int v);
void writeInt(int v);
void writeLong(long v);
void writeFloat(float v);
void writeDouble(double v);
```

Ejemplo: un pedido

 **archivo7:** [ArchivoApp7.java](#)

En este ejemplo, se escriben datos a un archivo y se leen del mismo, que corresponden a un pedido

- La descripción del item, un objeto de la clase *String*
- El número de unidades, un dato del tipo primitivo **int**
- El precio de cada item, un dato de tipo **double**.

Observamos en la tabla y en el código el nombre de las funciones que leen y escriben los distintos tipos de datos.

| | Escritura | Lectura |
|-------------------|--------------------|-------------------|
| Un carácter | <i>writeChar</i> | <i>readChar</i> |
| Un entero | <i>writeInt</i> | <i>readInt</i> |
| Un número decimal | <i>writeDouble</i> | <i>readDouble</i> |
| Un string | <i>writeChars</i> | <i>readLine</i> |

Veamos el código que escribe los datos a un archivo pedido.txt en disco

1. Se parte de los datos que se guardan en los arrays denominados *descripciones*, *unidades* y *precios*
2. Se crea un objeto de la clase *DataOutputStream* vinculándolo a un objeto *FileOutputStream* para escribir en un archivo en disco denominado pedido.txt..
3. Se escribe en el flujo de salida los distintos datos llamando a las distintas versiones de la función *writeXXX* según el tipo de dato (segunda columna de la tabla).
4. Se cierra el flujo de salida, llamando a su función miembro *close*.

```
double[] precios={1350, 400, 890, 6200, 8730};
int[] unidades={5, 7, 12, 8, 30};
String[] descripciones={"paquetes de papel", "lápices", "bolígrafos", "carteras",
"mesas"};

DataOutputStream salida=new DataOutputStream(new FileOutputStream("pedido.txt"));
for (int i=0; i<precios.length; i++) {
    salida.writeChars(descripciones[i]);
    salida.writeChar('\n');
    salida.writeInt(unidades[i]);
    salida.writeChar('\t');
    salida.writeDouble(precios[i]);
}
salida.close();
```

Para leer bien los datos, el string ha de separarse del siguiente dato con un carácter nueva línea '\n'. Esto no es necesario si el string se escribe en el último lugar, después de los números. Por otra parte, el carácter tabulador como separador no es estrictamente necesario.

Veamos el código que lee los datos a un archivo pedido.txt en disco

1. Se crea un objeto de la clase *DataInputStream* vinculándolo a un objeto *FileInputStream* para leer en un archivo en disco denominado pedido.txt..
2. Se lee el flujo de entrada los distintos datos en el mismo orden en el que han sido escritos, llamando a las distintas versiones de la función *readXXX* según el tipo de dato (tercera columna de la tabla).
3. Se guardan los datos leídos en memoria en las variables denominadas *descripcion*, *unidad* y *precio* y se usan para distintos propósitos
4. Se cierra el flujo de entrada, llamando a su función miembro *close*.

```
double precio;
int unidad;
String descripcion;
double total=0.0;

DataInputStream entrada=new DataInputStream(new FileInputStream("pedido.txt"));
try {
    while ((descripcion=entrada.readLine())!=null) {
        unidad=entrada.readInt();
        entrada.readChar(); //lee el carácter tabulador
        precio=entrada.readDouble();
        System.out.println("has pedido "+unidad+" "+descripcion+" a "+precio+"
pts.");
    }
}
```

```

        total=total+unidad*precio;
    }
} catch (EOFException e) {}
System.out.println("por un TOTAL de: "+total+" pts.");
entrada.close();

```

Como vemos en esta porción de código, según se van leyendo los datos del archivo, se imprimen y se calcula el precio total del pedido.

```

        System.out.println("has pedido "+unidad+" "+descripcion+" a "+precio+"
pts.");
        total=total+unidad*precio;

```

El final del archivo

El final del archivo se detecta cuando la función *readLine* devuelve **null**. Alternativamente, cuando se alcanza el final del archivo se produce una excepción del tipo *EOFException*. Podemos comprobarlo del siguiente modo

Si escribimos la siguiente porción de código

```

try {
    while(true){
        descripcion=entrada.readLine();
        unidad=entrada.readInt();
        entrada.readChar();          //lee el carácter tabulador
        precio=entrada.readDouble();
        System.out.println("has pedido "+unidad+" "+descripcion+" a
"+precio+" pts.");
        total=total+unidad*precio;
    }
} catch (EOFException e) {
    System.out.println("Excepción cuando se alcanza el final del archivo");
}

```

Cuando se alcanza el final del archivo se produce una excepción del tipo *EOFException* que interrumpe la ejecución del bucle indefinido al ser capturada por el correspondiente bloque **catch**, el cual imprime en la pantalla el mensaje "Excepción cuando se alcanza el final del archivo".

Si escribimos la siguiente porción de código

```

try {
    while ((descripcion=entrada.readLine())!=null) {
        unidad=entrada.readInt();
        entrada.readChar();          //lee el carácter tabulador
        precio=entrada.readDouble();
        System.out.println("has pedido "+unidad+" "+descripcion+" a
"+precio+" pts.");
        total=total+unidad*precio;
    }
}

```

```
    }  
    System.out.println("Final del archivo");  
}catch (EOFException e) {  
    System.out.println("Excepción cuando se alcanza el final del archivo");  
}
```

Se imprime "Final de archivo" ya que cuando *readLine* toma el valor **null** (no hay más que leer) se sale del bucle **while**, y por tanto, no se lanza ninguna excepción.

Leer y escribir objetos



[Entrada/salida](#)

[El interface *Serializable*](#)

[Lectura/escritura](#)

[El modificador *transient*](#)

[Objetos compuestos](#)

[La herencia](#)

[Serialización personalizada](#)

Java ha añadido una interesante faceta al lenguaje denominada serialización de objetos que permite convertir cualquier objeto cuya clase implemente el interface *Serializable* en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original. Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en un ordenador que corra bajo Windows 95/98, serializarlo y enviarlo a través de la red a una estación de trabajo que corra bajo UNIX donde será correctamente reconstruido. No tenemos que preocuparnos, en absoluto, de las diferentes representaciones de datos en los distintos ordenadores.

La serialización es una característica añadida al lenguaje Java para dar soporte a

- La invocación remota de objetos (RMI)
- La persistencia

La invocación remota de objetos permite a los objetos que viven en otros ordenadores comportarse como si vivieran en nuestra propia máquina. La serialización es necesaria para transportar los argumentos y los valores de retorno.

La persistencia, es una característica importante de [los JavaBeans](#). El estado de un componente es configurado durante el diseño. La serialización nos permite guardar el estado de un componente en disco, abandonar el Entorno Integrado de Desarrollo (IDE) y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

El interface *Serializable*

Un objeto se puede serializar si implementa el [interface](#) *Serializable*. Este interface no declara ninguna función miembro, se trata de un interface vacío.

```
import java.io.*;
public interface Serializable{
```

}

Para hacer una clase serializable simplemente ha de implementar el interface *Serializable*, por ejemplo, a [la clase *Lista*](#) que estudiamos en el capítulo Clases y objetos se le añade la implementación del interface

```
public class Lista implements java.io.Serializable{
    private int[] x;
    private int n;
    //otros miembros...
}
```

No tenemos que escribir ningún otro método. El método *defaultWriteObject* de la clase *ObjectOutputStream* realiza la serialización de los objetos de una clase. Este método escribe en el flujo de salida todo lo necesario para reconstruir dichos objetos:

- La clase del objeto
- La firma de la clase (class signature)
- Los valores de los miembros que no tengan los modificadores **static** o **transient**, incluyendo los miembros que se refieren a otros objetos.

El método *defaultReadObject* de la clase *ObjectInputStream* realiza la deserialización de los objetos de una clase. Este método lee el flujo de entrada y reconstruye los objetos de dicha clase.

Lectura/escritura



archivo4: [Lista.java](#) [ArchivoApp4.java](#)

Dos [flujos de datos](#) *ObjectInputStream* y *ObjectOutputStream* están especializados en la lectura y escritura de objetos. El comportamiento de estos dos flujos es similar a sus correspondientes que procesan flujos de datos primitivos [DataInputStream](#) y [DataOutputStream](#), que hemos visto en la página previa

Escribir objetos al flujo de salida *ObjectOutputStream* es muy simple y requiere los siguientes pasos:

1. Creamos un objeto de la clase *Lista*

```
Lista lista1= new Lista(new int[]{12, 15, 11, 4, 32});
```

2. Creamos un [fujo de salida a disco](#), pasándole el nombre del archivo en disco o un objeto de la clase [File](#).

```
FileOutputStream fileOut=new FileOutputStream("media.obj");
```

3. El fujo de salida *ObjectOutputStream* es el que procesa los datos y se ha de vincular a un objeto *fileOut* de la clase *FileOutputStream*.

```
ObjectOutputStream salida=new ObjectOutputStream(fileOut);
```

o en una sólo línea

```
ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("media.obj"));
```

4. El método *writeObject* escribe los objetos al flujo de salida y los guarda en un archivo en disco. Por ejemplo, un string y un objeto de la clase *Lista*.

```
salida.writeObject("guardar este string y un objeto\n");
salida.writeObject(lista1);
```

5. Finalmente, se cierran los flujos

```
salida.close();
```

```
Lista lista1= new Lista(new int[]{12, 15, 11, 4, 32});

ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("media.obj"));
salida.writeObject("guardar este string y un objeto\n");
salida.writeObject(lista1);
salida.close();
```

El proceso de lectura es paralelo al proceso de escritura, por lo que leer objetos del flujo de entrada *ObjectInputStream* es muy simple y requiere los siguientes pasos.

1. Creamos un [flujo de entrada a disco](#), pasándole el nombre del archivo en disco o un objeto de la clase [File](#).

```
FileInputStream fileIn=new FileInputStream("media.obj");
```

2. El flujo de entrada *ObjectInputStream* es el que procesa los datos y se ha de vincular a un objeto *fileIn* de la clase *FileInputStream*.

```
ObjectInputStream entrada=new ObjectInputStream(fileIn);
```

o en una sólo línea

```
ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("media.obj"));
```

3. El método *readObject* lee los objetos del flujo de entrada, en el mismo orden en el que ha sido escritos. Primero un string y luego, un objeto de la clase *Lista*.

```
String str=(String)entrada.readObject();
Lista obj1=(Lista)entrada.readObject();
```

4. Se realizan tareas con dichos objetos, por ejemplo, desde el objeto *obj1* de la clase *Lista* se llama a la función miembro *valorMedio*, para hallar el valor medio del array de datos, o se muestran en la pantalla

```

System.out.println("Valor medio "+obj1.valorMedio());
System.out.println("-----");
System.out.println(str+obj1);

```

5. Finalmente, se cierra los flujos

```

entrada.close();

```

```

ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("media.obj"));
String str=(String)entrada.readObject();
Lista obj1=(Lista)entrada.readObject();
System.out.println("Valor medio "+obj1.valorMedio());
System.out.println("-----");
System.out.println(str+obj1);
System.out.println("-----");
entrada.close();

```

El modificador *transient*



archivo6: [Cliente.java](#) [ArchivoApp6.java](#)

Cuando un miembro dato de una clase contiene información sensible, hay disponibles varias técnicas para protegerla. Incluso cuando dicha información es privada (el miembro dato tiene el modificador **private**) una vez que se ha enviado al flujo de salida alguien puede leerla en el archivo en disco o interceptarla en la red.

El modo más simple de proteger la información sensible, como una contraseña (password) es la de poner el modificador **transient** delante del miembro dato que la guarda.

La clase *Cliente* tiene dos miembros dato, el nombre del cliente y la contraseña o password.

Redefine la [función toString](#) miembro de la clase base *Object*. Esta función devolverá el nombre del cliente y la contraseña. En el caso de que el miembro *password* guarde el valor **null** se imprimirá el texto (no disponible).

En el cuadro que sigue se muestra el código que define la clase *Cliente*.

```

public class Cliente implements java.io.Serializable{
    private String nombre;
    private transient String passWord;
    public Cliente(String nombre, String pw) {
        this.nombre=nombre;
        passWord=pw;
    }
    public String toString(){
        String texto=(passWord==null) ? "(no disponible)" : passWord;
        texto+=nombre;
        return texto;
    }
}

```

En el cuadro siguiente se muestra los pasos para guardar un objeto de la clase *Cliente* en el archivo cliente.obj. Posteriormente, se lee el archivo para reconstruir el objeto *obj1* de dicha clase.

1. Se crea el objeto *cliente* de la clase *Cliente* pasándole el nombre del cliente "Angel" y la contraseña "xyz".
2. Se crea un flujo de salida (objeto *salida* de la clase *ObjectOutputStream*) y se asocia con un objeto de la clase *FileOutputStream* para guardar la información en el archivo cliente.obj.
3. Se escribe el objeto *cliente* en el flujo de salida mediante *writeObject*.
4. Se cierra el flujo de salida llamando a *close*.

```

Cliente cliente=new Cliente("Angel", "xyz");

ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("cliente.obj"));
salida.writeObject("Datos del cliente\n");
salida.writeObject(cliente);
salida.close();

```

Para reconstruir el objeto *obj1* de la clase *Cliente* se procede del siguiente modo:

1. Se crea un flujo de entrada (objeto *entrada* de la clase *ObjectInputStream*) y se asocia con un objeto de la clase *FileInputStream* para leer la información que gurada el archivo cliente.obj.
2. Se lee el objeto *cliente* en el flujo de salida mediante *readObject*.
3. Se imprime en la pantalla dicho objeto llamando implícitamente a su función miembro *toString*.
4. Se cierra el flujo de entrada llamando a *close*.

```

ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("cliente.obj"));
String str=(String)entrada.readObject();
Cliente obj1=(Cliente)entrada.readObject();
System.out.println("-----");
System.out.println(str+obj1);
System.out.println("-----");
entrada.close();

```


La salida del programa es

```
Datos del cliente
(no disponible) Angel
```

Lo que nos indica que la información sensible guardada en el miembro dato *password* que tiene por modificador **transient** no ha sido guardada en el archivo. En la reconstrucción del objeto *obj1* con la información guardada en el archivo el miembro dato *password* toma el valor **null**.

Objetos compuestos



archivo5: [Punto.java](#), [Rectangulo.java](#) [ArchivoApp5.java](#)

Volvemos de nuevo al estudio de [la clase Rectangulo](#) que contiene un subobjeto de la clase *Punto*.

A dichas clases se les ha añadido la redefinición de la [función toString](#) miembro de la clase base *Object* (esta redefinición no es necesaria aunque es ilustrativa para explicar el comportamiento de un objeto compuesto). Como podemos apreciar, ambas clases implementan el interface *Serializable*.

En el cuadro que sigue se muestra parte del código que define la clase *Punto*.

```
public class Punto implements java.io.Serializable{
    private int x;
    private int y;
    //otros miembros...

    public String toString(){
        return new String("(" + x + ", " + y + ")");
    }
}
```

La definición de la clase *Rectangulo* se muestra en el siguiente cuadro

```
public class Rectangulo implements java.io.Serializable{
    private int ancho ;
    private int alto ;
    private Punto origen;
    //otras funciones miembro...

    public String toString(){
        String texto=origen+" w:"+ancho+" h:"+alto;
        return texto;
    }
}
```

Como podemos observar, en la definición de *toString* de la clase *Rectangulo* se hace una llamada implícita a la función *toString* miembro de la clase *Punto*. La composición como se ha estudiado permite reutilizar el código existente.

Para guardar en un archivo un objeto de la clase *Rectangulo* hay que seguir los mismos pasos que para guardar un objeto de la clase *Lista* o de la clase *Cliente*.

```
Rectangulo rect=new Rectangulo(new Punto(10,10), 30, 60);

ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("figura.obj"));
salida.writeObject("guardar un objeto compuesto\n");
salida.writeObject(rect);
salida.close();
```

Para reconstruir un objeto de la clase *Rectangulo* a partir de los datos guardados en el archivo hay que seguir los mismos pasos que en los dos ejemplos previos.

```
ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("figura.obj"));
String str=(String)entrada.readObject();
Rectangulo obj1=(Rectangulo)entrada.readObject();
System.out.println("-----");
System.out.println(str+obj1);
System.out.println("-----");
entrada.close();
```

En el caso de que nos olvidemos de implementar el interface *Serializable* en la clase *Punto* que describe el subobjeto de la clase *Rectangulo*, se lanza una excepción, imprimiéndose en la consola.

```
java.io.NotSerializableException: archivo5.Punto.
```

La herencia



archivo8: [Figura.java](#), [ArchivoApp8.java](#)

En el apartado anterior hemos examinado la composición, ahora examinemos la herencia. En el [capítulo de la herencia](#) examinamos una jerarquía formada por una clase base denominada *Figura* y dos clases derivadas denominadas *Circulo* y *Rectangulo*.

Como podemos observar en el cuadro adjunto se han hecho dos modificaciones. La clase base *Figura* implementa el interface *Serializable* y en la clase *Circulo* en vez de usar el número PI proporcionado por la [clase Math](#), definimos una constante estática *PI* con una aproximación de 4 decimales. De este modo probamos el comportamiento de un miembro estático en el proceso de serialización.

Para serializar objetos de una jerarquía solamente la clase base tiene que implementar el interface *Serializable*

```
public abstract class Figura implements java.io.Serializable{
    protected int x;
    protected int y;
    public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}

class Circulo extends Figura{
    protected double radio;
    private static final double PI=3.1416;
    public Circulo(int x, int y, double radio){
        super(x,y);
        this.radio=radio;
    }
    public double area(){
        return PI*radio*radio;
    }
}

class Rectangulo extends Figura{
    protected double ancho, alto;
    public Rectangulo(int x, int y, double ancho, double alto){
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public double area(){
        return ancho*alto;
    }
}
```

Vamos a serializar dos objetos uno de la clase *Rectangulo* y otro de la clase *Circulo*, y a continuación reconstruiremos dichos objetos. Una vez de que dispongamos de los objetos llamaremos a las funciones *area* para calcular el área de cada una de las

figuras.

Para guardar en el archivo `figura.obj` un objeto *fig1* de la clase *Rectangulo* y otro objeto *fig2* de la clase *Circulo*, se siguen los mismos pasos que hemos estudiado en apartados anteriores

```
Figura fig1=new Rectangulo(10,15, 30, 60);
Figura fig2=new Circulo(12,19, 60);

ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("figura.obj"));
salida.writeObject("guardar un objeto de una clase derivada\n");
salida.writeObject(fig1);
salida.writeObject(fig2);
salida.close();
```

Fijarse que *fig1* y *fig2* son dos referencias de la clase base *Figura* en la que se guardan objetos de las clases derivadas *Rectangulo* y *Circulo*, respectivamente

Para leer los datos guardados en el archivo `figura.obj` y reconstruir dos objetos *obj1* y *obj2* de las clases *Rectangulo* y *Circulo* respectivamente, se procede de forma similar a la estudiada en los apartados previos.

```
ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("figura.obj"));
String str=(String)entrada.readObject();
Figura obj1=(Figura)entrada.readObject();
Figura obj2=(Figura)entrada.readObject();
System.out.println("-----");
System.out.println(obj1.getClass().getName()+" origen (" +obj1.x+", "+obj1.y+")"+"
area="+obj1.area());
System.out.println(obj2.getClass().getName()+" origen (" +obj2.x+", "+obj2.y+")"+"
area="+obj2.area());
System.out.println("-----");
entrada.close();
```

Fijarse que *obj1* y *obj2* son referencias a la clase base *Figura*. Sin embargo, cuando *obj1* llama a la función *area* nos devuelve (correctamente) el área del rectángulo y cuando, *obj2* llama a la función *area* devuelve el área del círculo.

Fijarse también que aunque *PI* es un miembro estático de la clase *Circulo*, se reconstruye el objeto *obj2* con el valor del miembro estático con el que se calcula el área del círculo

Serialización personalizada

El proceso de serialización proporcionado por el lenguaje Java es suficiente para la mayor parte de las clases, ahora bien, se puede personalizar para aquellos casos específicos.

Para personalizar la serialización, es necesario definir dos funciones miembros *writeObject* y *readObject*. El primero, controla que información es enviada al flujo de salida. La segunda, lee la información escrita por *writeObject*.

La definición de *writeObject* ha de ser la siguiente

```
private void writeObject (ObjectOutputStream s) throws IOException{
    s.defaultWriteObject();
    //...código para escribir datos
}
```

La función *readObject* ha de leer todo lo que se ha escrito con *writeObject* en el mismo orden en el que se ha escrito. Además, puede realizar otras tareas necesarias para actualizar el estado del objeto.

```
private void readObject (ObjectInputStream s) throws IOException{
    s.defaultReadObject();
    //...código para leer datos
    //...
    //actualización del estado del objeto, si es necesario
}
```

Para un control explícito del proceso de serialización la clase ha de implementar el interface *Externalizable*. La clase es responsable de escribir y de leer su contenido, y ha de estar coordinada con sus clases base para hacer esto.

La definición del interface *Externalizable* es la siguiente

```
package java.io;

public interface Externalizable extends Serializable{
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
    java.lang.ClassNotFoundException;
}
```

La Máquina Virtual Java

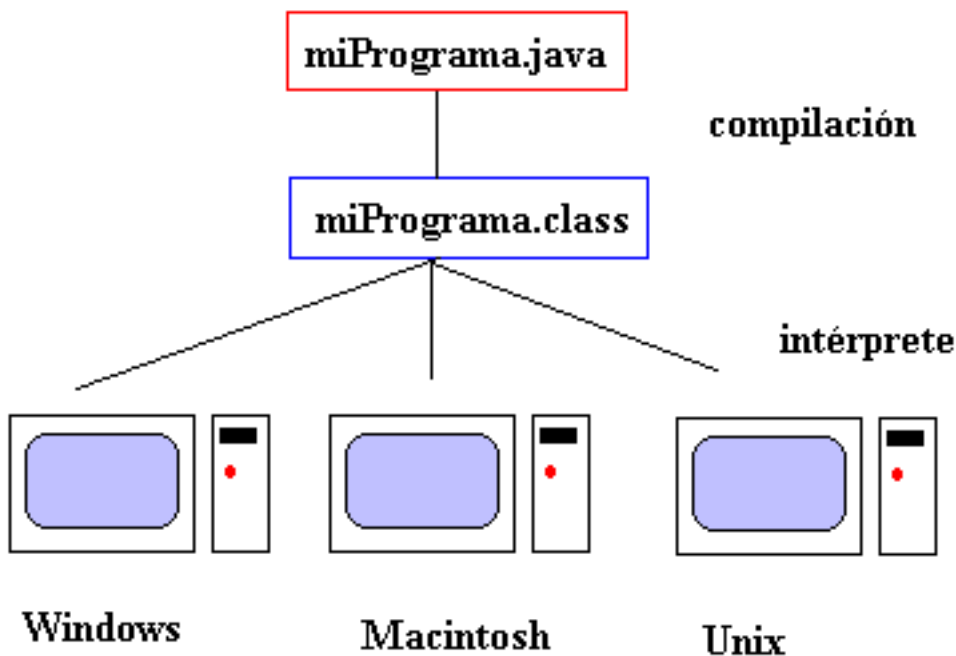


Introducción

La Máquina Virtual Java

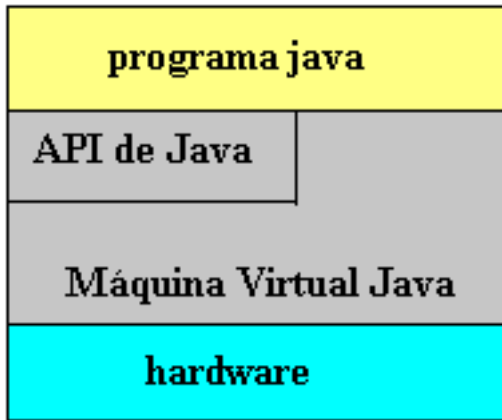
El lenguaje Java

El lenguaje Java es a la vez compilado e interpretado. Con el compilador se convierte el código fuente que reside en archivos cuya extensión es **.java**, a un conjunto de instrucciones que recibe el nombre de *bytecodes* que se guardan en un archivo cuya extensión es **.class**. Estas instrucciones son independientes del tipo de ordenador. El intérprete ejecuta cada una de estas instrucciones en un ordenador específico (Windows, Macintosh, etc). Solamente es necesario, por tanto, compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un ordenador.



Cada intérprete Java es una implementación de la Máquina Virtual Java (JVM). Los *bytecodes* posibilitan el objetivo de "write once, run anywhere", de escribir el programa una vez y que se pueda correr en cualquier plataforma que disponga de una implementación de la JVM. Por ejemplo, el mismo programa Java puede correr en Windows 98, Solaris, Macintosh, etc.

Java es, por tanto, algo más que un lenguaje, ya que la palabra Java se refiere a dos cosas inseparables: el lenguaje que nos sirve para crear programas y la Máquina Virtual Java que sirve para ejecutarlos. Como vemos en la figura, el API de Java y la Máquina Virtual Java forman una capa intermedia (Java platform) que aísla el programa Java de las especificidades del hardware (hardware-based platform).



La Máquina Virtual Java

La Máquina Virtual Java (JVM) es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador abstracto y especifica las instrucciones (*bytecodes*) que este ordenador puede ejecutar. El intérprete Java específico ejecuta las instrucciones que se guardan en los archivos cuya extensión es **.class**. Las tareas principales de la JVM son las siguientes:

- Reservar espacio en memoria para los objetos creados
- Liberar la memoria no usada (garbage collection).
- Asignar variables a registros y pilas
- Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java

Esta última tarea, es una de las más importantes que realiza la JVM. Además, las propias especificaciones del lenguaje Java contribuyen extraordinariamente a este objetivo:

- Las referencias a arrays son verificadas en el momento de la ejecución del programa
- No hay manera de manipular de forma directa los punteros
- La JVM gestiona automáticamente el uso de la memoria, de modo que no queden huecos.
- No se permiten realizar ciertas conversiones (casting) entre distintos tipos de datos.

Por ejemplo, cuando el navegador encuentra una página web con un applet, pone en marcha la JVM y proporciona la información que aparece en la [etiqueta](#) `<APPLET > ... </APPLET>`. El cargador de clases

dentro de la JVM ve que clases necesita el applet. Dentro del proceso de carga, las clases se examinan mediante un verificador que asegura que las clases contienen código válido y no malicioso. Finalmente, se ejecuta el applet.

El lenguaje Java

El lenguaje Java no está diseñado solamente para crear applets que corren en la ventana del navegador. Java es un lenguaje de propósito general, de alto nivel, y orientado a objetos.

Java es un lenguaje de programación orientado a objetos puro, en el sentido de que no hay ninguna variable, función o constante que no esté dentro de una clase. Se accede a los miembros de datos y las funciones miembro a través de los objetos y de las clases. Por razones de eficiencia, se han conservado los tipos básicos de datos, **int**, **float**, **double**, **char**, etc, similares a los del lenguaje C/C++.

Los tipos de programas más comunes que se pueden hacer con Java son los applets (se ejecutan en el navegador de la máquina cliente) y las aplicaciones (programas que se ejecutan directamente en la JVM). Otro tipo especial de programa se denomina servlet que es similar a los applets pero se ejecutan en los servidores Java.

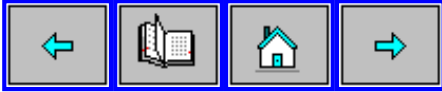
La API de Java es muy rica, está formada un conjunto de paquetes de clases que le proporcionan una gran funcionalidad. El núcleo de la API viene con cada una de las implementaciones de la JVM:

- Lo esencial: tipos de datos, clases y objetos, arrays, cadenas de caracteres (strings), subprocesos (threads), entrada/salida, propiedades del sistema, etc.
- Applets
- Manejo de la red (networking)
- Internacionalización
- Seguridad
- Componentes (JavaBeans)
- Persistencia (Object serialization)
- Conexión a bases de datos (JDBC)

Java proporciona también extensiones, por ejemplo define un API para 3D, para los servidores, telefonía, reconocimiento de voz, etc.

La primera aplicación

[Introducción](#)



[Los subdirectorios de trabajo](#)

[Crear un nuevo proyecto](#)

[La aplicación](#)

[Ejecutar la aplicación](#)

[Resumen](#)

Los programas de este curso han sido creados con el Entorno Integrado de Desarrollo (IDE) JBuilder 2.0 de Borland International, aunque el Curso de Java en sí mismo, es independiente de la herramienta de programación. El lector interesado en este curso puede emplear el IDE de su preferencia.

Vamos a ver mediante una serie de imágenes los pasos para crear una aplicación que corre en la consola, una ventana DOS en el escritorio de Windows 95/98. Estas aplicaciones solamente se crearán durante el proceso de aprendizaje del lenguaje Java, en esta primera parte del Curso

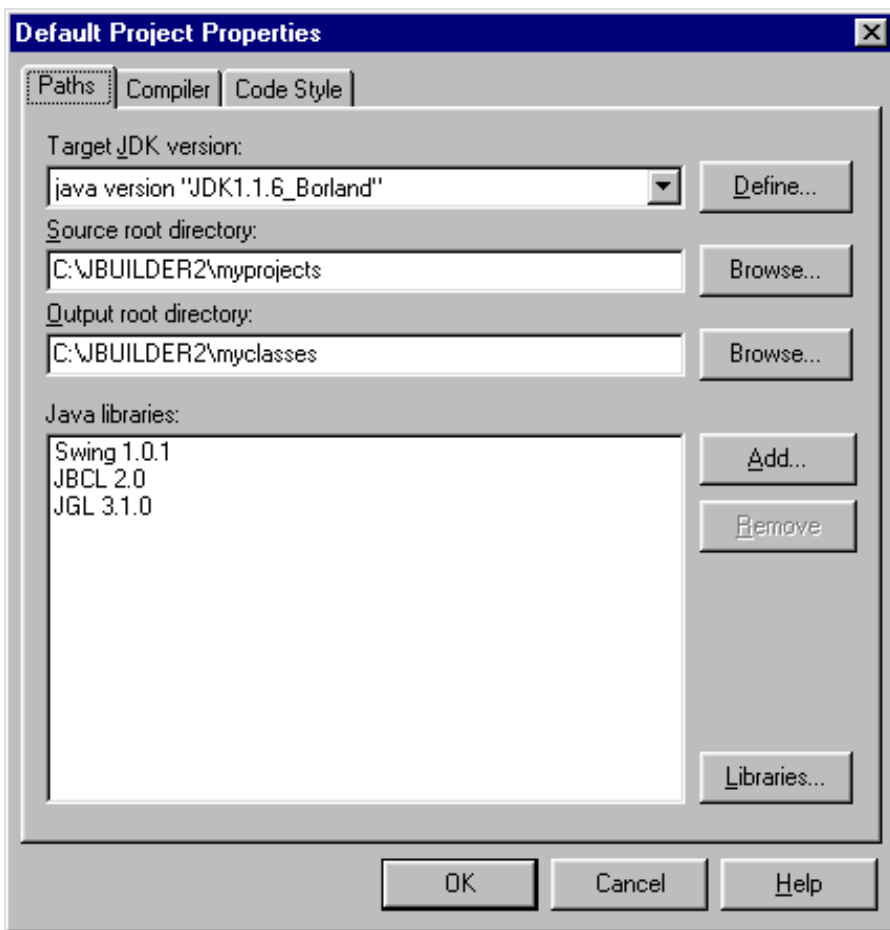
El objetivo de esta página es la de enseñar al lector a crear un proyecto, de modo que cuando vea la imagen de un disquette a lo largo de las páginas de este Curso lo asocie a un proyecto nuevo cuyo nombre (**primero**) aparece en negrita a continuación de la imagen, y cuyos componentes son archivos código fuente (PrimeroApp.java) que se pueden ver o descargar pulsando en el enlace correspondiente.



primero: [PrimeroApp.java](#)

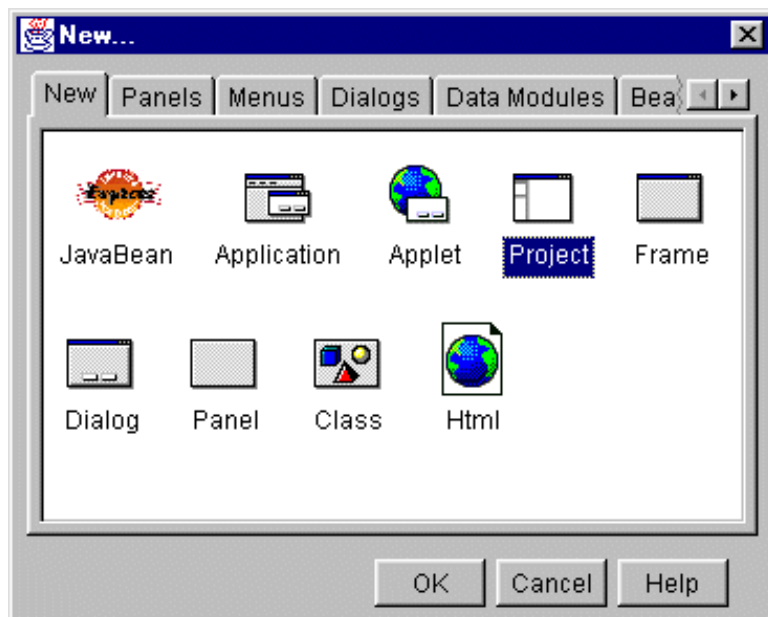
Los subdirectorios de trabajo

Por defecto el IDE de JBuilder 2.0 sitúa el código fuente (archivos .java) en el subdirectorio **myprojects**, y el código compilado (archivos .class) en el subdirectorio **myclasses**. Si queremos cambiar los subdirectorios de trabajo tenemos que seleccionar el elemento del menú **Tools/Default project properties**. En el diálogo que aparece **Default project properties**, cambiamos el nombre del subdirectorio del código fuente, en el control de edición titulado **Source root directory** o bien, pulsando en el botón titulado **Browse**. También podemos cambiar el subdirectorio donde se guardan los archivos compilados en el control de edición titulado **Output root directory** o bien, pulsando el botón asociado titulado **Browse**.



Crear un nuevo proyecto

Para crear un nuevo proyecto se selecciona en el menú **File/New**, apareciendo el diálogo **New**



Se selecciona el icono **Project**, y se pulsa en el botón titulado **OK**, o bien, se hace doble-clic sobre el icono representativo.

Aparece un asistente para la creación del proyecto, un diálogo titulado **Project Wizard: Step 1 of 1**

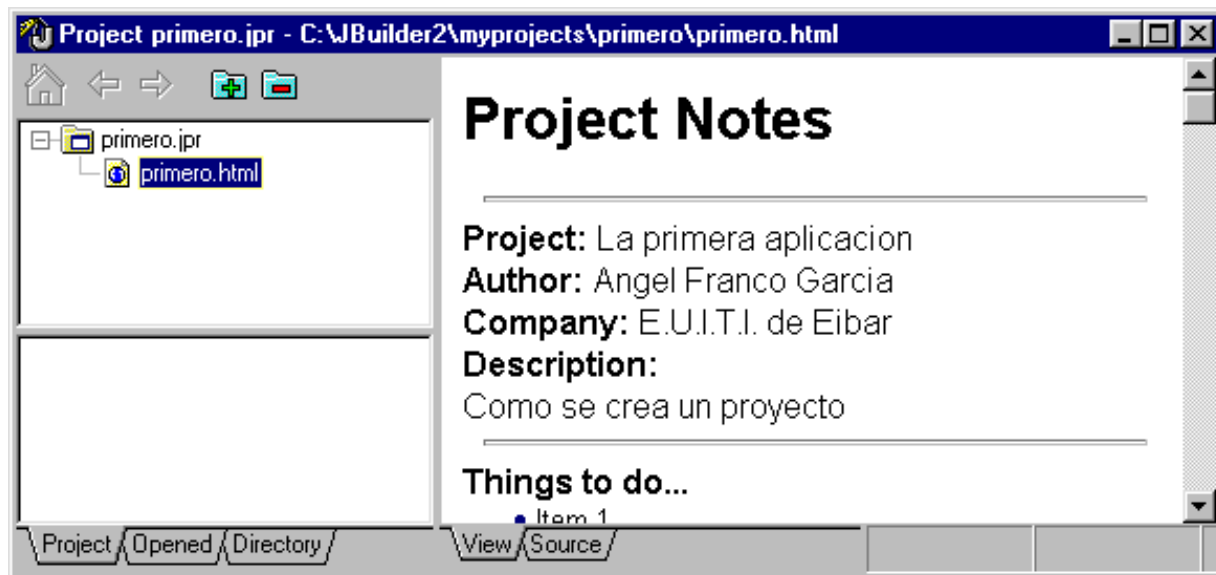


Vamos a crear un proyecto titulado **primero** y que se va a guardar en un subdirectorio denominado **primero**.

Por defecto, JBuilder sitúa los proyectos en el subdirectorio **myprojects**. Dentro de este subdirectorio se crea la carpeta titulada **primero** (**untitled1** es el valor por defecto) y crea un archivo denominado **primero.jpr** (**untitled1.jpr** por defecto). En todos, los programas ejemplo, el nombre del proyecto será idéntico al nombre de la carpeta que lo contiene.

Opcionalmente, se puede rellenar los campos titulados Title, Author, Company y Description. JBuilder genera el archivo **primero.html** que contiene la información que suministramos en dichos campos.

Al cerrar el diálogo pulsando en el botón titulado **Finish** vemos lo siguiente



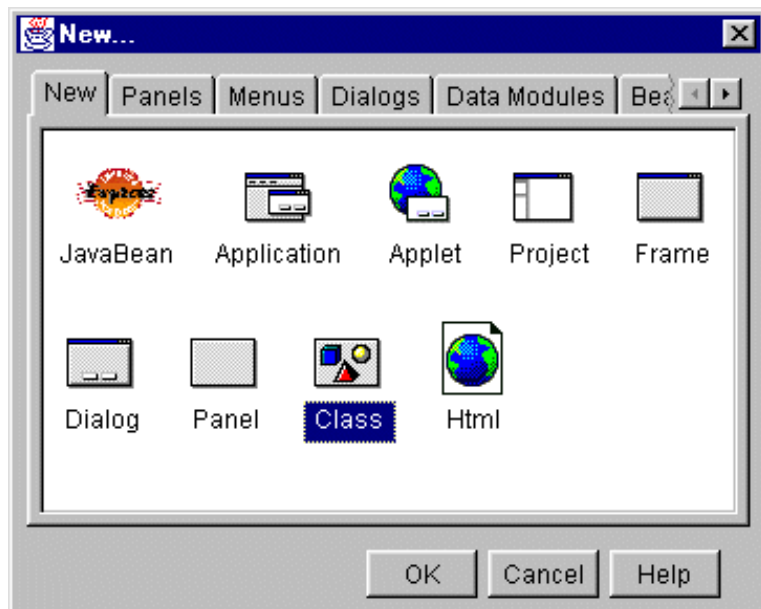
El IDE de JBuilder tiene dos ventanas, la superior tiene los menús, barra de herramientas y las paletas de los componentes. La ventana inferior está compuesta por tres paneles. El superior izquierdo es el panel de navegación, el inferior el panel de estructura y el de la derecha, más grande es el panel de contenido. El panel de contenido cambia según la pestaña que se elija (el código fuente, el diseño, etc.)

En la imagen, vemos el contenido del archivo **primero.html**. Si pulsamos en la pestaña **Source** veremos la misma información con las etiquetas HTML.

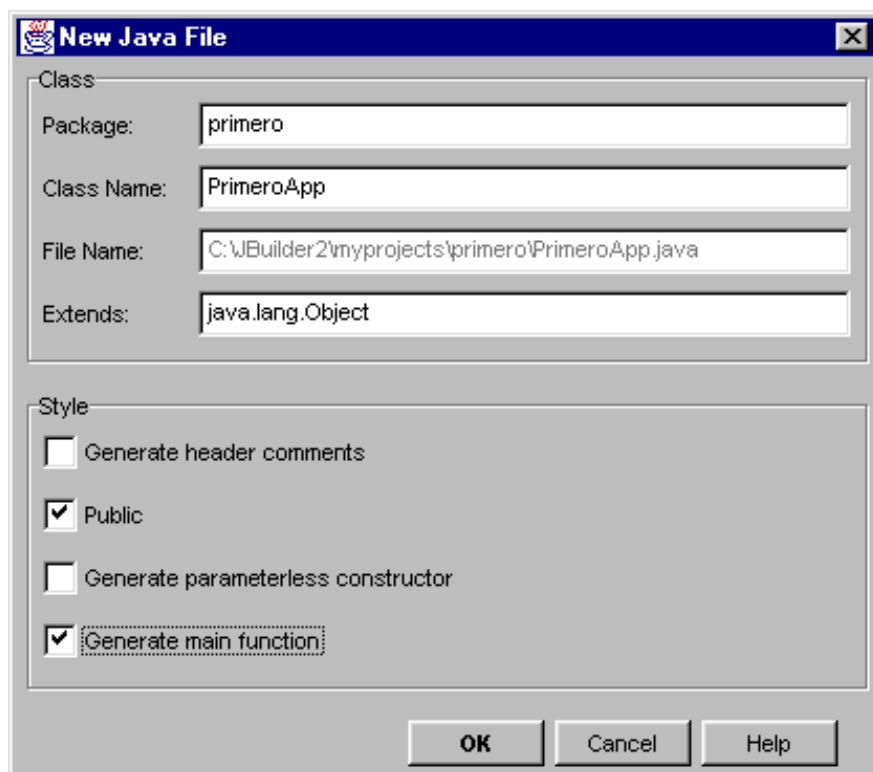
La aplicación

Para crear una aplicación hemos de crear una clase que contenga el método **main**. Para ello, seguimos los pasos siguientes:

Seleccionamos **File/New** y aparece el diálogo **New**, en el cual seleccionamos el icono titulado **Class**.

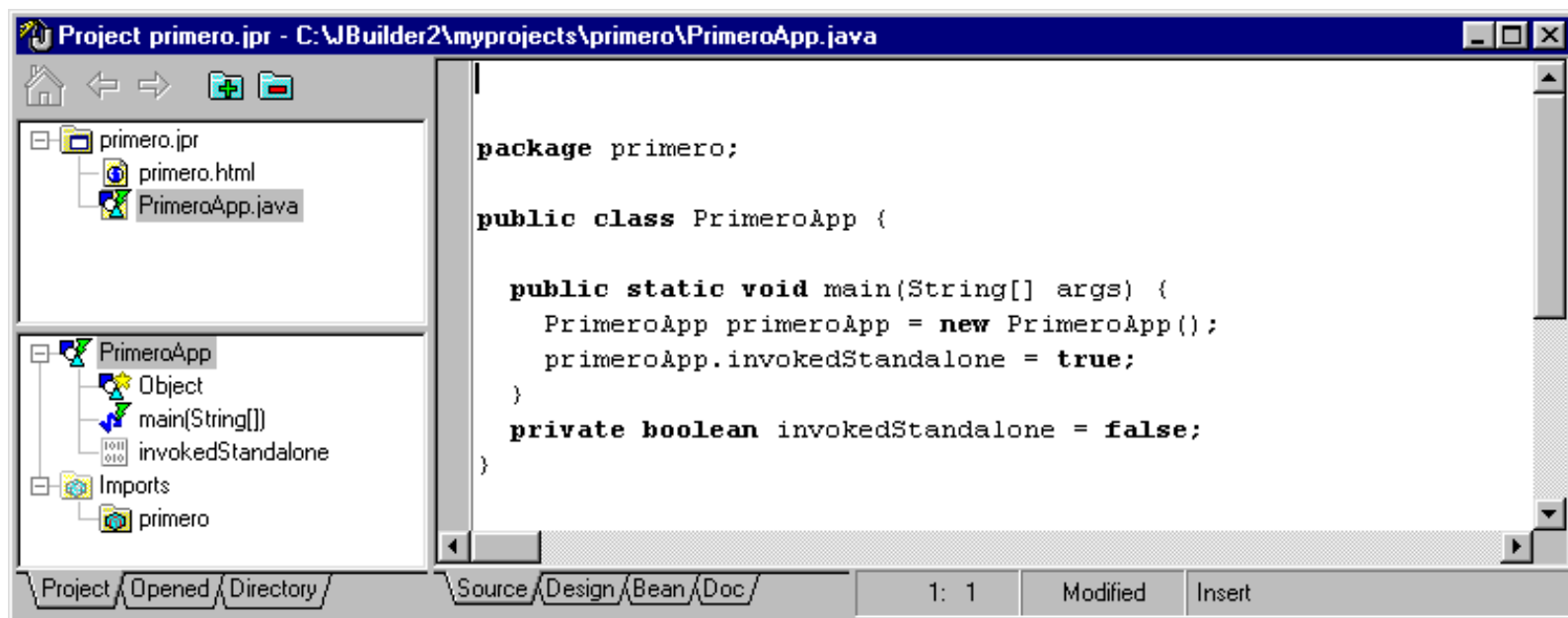


Pulsamos el botón titulado **OK**, o hacemos doble-clic sobre el icono representativo, apareciendo el diálogo titulado **New Java File**



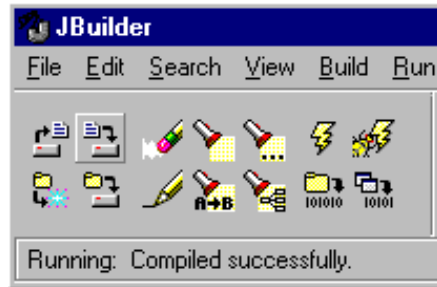
En el campo **Class Name** introducimos el nombre de la clase **PrimeroApp**. A continuación, activamos las casillas del grupo **Style** tituladas **Public** y **Generate main function**, tal como se ve en la figura

Cuando se pulsa el botón titulado **OK** se crea un nuevo archivo **PrimeroApp.java** en el subdirectorio **primero**.



En el panel de navegación (superior izquierdo) tenemos el proyecto: **primero.jpr** es el archivo proyecto, que está formado por dos archivos: **primero.html** que contiene la documentación relativa al proyecto, y **PrimeroApp.java** que es la aplicación cuyo código fuente (pestaña **Source**) vemos en el panel de contenido a la derecha.

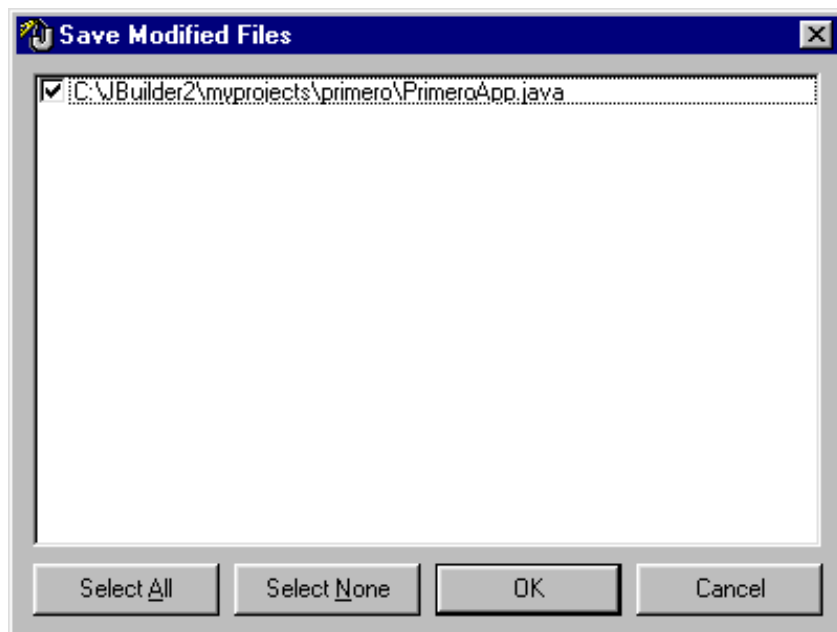
En el panel inferior izquierdo, vemos la estructura del archivo **PrimeroApp.java**, el nombre de la clase seguido por las datos y funciones miembro definidos en dicha clase.



Para guardar los cambios en un archivo, se selecciona **File/Save**, o el icono equivalente en la barra de herramientas (el segundo de la primera fila). Para guardar todos los archivos, se selecciona **File/save All**, o el icono correspondiente en la barra de herramientas (el segundo en la segunda fila).

Una vez guardados todos los cambios producidos en los archivos del proyecto, se cierra el proyecto seleccionando **File/Close** o el icono correspondiente en la barra de herramientas (el primero en la segunda fila).

Si se cierra el proyecto sin haber guardado todos los cambios aparece el diálogo **Save Modified Files**



Se pulsa el botón titulado **Select All** y a continuación el botón **OK**, o directamente **OK**.

La aplicación mínima

La aplicación mínima consta de una clase que define la función estática *main*.

```
public class PrimeroApp{
    public static void main(String[] args) {

    }
}
```

Donde **class** es la palabra clave que se emplea para definir una clase cuyo nombre es *PrimeroApp*. Dentro de la clase se define una función denominada *main*, que no devuelve nada **void** y que es estática (**static**). Como en el lenguaje C/C++ a esta función se le pueden pasar argumentos en el array de objetos del tipo *String*.

La definición de una clase está entre dos llaves de apertura { y cierre }. El cuerpo o definición de la función *main* está asimismo, entre las llaves de apertura y cierre, aquí es donde situaremos nuestro código para que la aplicación tenga alguna funcionalidad.

Varios son los conceptos que surgen en la primera aplicación, que iremos estudiando a lo largo de estos capítulos:

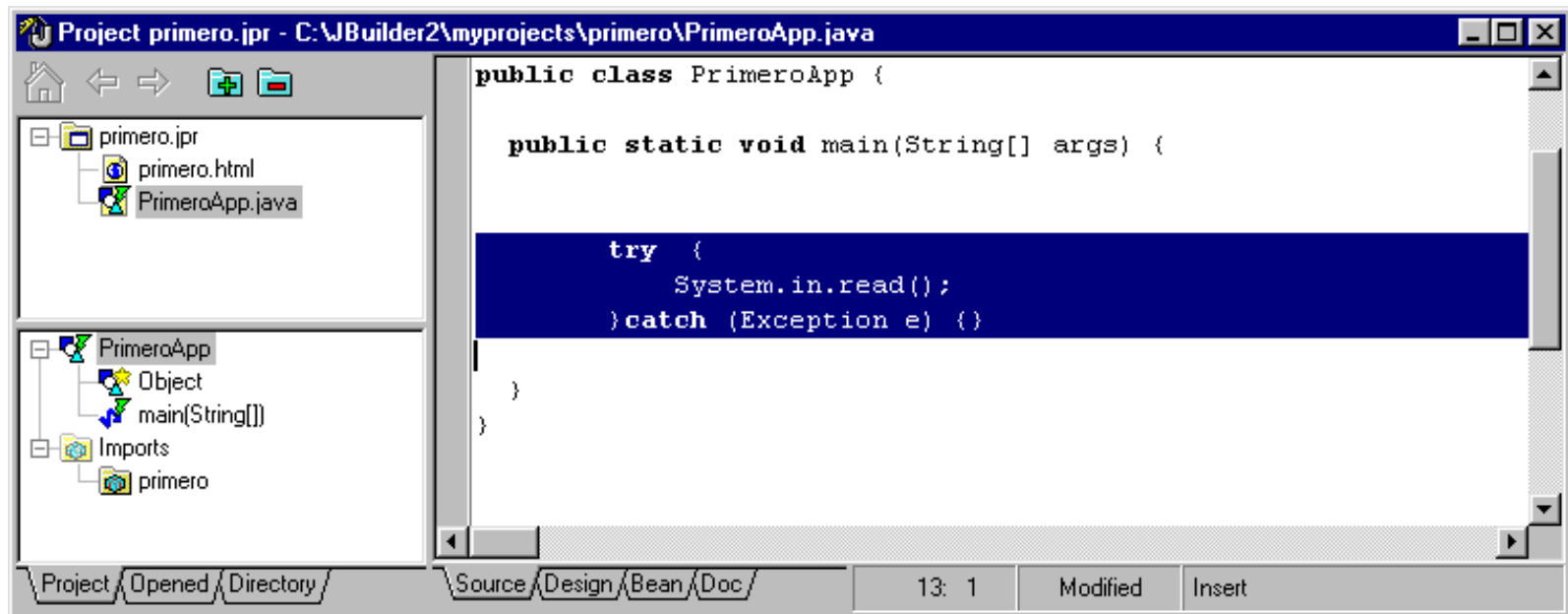
- El concepto de clase.
- Las funciones miembro.
- Los grados de protección o permisos de acceso a los miembros.
- Las cadenas de caracteres o strings, objetos de la clase *String*.
- Los arrays.

Estas pocas líneas de código de la aplicación mínima, son las que usaremos como plantilla a lo largo de esta primera parte del curso.

Ejecutar la aplicación

Para correr la aplicación se selecciona en el menú **Run/Run "PrimeroApp"**, o bien se pulsa la tecla **F9**, o bien se pulsa sobre el icono en forma de rayo en la barra de herramientas.

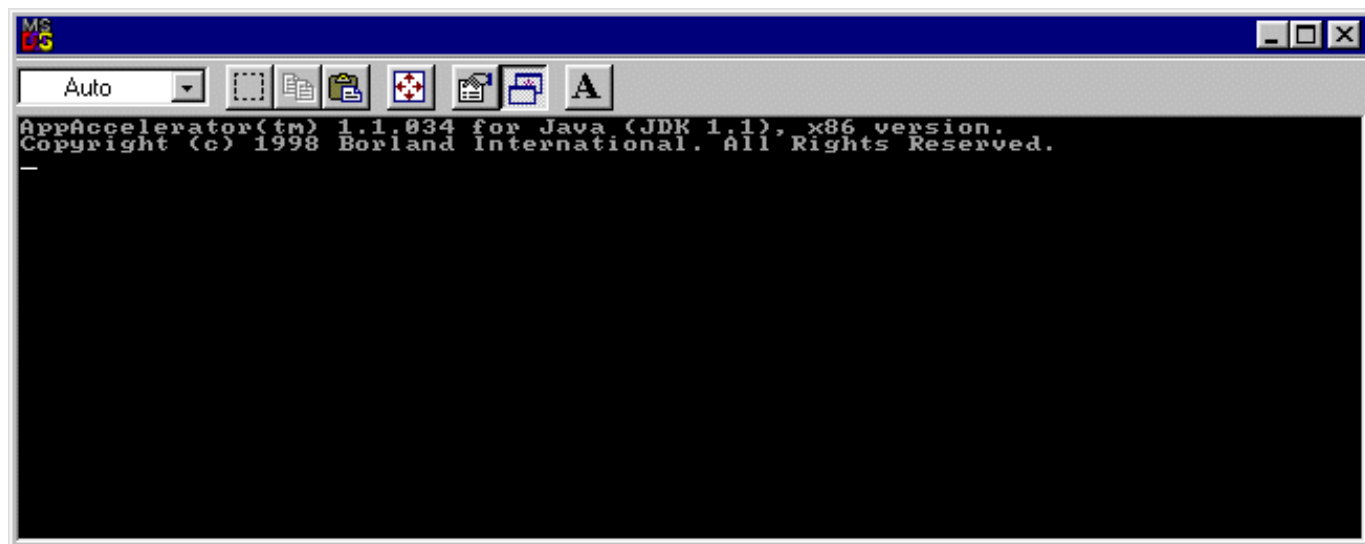
Aparece momentáneamente una ventana DOS en el escritorio de Windows 95/98 . Para que no desaparezca la ventana añadimos al final de la función miembro *main* el código que aparece seleccionado en la figura



En el capítulo dedicado al estudio de las [excepciones](#), explicaremos el significado de estas líneas de código.

La consola

Al correr la aplicación, seleccionando en el menú **Run/Run "PrimeroApp"**, o pulsando en el icono en forma de rayo, en la barra de herramientas, aparece la ventana DOS en el escritorio Windows 95/98, y se detiene la ejecución de la aplicación, esperando la pulsación de la tecla RETORNO o ENTER.

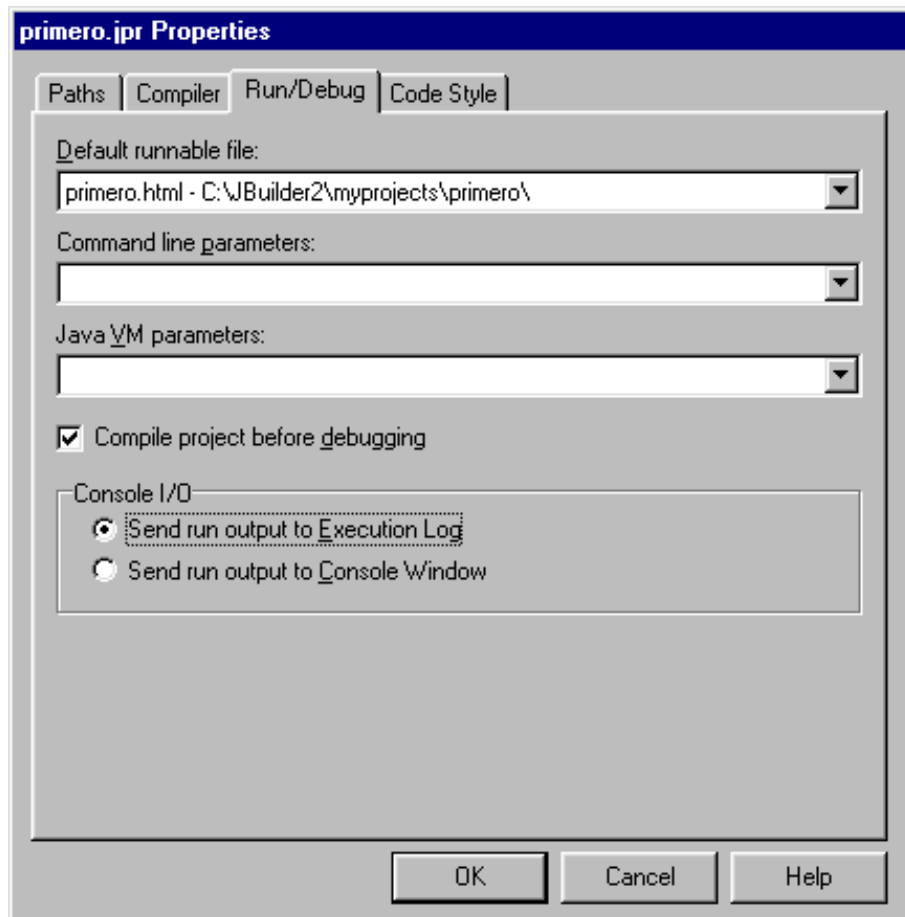


Esta es la ventana de salida estándar de una aplicación que veremos en todos los ejemplos de la primera parte de este curso.

Execution Log

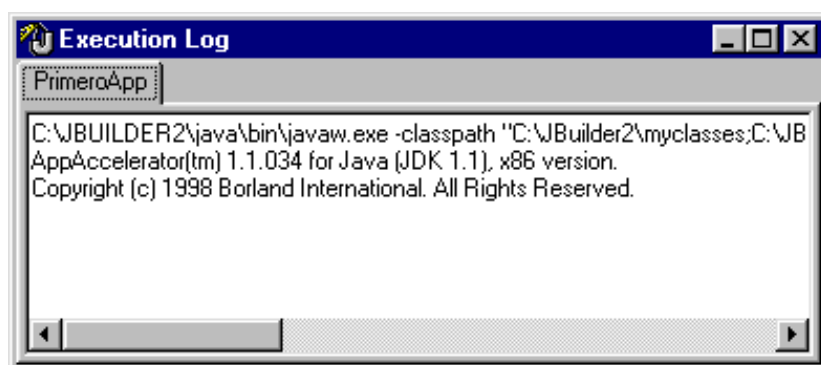
La salida del programa, o los errores que se generan durante su ejecución podemos verla en una ventana denominada **Execution Log**.

Se selecciona en el menú **Run/Parameters...** apareciendo el diálogo titulado **primero.jpr Properties**, en dicho diálogo se pulsa sobre la pestaña **Run/Debug**.



Se activa el botón de radio titulado **Send run output to Execution Log**.

Cuando se corre la aplicación ya no aparece la ventana DOS en el escritorio Windows 95/98. Ahora, hemos de seleccionar en el menú **View/Execution Log**, apareciendo una ventana titulada **Execution Log**, tal como se ve en la figura



Para que vuelva a aparecer la ventana DOS en el escritorio Windows 95/98 es necesario volver a seleccionar en el menú **Run/Parameters...** y activar el botón de radio **Send output to Console Window**, en el diálogo titulado **Properties**.

Esta aproximación es muy útil para conocer el origen de los errores que se producen en una programa.

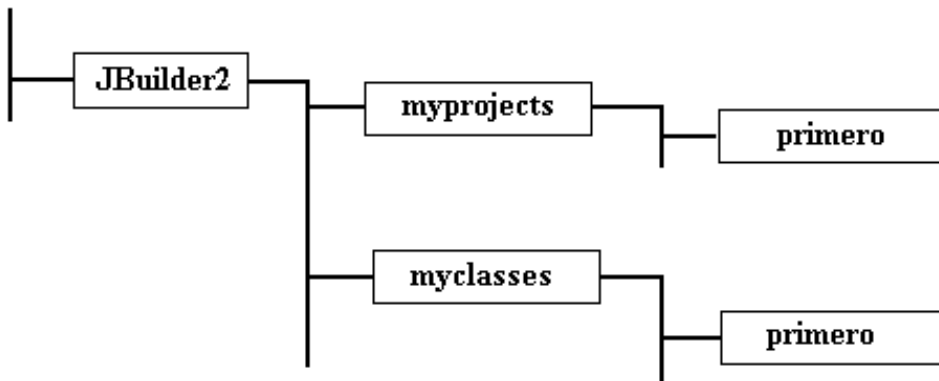
Resumen

El nombre de la carpeta (**primero**) ha de coincidir con el nombre del proyecto (**primero**), que a su vez coincide con el nombre del paquete (**package**). Los [paquetes](#) serán estudiados en otro capítulo.

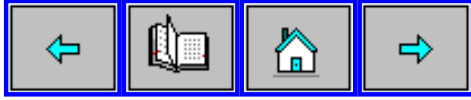
Cuando se crea una clase pública, el nombre de la clase (**PrimeroApp**) coincide con el nombre del archivo que la guarda (**PrimeroApp.java**)

Cuando se compila una aplicación se crea una carpeta (**primero**) en el subdirectorio **myclasses** cuyo nombre es el mismo que el del proyecto. El resultado de la compilación produce archivos cuya extensión es **.class**.

La estructura de los directorios se puede examinar con el programa Explorador tal como viene esquematizado en la siguiente figura



Los elementos del lenguaje Java



[Introducción](#)

[Identificadores](#)

[Comentarios](#)

[Sentencias](#)

[Bloques de código](#)

[Expresiones](#)

[Variables](#)

[Los tipos básicos de datos](#)

[Las cadenas de caracteres o strings](#)

[Palabras reservadas](#)

La sintáxis de un lenguaje define los elementos de dicho lenguaje y cómo se combinan para formar un programa. Los elementos típicos de cualquier lenguaje son los siguientes:

- Identificadores: los nombres que se dan a las variables
- Tipos de datos
- Palabras reservadas: las palabras que utiliza el propio lenguaje
- Sentencias
- Bloques de código
- Comentarios
- Expresiones
- Operadores

A lo largo de las páginas que siguen examinaremos en detalle cada uno de estos elementos.

Identificadores

Un identificador es un nombre que identifica a una variable, a un método o función miembro, a una clase. Todos los lenguajes tienen ciertas reglas para componer los identificadores:

- Todos los identificadores han de comenzar con una letra, el carácter subrayado (_) o el carácter dollar (\$).
- Puede incluir, pero no comenzar por un número
- No puede incluir el carácter espacio en blanco
- Distingue entre letras mayúsculas y minúsculas
- No se pueden utilizar las palabras reservadas como identificadores

Además de estas restricciones, hay ciertas convenciones que hacen que el programa sea más legible, pero que no afectan a la ejecución del programa. La primera y fundamental es la de encontrar un nombre que sea significativo, de modo que el programa sea lo más legible posible. El tiempo que se pretende ahorrar eligiendo nombres cortos y poco significativos se pierde con creces cuando se revisa el programa después de cierto tiempo.

| Tipo de identificador | Convención | Ejemplo |
|-----------------------|------------------------------|--------------------------------------|
| nombre de una clase | Comienza por letra mayúscula | String, Rectangulo, CinematicaApplet |
| nombre de función | comienza con letra minúscula | calcularArea, getValue, setColor |
| nombre de variable | comienza por letra minúscula | area, color, appletSize |
| nombre de constante | En letras mayúsculas | PI, MAX_ANCHO |

Comentarios

Un comentario es un texto adicional que se añade al código para explicar su funcionalidad, bien a otras personas que lean el programa, o al propio autor como recordatorio. Los comentarios son una parte importante de la documentación de un programa. Los comentarios son ignorados por el compilador, por lo que no incrementan el tamaño del archivo ejecutable; se pueden por tanto, añadir libremente al código para que pueda entenderse mejor.

La programación orientada a objetos facilita mucho la lectura del código, por lo que lo que no se precisa hacer tanto uso de los comentarios como en los lenguajes estructurados. En Java existen tres tipos de comentarios

- Comentarios en una sola línea
- Comentarios de varias líneas
- Comentarios de documentación

Como podemos observar un comentario en varias líneas es un bloque de texto situado entre el símbolo de comienzo del bloque `/*`, y otro de terminación del mismo `*/`. Teniendo en cuenta este hecho, los programadores diseñan comentarios como el siguiente:

```
/*-----|
|  (C) Angel Franco García  |
|  fecha: Marzo 1999        |
|  programa: PrimeroApp.java |
|-----*/
```

Los comentarios de documentación es un bloque de texto situado entre el símbolo de comienzo del bloque `/**`, y otro de terminación del mismo `*/`. El programa *javadoc* utiliza estos comentarios para generar la documentación del código.

```
/** Este es el primer programa de una
```

serie dedicada a explicar los fundamentos del lenguaje Java */

Habitualmente, usaremos comentarios en una sola línea `//`, ya que no tiene el inconveniente de aprendernos los símbolos de comienzo y terminación del bloque, u olvidarnos de poner este último, dando lugar a un error en el momento de la compilación. En la ventana de edición del Entorno Integrado de Desarrollo (IDE) los comentarios se distinguen del resto del código por el color del texto.

```
public class PrimeroApp{
    public static void main(String[] args) {
//imprime un mensaje
        System.out.println("El primer programa");
    }
}
```

Un procedimiento elemental de depuración de un programa consiste en anular ciertas sentencias de un programa mediante los delimitadores de comentarios. Por ejemplo, se puede modificar el programa y anular la sentencia que imprime el mensaje, poniendo delante de ella el delimitador de comentarios en una sola línea.

```
//System.out.println("El primer programa");
```

Al correr el programa, observaremos que no imprime nada en la pantalla.

La sentencia `System.out.println()` imprime un mensaje en la consola, una ventana DOS que se abre en el escritorio de Windows 95. La función *println* tiene un sólo argumento una cadena de caracteres u objeto de la [clase String](#).

Sentencias

Una sentencia es una orden que se le da al programa para realizar una tarea específica, esta puede ser: mostrar un mensaje en la pantalla, declarar una variable (para reservar espacio en memoria), inicializarla, llamar a una función, etc. Las sentencias acaban con `;`. este carácter separa una sentencia de la siguiente. Normalmente, las sentencias se ponen unas debajo de otras, aunque sentencias cortas pueden colocarse en una misma línea. He aquí algunos ejemplos de sentencias

```
int i=1;
import java.awt.*;
System.out.println("El primer programa");
rect.mover(10, 20);
```

En el lenguaje Java, los caracteres espacio en blanco se pueden emplear libremente. Como podremos ver en los sucesivos ejemplos, es muy importante para la legibilidad de un programa la colocación de unas líneas debajo de otras empleando tabuladores. El editor del IDE nos ayudará plenamente en esta tarea sin apenas percibirlo.

Bloques de código

Un bloque de código es un grupo de sentencias que se comportan como una unidad. Un bloque de código está limitado por las llaves de apertura `{` y cierre `}`. Como ejemplos de bloques de código tenemos la definición de una clase, la definición de una

función miembro, una sentencia iterativa **for**, los bloques **try ... catch**, para el tratamiento de las excepciones, etc.

Expresiones

Una expresión es todo aquello que se puede poner a la derecha del operador asignación =. Por ejemplo:

```
x=123;
y=(x+100)/4;
area=circulo.calcularArea(2.5);
Rectangulo r=new Rectangulo(10, 10, 200, 300);
```

La primera expresión asigna un valor a la variable *x*.

La segunda, realiza una operación

La tercera, es una llamada a una función miembro *calcularArea* desde un objeto *circulo* de una clase determinada

La cuarta, reserva espacio en memoria para un objeto de la clase *Rectangulo* mediante la llamada a una función especial denominada constructor.

Variables

Una variable es un nombre que se asocia con una porción de la memoria del ordenador, en la que se guarda el valor asignado a dicha variable. Hay varios tipos de variables que requieren distintas cantidades de memoria para guardar datos.

Todas las variables han de declararse antes de usarlas, la declaración consiste en una sentencia en la que figura el tipo de dato y el nombre que asignamos a la variable. Una vez declarada se le podrá asignar valores.

Java tiene tres tipos de variables:

- de instancia
- de clase
- locales

Las variables de instancia o miembros dato como veremos más adelante, se usan para guardar los atributos de un objeto particular.

Las variables de clase o miembros dato estáticos son similares a las variables de instancia, con la excepción de que los valores que guardan son los mismos para todos los objetos de una determinada clase. En el siguiente ejemplo, *PI* es una variable de clase y *radio* es una variable de instancia. *PI* guarda el mismo valor para todos los objetos de la clase *Circulo*, pero el radio de cada círculo puede ser diferente

```
class Circulo{
    static final double PI=3.1416;
```

```

        double radio;
//...
}

```

Las variables locales se utilizan dentro de las funciones miembro o métodos. En el siguiente ejemplo *area* es una variable local a la función *calcularArea* en la que se guarda el valor del área de un objeto de la clase *Circulo*. Una variable local existe desde el momento de su definición hasta el final del bloque en el que se encuentra.

```

class Circulo{
//...
    double calcularArea(){
        double area=PI*radio*radio;
        return area;
    }
}

```

En el lenguaje Java, las variables locales se declaran en el momento en el que son necesarias. Es una buena costumbre inicializar las variables en el momento en el que son declaradas. Veamos algunos ejemplos de declaración de algunas variables

```

int x=0;
String nombre="Angel ";
double a=3.5, b=0.0, c=-2.4;
boolean bNuevo=true;
int[] datos;

```

Delante del nombre de cada variable se ha de especificar el tipo de variable que hemos destacado en letra negrita. Las variables pueden ser

- Un tipo de dato primitivo
- El nombre de una clase
- Un array

El lenguaje Java utiliza el conjunto de caracteres Unicode, que incluye no solamente el conjunto ASCII sino también caracteres específicos de la mayoría de los alfabetos. Así, podemos declarar una variable que contenga la letra ñ

```

int año=1999;

```

Se ha de poner nombres significativos a las variables, generalmente formados por varias palabras combinadas, la primera empieza por minúscula, pero las que le siguen llevan la letra inicial en mayúsculas. Se debe evitar en todos los casos nombres de variables cortos como *xx*, *i*, etc.

```

double radioCirculo=3.2;

```

Las variables son uno de los elementos básicos de un programa, y se deben

- Declarar
- Inicializar
- Usar

Tipos de datos primitivos

| Tipo | Descripcion |
|----------------|---|
| boolean | Tiene dos valores true o false . |
| char | Caracteres Unicode de 16 bits Los caracteres alfa-numéricos son los mismos que los ASCII con el bit alto puesto a 0. El intervalo de valores va desde 0 hasta 65535 (valores de 16-bits sin signo). |
| byte | Tamaño 8 bits. El intervalo de valores va desde -2^7 hasta $2^7 - 1$ (-128 a 127) |
| short | Tamaño 16 bits. El intervalo de valores va desde -2^{15} hasta $2^{15} - 1$ (-32768 a 32767) |
| int | Tamaño 32 bits. El intervalo de valores va desde -2^{31} hasta $2^{31} - 1$ (-2147483648 a 2147483647) |
| long | Tamaño 64 bits. El intervalo de valores va desde -2^{63} hasta $2^{63} - 1$ (-9223372036854775808 a 9223372036854775807) |
| float | Tamaño 32 bits. Números en coma flotante de simple precisión. Estándar IEEE 754-1985 (de 1.40239846e-45f a 3.40282347e+38f) |
| double | Tamaño 64 bits. Números en coma flotante de doble precisión. Estándar IEEE 754-1985. (de 4.94065645841246544e-324d a 1.7976931348623157e+308d.) |

Los tipos básicos que utilizaremos en la mayor parte de los programas serán **boolean**, **int** y **double**.

Caracteres

En Java los caracteres no están restringidos a los ASCII sino son Unicode. Un carácter está siempre rodeado de comillas simples como 'A', '9', 'ñ', etc. El tipo de dato **char** sirve para guardar estos caracteres.

Un tipo especial de carácter es la secuencia de escape, similares a las del lenguaje C/C++, que se utilizan para representar caracteres de control o caracteres que no se imprimen. Una secuencia de escape está formada por la barra invertida (\) y un carácter. En la siguiente tabla se dan las secuencias de escape más utilizadas.

| Carácter | Secuencia de escape |
|----------------------|---------------------|
| retorno de carro | \r |
| tabulador horizontal | \t |
| nueva línea | \n |
| barra invertida | \\ |

Variables booleanas

En el lenguaje C/C++ el valor 0 se toma como falso y el 1 como verdadero. En el lenguaje Java existe el tipo de dato **boolean**. Una variable booleana solamente puede guardar uno de los dos posibles valores: true (verdadero) y false (falso).

```
boolean encontrado=false;
```



```
{...}
encontrado=true;
```

Variables enteras

Una variable entera consiste en cualquier combinación de cifras precedidos por el signo más (opcional), para los positivos, o el signo menos, para los negativos. Son ejemplos de números enteros:

12, -36, 0, 4687, -3598

Como ejemplos de declaración de variable enteras tenemos:

```
int numero=1205;
int x,y;
long m=30L;
```

int es la palabra reservada para declarar una variable entera. En el primer caso, el compilador reserva una porción de 32 bits de memoria en el que guarda el número 1205. Se accede a dicha porción de memoria mediante el nombre de la variable, *numero*. En el segundo caso, las porciones de memoria cuyos nombres son *x* e *y*, guardan cualquier valor entero si la variable es local o cero si la variable es de instancia o de clase. El uso de una variable local antes de ser convenientemente inicializada puede conducir a consecuencias desastrosas. Por tanto, declarar e inicializar una variable es una práctica aconsejable.

En la tercera línea 30 es un número de tipo **int** por defecto, le ponemos el sufijo **L** en mayúsculas o minúsculas para indicar que es de tipo **long**.

Existen como vemos en la tabla varios tipos de números enteros (**byte**, **short**, **int**, **long**), y también existe una clase denominada *BigInteger* cuyos objetos pueden guardar un número entero arbitrariamente grande.

Variables en coma flotante

Las variables del tipo **float** o **double** (coma flotante) se usan para guardar números en memoria que tienen parte entera y parte decimal.

```
double PI=3.14159;
double g=9.7805, c=2.9979e8;
```

El primero es una aproximación del número real π , el segundo es la aceleración de la gravedad a nivel del mar, el tercero es la velocidad de la luz en m/s, que es la forma de escribir $2.9979 \cdot 10^8$. El carácter punto '.', separa la parte entera de la parte decimal, en vez del carácter coma ',' que usamos habitualmente en nuestro idioma.

Otras ejemplos son los siguientes

```
float a=12.5f;
float b=7f;
double c=7.0;
```

```
double d=7d;
```

En la primera línea 12.5 lleva el sufijo **f**, ya que por defecto 12.5 es **double**. En la segunda línea 7 es un entero y por tanto 7f es un número de tipo **float**. Y así el resto de los ejemplos.

Conceptualmente, hay infinitos números de valores entre dos números reales. Ya que los valores de las variables se guardan en un número prefijado de bits, algunos valores no se pueden representar de forma precisa en memoria. Por tanto, los valores de las variables en coma flotante en un ordenador solamente se aproximan a los verdaderos números reales en matemáticas. La aproximación es tanto mejor, cuanto mayor sea el tamaño de la memoria que reservamos para guardarlo. De este hecho, surgen las variables del tipo **float** y **double**. Para números de precisión arbitraria se emplea la clase *BigDecimal*.

Valores constantes

Cuando se declara una variable de tipo **final**, se ha de inicializar y cualquier intento de modificarla en el curso de la ejecución del programa da lugar a un error en tiempo de compilación.

Normalmente, las constantes de un programa se suelen poner en letras mayúsculas, para distinguirlas de las que no son constantes. He aquí ejemplos de declaración de constantes.

```
final double PI=3.141592653589793;
final int MAX_DATOS=150;
```

Las cadenas de caracteres o strings

Además de los ocho tipos de datos primitivos, las variables en Java pueden ser declaradas para guardar una instancia de una clase, como veremos en el siguiente capítulo ([Clases y objetos](#)).

Las [cadenas de caracteres o strings](#) son distintas en Java y en el lenguaje C/C++, en este último, las cadenas son arrays de caracteres terminados en el carácter '\0'. Sin embargo, en Java son objetos de la clase *String*.

```
String mensaje="El primer programa";
```

Empleando strings, el primer programa quedaría de la forma equivalente

```
public class PrimeroApp{
    public static void main(String[] args) {
//imprime un mensaje
        String mensaje="El primer programa";
        System.out.println(mensaje);
    }
}
```

En una cadena se pueden insertar caracteres especiales como el carácter tabulador '\t' o el de nueva línea '\n'

```
String texto="Un string con \t un carácter tabulador y \n un salto de línea";
```

Palabras reservadas

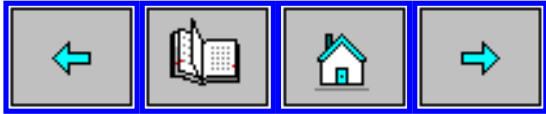
En el siguiente cuadro se listan las palabras reservadas, aquellas que emplea el lenguaje Java, y que el programador no puede utilizar como [identificadores](#). Algunas de estas palabras le resultarán familiares al programador del lenguaje C/C++. Las palabras reservadas señaladas con un asterisco (*) no se utilizan.

| | | | | |
|-----------|------------|---------|--------------|------------|
| abstract | boolean | break | byte | byvalue* |
| case | cast* | catch | char | class |
| const* | continue | default | do | double |
| else | extends | false | final | finally |
| float | for | future* | generic* | goto* |
| if | implements | import | inner* | instanceof |
| int | interface | long | native | new |
| null | operator* | outer* | package | private |
| protected | public | rest* | return | short |
| static | super | switch | synchronized | this |
| throw | transient | true | try | var* |
| void | volatile | while | | |

Las palabras reservadas se pueden clasificar en las siguientes categorías:

- Tipos de datos: **boolean, float, double, int, char**
- Sentencias condicionales: **if, else, switch**
- Sentencias iterativas: **for, do, while, continue**
- Tratamiento de las excepciones: **try, catch, finally, throw**
- Estructura de datos: **class, interface, implements, extends**
- Modificadores y control de acceso: **public, private, protected, transient**
- Otras: **super, null, this**.

Los operadores (aritméticos)



[Introducción](#)

[Los operadores aritméticos](#)

[Concatenación de strings](#)

[La precedencia de operadores](#)

[La conversión automática y promoción](#)

[Los operadores unarios](#)

Todos los lenguajes de programación permiten realizar operaciones entre los tipos de datos básicos: suma, resta, producto, cociente, etc., de dos números. Otros lenguajes como el BASIC y Java permiten "sumar", concatenar cadenas de caracteres.

En la página titulada "La primera aplicación", hemos aprendido a crear un [proyecto nuevo](#), y la clase que describe la [aplicación mínima](#) que contiene la función estática *main*. Luego, le hemos añadido código para dar cierta funcionalidad a la aplicación, que ha consistido en imprimir un mensaje en la [consola](#) o mostrarlo en la ventana denominada [Execution Log](#).

Recordaremos que la imagen del disquette significa un proyecto nuevo cuyo nombre aparece en letra negrita, y cuyos componentes son archivos código fuente en el que se guardan las clases.

Los operadores aritméticos



operador: [OperadorAp.java](#)

Java tiene cinco operadores aritméticos cuyo significado se muestra en la tabla adjunta

| Operador | Nombre | Ejemplo |
|----------|------------|---------|
| + | Suma | 3+4 |
| - | Diferencia | 3-4 |
| * | Producto | 3*4 |
| / | Cociente | 20/7 |
| % | Módulo | 20%7 |

El cociente entre dos enteros da como resultado un entero. Por ejemplo, al dividir 20 entre 7 nos da como resultado 2.

El operador módulo da como resultado el resto de la división entera. Por ejemplo 20%7 da como resultado 6 que es el resto de la división entre 20 y 7.

El operador módulo también se puede emplear con números reales. Por ejemplo, el cociente entre 7.5 y 3.0 es 2.5 y el resto es cero, es decir, $7.5 = 3.0 \times 2.5 + 0$. El operador módulo, funciona de la siguiente forma $7.5 = 3.0 \times 2 + 1.5$, calcula la diferencia entre el dividendo (7.5) y el producto del divisor (3.0) por la parte entera (2) del cociente, devolviendo 1.5. Así pues, la operación $7.5 \% 3.0$ da como resultado 1.5.

El operador asignación

Nos habremos dado cuenta que el operador más importante y más frecuentemente usado es el operador asignación `=`, que hemos empleado para la inicialización de las variables. Así,

```
int numero;
numero=20;
```

la primera sentencia declara una variable entera de tipo **int** y le da un nombre (*numero*). La segunda sentencia usa el operador asignación para inicializar la variable con el número 20.

Consideremos ahora, la siguiente sentencia.

```
a=b;
```

que asigna a *a* el valor de *b*. A la izquierda siempre tendremos una variable tal como *a*, que recibe valores, a la derecha otra variable *b*, o expresión que tiene un valor. Por tanto, tienen sentido las expresiones

```
a=1234;
double area=calculaArea(radio);
superficie=ancho*alto;
```

Sin embargo, no tienen sentido las expresiones

```
1234=a ;
calculaArea(radio)=area ;
```

Las asignaciones múltiples son también posibles. Por ejemplo, es válida la sentencia

```
c=a=b ;           //equivalente a c=(a=b) ;
```

la cual puede ser empleada para inicializar en la misma línea varias variables

```
c=a=b=321 ;           //asigna 321 a a, b y c
```

El operador asignación se puede combinar con los operadores aritméticos

| Expresión | Significado |
|-----------|-------------|
| $x+=y$ | $x=x+y$ |
| $x-=y$ | $x=x-y$ |
| $x*=y$ | $x=x*y$ |
| $x/=y$ | $x=x/y$ |

Así, la sentencia

```
x=x+23 ;
```

evalúa la expresión $x+23$, que es asignada de nuevo a x . El compilador lee primero el contenido de la porción de memoria nombrada x , realiza la suma, y guarda el resultado en la misma porción de memoria. Se puede escribir la sentencia anterior de una forma equivalente más simple

```
x+=23 ;
```

Concatenación de strings

En Java se usa el operador `+` para concatenar cadenas de caracteres o strings. Veremos en el siguiente apartado una sentencia como la siguiente:

```
System.out.println("la temperatura centígrada es "+tC) ;
```

El operador + cuando se utiliza con strings y otros objetos, crea un solo string que contiene la concatenación de todos sus operandos. Si alguno de los operandos no es una cadena, se convierte automáticamente en una cadena. Por ejemplo, en la sentencia anterior el número del tipo **double** que guarda la variable *tC* se convierte en un string que se añade al string "la temperatura centígrada es ".

Como veremos más adelante, un objeto se convierte automáticamente en un string si su clase redefine la [función miembro *toString*](#) de la clase base *Object*.

Como vemos en el listado, para mostrar un resultado de una operación, por ejemplo, la suma de dos números enteros, escribimos

```
iSuma=ia+ib;
System.out.println("El resultado de la suma es "+iSuma);
```

Concatena una cadena de caracteres con un tipo básico de dato, que convierte automáticamente en un string.

El operador += también funciona con cadenas.

```
String nombre="Juan ";
nombre+="García";
System.out.println(nombre);
```

```
public class OperadorAp {
    public static void main(String[] args) {
        System.out.println("Operaciones con enteros");
        int ia=7, ib=3;
        int iSuma, iResto;
        iSuma=ia+ib;
        System.out.println("El resultado de la suma es "+iSuma);
        int iProducto=ia*ib;
        System.out.println("El resultado del producto es "+iProducto);
        System.out.println("El resultado del cociente es "+(ia/ib));
        iResto=ia%ib;
        System.out.println("El resto de la división entera es "+iResto);

        System.out.println("*****");
        System.out.println("Operaciones con números decimales");
        double da=7.5, db=3.0;
        double dSuma=da+db;
        System.out.println("El resultado de la suma es "+dSuma);
        double dProducto=da*db;
```

```

        System.out.println("El resultado del producto es "+dProducto);
        double dCociente=da/db;
        System.out.println("El resultado del cociente es "+dCociente);
        double dResto=da%db;
        System.out.println("El resto de la división es "+dResto);
    }
}

```

La precedencia de operadores



precede: [PrecedeApp.java](#)

El lector conocerá que los operadores aritméticos tienen distinta precedencia, así la expresión

$$a+b*c$$

es equivalente a

$$a+(b*c)$$

ya que el producto y el cociente tienen mayor precedencia que la suma o la resta. Por tanto, en la segunda expresión el paréntesis no es necesario. Sin embargo, si queremos que se efectúe antes la suma que la multiplicación tenemos de emplear los paréntesis

$$(a+b)*c$$

Para realizar la operación $\frac{a}{bc}$ escribiremos

$$a/(b*c);$$

o bien,

$$a/b/c;$$

En la mayoría de los casos, la precedencia de las operaciones es evidente, sin embargo, en otros que no lo son tanto, se aconseja emplear paréntesis. Como ejemplo, estudiemos un programa que nos permite convertir una temperatura en grados Fahrenheit en su equivalente en la escala Celsius. La fórmula de conversión es

$$tC = \frac{(tF - 32) * 5}{9}$$

cuya codificación es

```
tC=(tF-32)*5/9;
```

Las operaciones se realizan como suponemos, ya que si primero se realizase el cociente 5/9, el resultado de la división entera sería cero, y el producto por el resultado de evaluar el paréntesis sería también cero. Si tenemos dudas sobre la precedencia de operadores podemos escribir

```
tC=((tF-32)*5)/9;
```

```
public class PrecedeApp {
    public static void main(String[] args) {
        int tF=80;
        System.out.println("la temperatura Fahrenheit es "+tF);
        int tC=(tF-32)*5/9;
        System.out.println("la temperatura centígrada es "+tC);
    }
}
```

La conversión automática y promoción (casting)

Cuando se realiza una operación, si un operando es entero (**int**) y el otro es de coma flotante (**double**) el resultado es en coma flotante (**double**).

```
int a=5;
double b=3.2;
double suma=a+b;
```

Cuando se declaran dos variables una de tipo **int** y otra de tipo **double**.

```
int entero;
double real=3.20567;
```

¿qué ocurrirá cuando asignamos a la variable *entero* el número guardado en la variable *real*?. Como hemos

visto se trata de dos tipos de variables distintos cuyo tamaño en memoria es de 32 y 64 bits respectivamente. Por tanto, la sentencia

```
entero=real;
```

convierte el número real en un número entero eliminando los decimales. La variable *entero* guardará el número 3.

Se ha de tener cuidado, ya que la conversión de un tipo de dato en otro es una fuente frecuente de error entre los programadores principiantes. Ahora bien, supongamos que deseamos calcular la división $7/3$, como hemos visto, el resultado de la división entera es 2, aún en el caso de que tratemos de guardar el resultado en una variable del tipo **double**, como lo prueba la siguiente porción de código.

```
int ia=7;
int ib=3;
double dc=ia/ib;
```

Si queremos obtener una aproximación decimal del número $7/3$, hemos de promocionar el entero *ia* a un número en coma flotante, mediante un procedimiento denominado promoción o *casting*.

```
int ia=7;
int ib=3;
double dc=(double)ia/ib;
```

Como aplicación, consideremos el cálculo del valor medio de dos o más números enteros

```
int edad1=10;
int edad2=15;
double media=(double)(edad1+edad2)/2;
```

El valor medio de 10 y 15 es 12.5, sin la promoción se obtendría el valor erróneo 12.

Imaginemos ahora, una función que devuelve un entero **int** y queremos guardarlo en una variable de tipo **float**. Escribiremos

```
float resultado=(float)retornaInt();
```

Existen también conversiones implícitas realizadas por el compilador, por ejemplo cuando pasamos un entero **int** a una función cuyo único parámetro es de tipo **long**.

Los operadores unarios



unario: [UnarioApp.java](#)

Los operadores unarios son:

- ++ Incremento
- -- Decremento

actúan sobre un único operando. Se trata de uno de los aspecto más confusos para el programador, ya que el resultado de la operación depende de que el operador esté a la derecha $i++$ o a la izquierda $++i$.

Conoceremos, primero el significado de estos dos operadores a partir de las sentencias equivalentes:

```
i=i+1;           //añadir 1 a i
i++;
```

Del mismo modo, lo son

```
i=i-1;           //restar 1 a i
i--;
```

Examinemos ahora, la posición del operador respecto del operando. Consideremos en primer lugar que el operador unario ++ está a la derecha del operando. La sentencia

```
j=i++;
```

asigna a j , el valor que tenía i . Por ejemplo, si i valía 3, después de ejecutar la sentencia, j toma el valor de 3 e i el valor de 4. Lo que es equivalente a las dos sentencias

```
j=i;
i++;
```

Un resultado distinto se obtiene si el operador ++ está a la izquierda del operando

```
j=++i;
```

asigna a j el valor incrementado de i . Por ejemplo, si i valía 3, después de ejecutar la sentencia j e i toman el valor de 4. Lo que es equivalente a las dos sentencias

```
++i;
```

```
j=i;
```

```
public class UnarioApp {
    public static void main(String[] args) {
        int i=8;
        int a, b, c;
        System.out.println("\tantes\t durante\t después");
        i=8; a=i; b=i++; c=i;
        System.out.println("i++\t"+a+'\t'+b+'\t'+c);
        i=8; a=i; b=i--; c=i;
        System.out.println("i--\t"+a+'\t'+b+'\t'+c);

        i=8; a=i; b=++i; c=i;
        System.out.println("++i\t"+a+'\t'+b+'\t'+c);
        i=8; a=i; b=--i; c=i;
        System.out.println("--i\t"+a+'\t'+b+'\t'+c);
    }
}
```

La salida del programa es, la siguiente

| | antes | durante | después |
|-----|-------|---------|---------|
| i++ | 8 | 8 | 9 |
| i-- | 8 | 8 | 7 |
| ++i | 8 | 9 | 9 |
| --i | 8 | 7 | 7 |

La primera columna (antes) muestra el valor inicial de *i*, la segunda columna (durante) muestra el valor de la expresión, y la última columna (después) muestra el valor final de *i*, después de evaluarse la expresión.

Se deberá de tener siempre el cuidado de inicializar la variable, antes de utilizar los operadores unarios con dicha variable.

Los operadores (relacionales)



[Introducción](#)

[Los operadores relacionales](#)

[Los operadores lógicos](#)

Los operadores relacionales



relacion: [RelacionApp.java](#)

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa. Por ejemplo, $8 > 4$ (ocho mayor que cuatro) es verdadera, se representa por el valor **true** del tipo básico **boolean**, en cambio, $8 < 4$ (ocho menor que cuatro) es falsa, **false**. En la primera columna de la tabla, se dan los símbolos de los operadores relacionales, en la segunda, el nombre de dichos operadores, y a continuación su significado mediante un ejemplo.

| Operador | nombre | ejemplo | significado |
|----------|---------------------|------------|--------------------------------|
| < | menor que | $a < b$ | a es menor que b |
| > | mayor que | $a > b$ | a es mayor que b |
| == | igual a | $a == b$ | a es igual a b |
| != | no igual a | $a != b$ | a no es igual a b |
| <= | menor que o igual a | $a \leq 5$ | a es menor que o igual a b |
| >= | mayor que o igual a | $a \geq b$ | a es menor que o igual a b |

Se debe tener especial cuidado en no confundir el operador asignación con el operador relacional igual a. Las asignaciones se realizan con el símbolo `=`, las comparaciones con `==`.

En el programa *RelacionApp*, se compara la variable *i* que guarda un 8, con un conjunto de valores, el resultado de la comparación es verdadero (**true**), o falso (**false**).

```
public class RelacionApp {
    public static void main(String[] args) {
        int x=8;
        int y=5;
        boolean compara=(x<y);
        System.out.println("x<y es "+compara);
        compara=(x>y);
        System.out.println("x>y es "+compara);
        compara=(x==y);
        System.out.println("x==y es "+compara);
        compara=(x!=y);
        System.out.println("x!=y es "+compara);
        compara=(x<=y);
        System.out.println("x<=y es "+compara);
        compara=(x>=y);
        System.out.println("x>=y es "+compara);
    }
}
```

Los operadores lógicos

Los operadores lógicos son:

- && AND (el resultado es verdadero si ambas expresiones son verdaderas)
- || OR (el resultado es verdadero si alguna expresión es verdadera)
- ! NOT (el resultado invierte la condición de la expresión)

AND y OR trabajan con dos operandos y retornan un valor lógico basadas en las denominadas tablas de verdad. El operador NOT actúa sobre un operando. Estas tablas de verdad son conocidas y usadas en el contexto de la vida diaria, por ejemplo: "si hace sol Y tengo tiempo, iré a la playa", "si NO hace sol, me quedaré en casa", "si llueve O hace viento, iré al cine". Las tablas de verdad de los operadores AND, OR y NOT se muestran en las tablas siguientes

El operador lógico AND

| x | y | resultado |
|----------|----------|------------------|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

El operador lógico OR

| x | y | resultado |
|----------|----------|------------------|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

El operador lógico NOT

| x | resultado |
|----------|------------------|
| true | false |
| false | true |

Los operadores AND y OR combinan expresiones relacionales cuyo resultado viene dado por la última columna de sus tablas de verdad. Por ejemplo:

$$(a < b) \ \&\& \ (b < c)$$

es verdadero (**true**), si ambas son verdaderas. Si alguna o ambas son falsas el resultado es falso (**false**). En cambio, la expresión

$$(a < b) \ || \ (b < c)$$

es verdadera si una de las dos comparaciones lo es. Si ambas, son falsas, el resultado es falso.

La expresión " NO a es menor que b "

$$!(a < b)$$

es falsa si $(a < b)$ es verdadero, y es verdadera si la comparación es falsa. Por tanto, el operador NOT actuando sobre $(a < b)$ es equivalente a

$$(a \geq b)$$

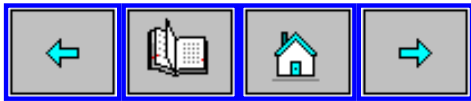
La expresión "NO a es igual a b "

$$!(a == b)$$

es verdadera si a es distinto de b , y es falsa si a es igual a b . Esta expresión es equivalente a

$$(a != b)$$

El flujo de un programa (sentencias condicionales)



[Introducción](#)

[La sentencia *if*](#)

[La sentencia *if...else*](#)

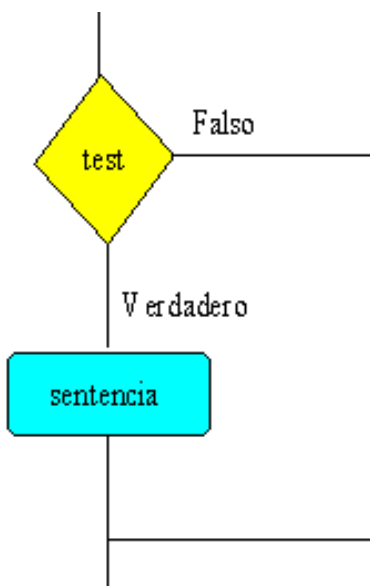
[La sentencia *switch*](#)

Del mismo modo que en la vida diaria, en un programa es necesario tomar decisiones basadas en ciertos hechos y actuar en consecuencia. El lenguaje Java tiene una sentencia básica denominada **if** (si condicional) que realiza un test y permite responder de acuerdo al resultado.

La sentencia *if*

La sentencia **if**, actúa como cabría esperar. Si la condición es verdadera, la sentencia se ejecuta, de otro modo, se salta dicha sentencia, continuando la ejecución del programa con otras sentencias a continuación de ésta. La forma general de la sentencia **if** es:

```
if (condición)
    sentencia;
```



Si el resultado del test es verdadero (**true**) se ejecuta la sentencia que sigue a continuación de **if**, en caso contrario, falso (**false**),

se salta dicha sentencia, tal como se indica en la figura. La sentencia puede consistir a su vez, en un conjunto de sentencias agrupadas en un bloque.

```
if (condición){
    sentencia1;
    sentencia2;
}
```

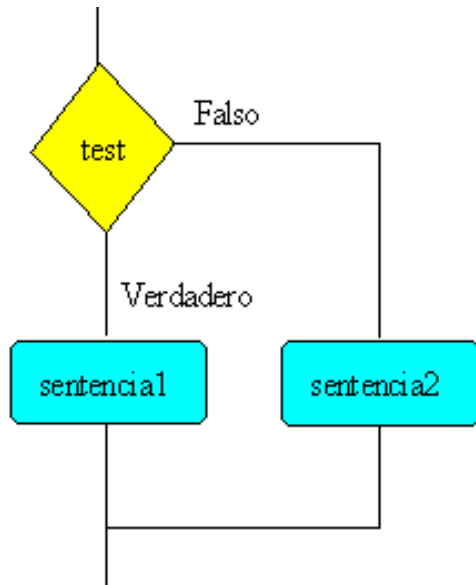
En el siguiente ejemplo, si el número del boleto que hemos adquirido coincide con el número aparecido en el sorteo, nos dicen que hemos obtenido un premio.

```
if(numeroBoleto==numeroSorteo)
    System.out.println("has obtenido un premio");
```

La sentencia *if...else*

La sentencia **if...else** completa la sentencia **if**, para realizar una acción alternativa

```
if (condición)
    sentencia1;
else
    sentencia2
```



Las dos primeras líneas indican que si la condición es verdadera se ejecuta la sentencia 1. La palabra clave **else**, significa que si la condición no es verdadera se ejecuta la sentencia 2, tal como se ve en la figura..

Dado que las sentencias pueden ser simples o compuestas la forma general de **if...else** es

```
if (condición){
    sentencia1;
    sentencia2;
```

```
}else{  
    sentencia3  
    sentencia4;  
    sentencia5;  
}
```

Existe una forma abreviada de escribir una sentencia condicional **if...else** como la siguiente:

```
if (numeroBoleto==numeroSoreteo)  
    premio=1000;  
else  
    premio=0;
```

en una sola línea

```
premio=(numeroBoleto==numeroSoreteo) ? 1000 : 0;
```

Un ejemplo significativo es el siguiente: el signo de un número elevado a una potencia par es positivo, y es negativo cuando está elevado a una potencia impar.

```
int signo=(exponente%2==0)?1:-1;
```

La condición entre paréntesis es la siguiente: un número es par, cuando el resto de la división entera de dicho número entre dos vale cero.

La sentencia *switch*

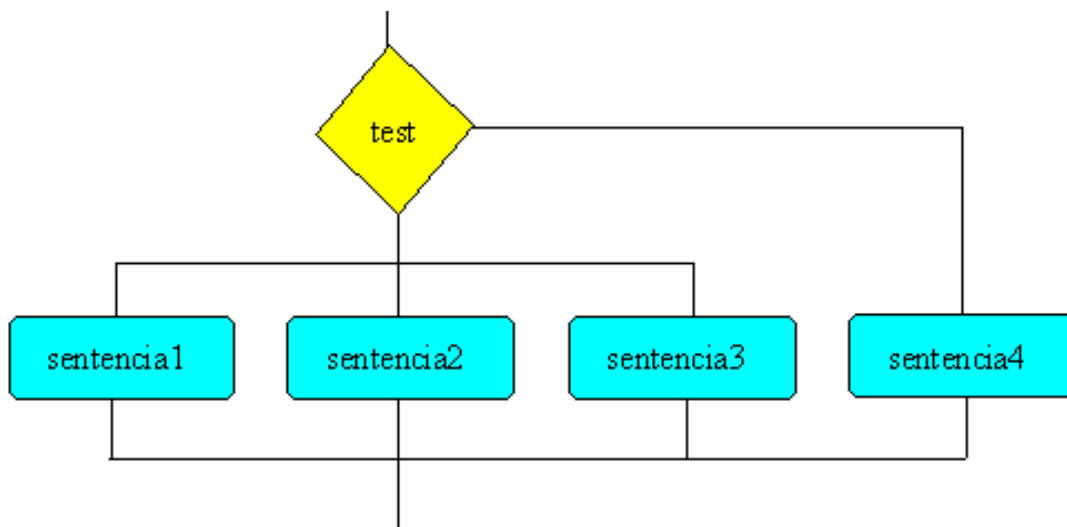


switch1: [SwitchApp1.java](#)



switch2: [SwitchApp2.java](#)

Como podemos ver en la figura del apartado anterior, la sentencia **if...else** tiene dos ramas, el programa va por una u otra rama dependiendo del valor verdadero o falso de la expresión evaluada. A veces, es necesario, elegir entre varias alternativas, como se muestra en la siguiente figura



Por ejemplo, considérese las siguientes series de sentencias **if...else**

```

if(expresion==valor1)
    sentencia1;
else if(expresion==valor2)
    sentencia2;
else if(expresion==valor3)
    sentencia3;
else
    sentencia4;
  
```

El código resultante puede ser difícil de seguir y confuso incluso para el programador avanzado. El lenguaje Java proporciona una solución elegante a este problema mediante la sentencia condicional **switch** para agrupar a un conjunto de sentencias **if...else**.

```

switch(expresion){
    case valor1:
        sentencia1;
        break;                    //sale de switch
    case valor2:
        sentencia2;
        break;                    //sale switch
    case valor3:
        sentencia3;
        break;                    //sale de switch
    default:
        sentencia4;
}
  
```

En la sentencia **switch**, se compara el valor de una variable o el resultado de evaluar una expresión, con un conjunto de números enteros *valor1*, *valor2*, *valor3*, .. o con un conjunto de caracteres, cuando coinciden se ejecuta el bloque de sentencias que están asociadas con dicho número o carácter constante. Dicho bloque de sentencias no está entre llaves sino que empieza en la palabra reservada **case** y termina en su asociado **break**. Si el compilador no encuentra coincidencia, se ejecuta la sentencia **default**, si es que está presente en el código.

Veamos ahora un ejemplo sencillo: dado el número que identifica al mes (del 1 al 12) imprimir el nombre del mes.

```

public class SwitchApp1 {
    public static void main(String[] args) {
        int mes=3;
        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("Este mes no existe"); break;
        }
    }
}

```

Ahora un ejemplo más complicado, escribir un programa que calcule el número de días de un mes determinado cuando se da el año.

Anotar primero, los meses que tienen 31 días y los que tienen 30 días. El mes de Febrero (2º mes) es el más complicado ya que tiene 28 días excepto en los años que son bisiestos que tiene 29. Son bisiestos los años múltiplos de cuatro, que no sean múltiplos de 100, pero si son bisiestos los múltiplos de 400.

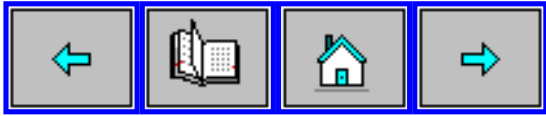
```

public class SwitchApp2 {
    public static void main(String[] args) {
        int mes=2;
        int año=1992;
        int numDias=30;
        switch (mes) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDias = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDias = 30;
                break;
            case 2:

```

```
        if ( ((año % 4 == 0) && !(año % 100 == 0)) || (año % 400 == 0) )
            numDias = 29;
        else
            numDias = 28;
        break;
    default:
        System.out.println("Este mes no existe");
        break;
    }
    System.out.println("El mes "+mes+" del año "+año+" tiene "+numDias+" días");
}
```

El flujo de un programa (sentencias iterativas)



[Introducción](#)

[La sentencia *for*](#)

[La sentencia *while*](#)

[La sentencia *do...while*](#)

[La sentencia *break*](#)

[La sentencia *continue*](#)

[Ejemplo: los números primos](#)

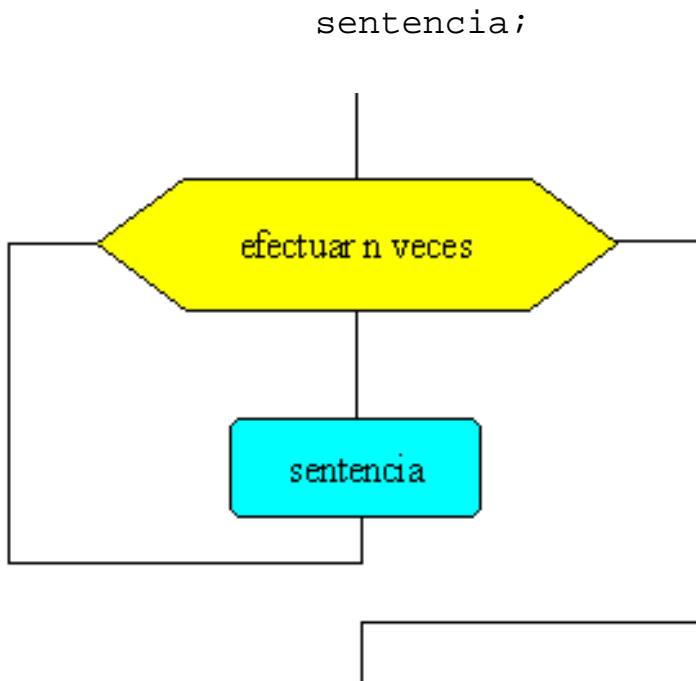
Tan importantes como las sentencias condiciones son las sentencias iterativas o repetitivas. Normalmente, las sentencias de un programa son ejecutadas en el orden en el que aparecen. Cada sentencia es ejecutada una y solamente una vez. El lenguaje Java, como la mayoría de los lenguajes, proporciona sentencias que permiten realizar una tarea una y otra vez hasta que se cumpla una determinada condición, dicha tarea viene definida por un conjunto de sentencias agrupadas en un bloque. Las sentencias iterativas son **for**, **while** y **do...while**

La sentencia *for*

 factorial: [FactorialApp.java](#)

Esta sentencia se encuentra en la mayoría de los lenguajes de programación. El bucle **for** se empleará cuando conocemos el número de veces que se ejecutará una sentencia o un bloque de sentencias, tal como se indica en la figura. La forma general que adopta la sentencia **for** es

```
for(inicialización; condición; incremento)
```



El primer término *inicialización*, se usa para inicializar una variable índice, que controla el número de veces que se ejecutará el bucle. La *condición* representa la condición que ha de ser satisfecha para que el bucle continúe su ejecución. El *incremento* representa la cantidad que se incrementa la variable índice en cada repetición.

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

El resultado será: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

La variable índice *i* se declara y se inicializa en el término *inicialización*, la *condición* se expresa de modo que *i* se debe mantener estrictamente menor que 10; la variable *i* se incrementa una unidad en cada repetición del bucle. La variable *i* es local al bucle, por lo que deja de existir una vez que se sale del bucle.

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20

```
for (int i=2; i <=20; i += 2) {
    System.out.println(i);
}
```

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20 en orden decreciente


```
for (int i=20; i >= 2; i -= 2) {  
    System.out.println(i);  
}
```

Ejemplo: Escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **for**. Guardar el resultado en un número entero de tipo **long**.

Definición: el factorial de un número n es el resultado del producto $1*2*3*...*(n-1)*n$.

```
public class FactorialApp {  
    public static void main(String[] args) {  
        int numero=4;  
        long resultado=1;  
        for(int i=1; i<=numero; i++){  
            resultado*=i;  
        }  
        System.out.println("El factorial es "+resultado);  
    }  
}
```

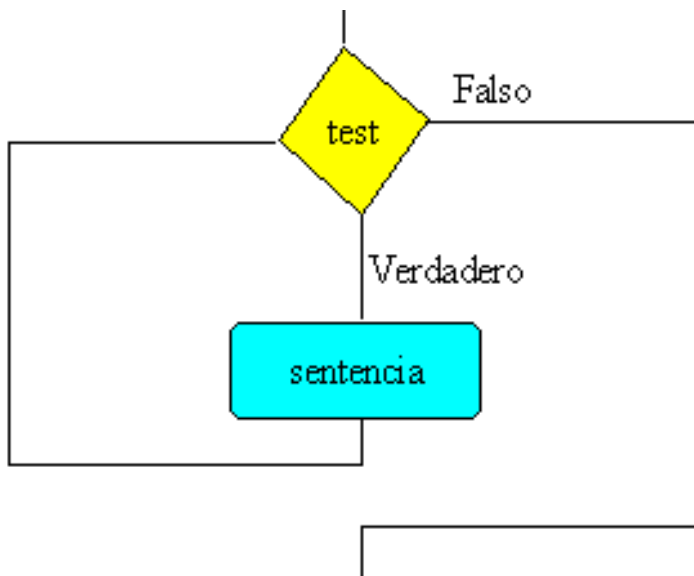
La sentencia *while*



factorial1: [FactorialApp1.java](#)

A la palabra reservada **while** le sigue una condición encerrada entre paréntesis. El bloque de sentencias que le siguen se ejecuta siempre que la condición sea verdadera tal como se ve en la figura. La forma general que adopta la sentencia **while** es:

```
while (condición)  
    sentencia;
```



Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa *while*.

```

int i=0;

while (i<10) {
    System.out.println(i);
    i++;
}

```

El valor inicial de *i* es cero, se comprueba la condición ($i < 10$), la cual resulta verdadera. Dentro del bucle, se imprime *i*, y se incrementa la variable contador *i*, en una unidad. Cuando *i* vale 10, la condición ($i < 10$) resulta falsa y el bucle ya no se ejecuta. Si el valor inicial de *i* fuese 10, no se ejecutaría el bucle. Por tanto, el bucle **while** no se ejecuta si la condición es falsa.

Ejemplo: escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **while**

```

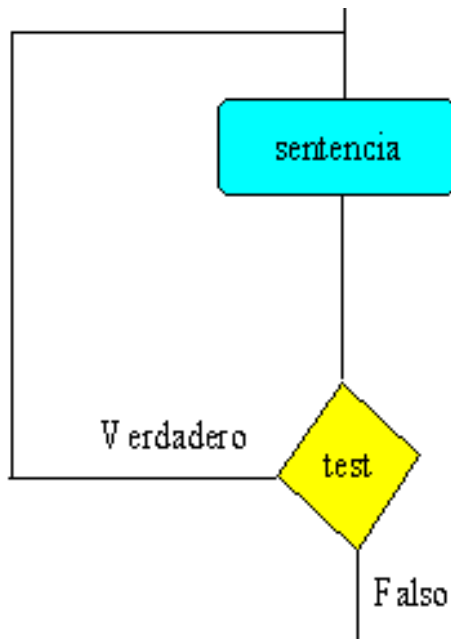
public class FactorialApp1 {
    public static void main(String[] args) {
        int numero=4;
        long resultado=1;
        while(numero>0){
            resultado*=numero;
            numero--;
        }
        System.out.println("El factorial es "+resultado);
    }
}

```

La sentencia *do...while*

Como hemos podido apreciar las sentencias **for** y **while** la condición está al principio del bucle, sin embargo, **do...while** la condición está al final del bucle, por lo que el bucle se ejecuta por lo menos una vez tal como se ve en la figura. **do** marca el comienzo del bucle y **while** el final del mismo. La forma general es:

```
do{
    sentencia;
}while(condición);
```



Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa *do..while*.

```
int i=0;

do{
    System.out.println(i);
    i++;
}while(i < 10);
```

El bucle **do...while**, se usa menos que el bucle **while**, ya que habitualmente evaluamos la expresión que controla el bucle al comienzo, no al final.

La sentencia *break*

A veces es necesario interrumpir la ejecución de un bucle **for**, **while**, o **do...while**.

```
for(int i = 0; i < 10; i++){
    if (i == 8)        break;
    System.out.println(i);
}
```

Consideremos de nuevo el ejemplo del bucle **for**, que imprime los 10 primeros números enteros, se interrumpe la ejecución del bucle cuando se cumple la condición de que la variable contador *i* valga 8. El código se leerá: "salir del bucle cuando la variable contador *i*, sea igual a 8".

Como podemos apreciar, la ejecución del bucle finaliza prematuramente. Quizás el lector pueda pensar que esto no es de gran utilidad pues, el código anterior es equivalente a

```
for(int i = 0; i <=8; i++)
    System.out.println(i);
```

Sin embargo, podemos salir fuera del bucle prematuramente si se cumple alguna condición de finalización.

```
while(true){
    if (condicionFinal)        break;
    //...otras sentencias
}
```

Como podemos apreciar en esta porción de código, la expresión en el bucle **while** es siempre verdadera, por tanto, tiene que haber algún mecanismo que nos permita salir del bucle. Si la condicion de finalización es verdadera, es decir la variable *condicionFinal* del tipo **boolean** toma el valor **true**, se sale del bucle, en caso contrario se continua el procesamiento de los datos.

La sentencia *continue*

La sentencia **continue**, fuerza al bucle a comenzar la siguiente iteración desde el principio. En la siguiente porción de código, se imprimen todos los números del 0 al 9 excepto el número 8.

```
for(int i = 0; i < 10; i++){
    if (i == 8)        continue;
    System.out.println(i);
}
```

Etiquetas

Tanto **break** como **continue** pueden tener una etiqueta opcional que indica a Java hacia donde dirigirse cuando se cumple una determinada condición.

salida:

```
for(int i=0; i<20; i++){
    while(j<70){
        if(i*j==500)    break salida;
        //...
    }
    //...
}
```

La etiqueta en este ejemplo se denomina *salida*, y se añade antes de la parte inicial del ciclo. La etiqueta debe terminar con el carácter dos puntos **:**. Si no disponemos de etiqueta, al cumplirse la condición $i*j==500$, se saldría del bucle interno **while**, pero el proceso de cálculo continuaría en el bucle externo **for**.

Ejemplo: los números primos



primos: [PrimosApp.java](#)

Escribir un programa que calcule los números primos comprendidos entre 3 y 100.

Los números primos tienen la siguiente característica: un número primo es solamente divisible por sí mismo y por la unidad, por tanto, un número primo no puede ser par excepto el 2. Para saber si un número impar es primo, dividimos dicho número por todos los números impares comprendidos entre 3 y la mitad de dicho número. Por ejemplo, para saber si 13 es un número primo basta dividirlo por 3, y 5. Para saber si 25 es número primo se divide entre 3, 5, 7, 9, y 11. Si el resto de la división (operación módulo **%**) es cero, el número no es primo.

```

public class PrimosApp {
    public static void main(String[] args) {
        boolean bPrimo;
        System.out.println("Números primos comprendidos entre 3 y 100");
        for(int numero=3; numero<100; numero+=2){
            bPrimo=true;
            for(int i=3; i<numero/2; i+=2){
                if(numero%i==0){
                    bPrimo=false;
                    break;
                }
            }
            if(bPrimo){
                System.out.print(numero+" - ");
            }
        }
    }
}

```

En primer lugar, hacemos un bucle **for** para examinar los números impares comprendidos entre 3 y 100.

Hacemos la suposición de que *numero* es primo, es decir, de que la variable de control *bPrimo* toma el valor **true**. Para confirmarlo, se halla el resto de la división entera entre *numero*, y los números *i* impares comprendidos entre 3 y *numero/2*. (Se recordará que todo número es divisible por la unidad). Si el número *numero* es divisible por algún número *i* (el resto de la división entera *numero%i* es cero), entonces el número *numero* no es primo, se abandona el bucle (**break**) con la variable de control *bPrimo* tomando el valor **false**. En el caso de que *numero* sea un número primo, se completa el bucle interno, tomando la variable de control *bPrimo* el valor inicial **true**.

Por último, si el número es primo, *bPrimo* es **true**, se imprime en la ventana, uno a continuación del otro separados por un guión.

En este programa podemos observar la diferencia entre *print* y *println*. El sufijo *ln* en la segunda función indica que se imprime el argumento y a continuación se pasa a la línea siguiente.

Conceptos básicos de la Programación Orientada a Objetos



[Clases y objetos](#)

[Introducción](#)

[El proyecto](#)

[La clase](#)

[Los objetos](#)

[La vida de un objeto](#)

[Identificadores](#)

Introducción

Cuando se escribe un programa en un lenguaje orientado a objetos, definimos una plantilla o clase que describe las características y el comportamiento de un conjunto de objetos similares. La clase automóvil describe las características comunes de todos los automóviles: sus atributos y su comportamiento. Los atributos o propiedades se refieren a la marca o fabricante, el color, las dimensiones, si tienen dos, tres, cuatro o más puertas, la potencia, si utiliza como combustible la gasolina o gasoil, etc. El comportamiento se refiere a la posibilidad de desplazarse por una carretera, frenar, acelerar, cambiar de marcha, girar, etc.

Luego, tenemos automóviles concretos, por ejemplo el automóvil propio de una determinada marca, color, potencia, etc, el automóvil del vecino de otra marca, de otro color, etc, , el automóvil de un amigo, etc.

Una clase es por tanto una plantilla implementada en software que describe un conjunto de objetos con atributos y comportamiento similares.

Una instancia u objeto de una clase es una representación concreta y específica de una clase y que reside en la memoria del ordenador.

Atributos

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos se guardan en variables denominadas de instancia, y cada objeto particular puede tener valores distintos para estas variables.

Las variables de instancia también denominados miembros de dato, son declaradas en la clase pero sus valores son fijados y cambiados en el objeto.

Además de las variables de instancia hay variables de clase, las cuales se aplican a la clase y a todas sus instancias. Por ejemplo, el número de ruedas de un automóvil es el mismo cuatro, para todos los automóviles.

Comportamiento

El comportamiento de los objetos de una clase se implementa mediante funciones miembro o métodos. Un método es un conjunto de instrucciones que realizan una determinada tarea y son similares a las funciones de los lenguajes estructurados.

Del mismo modo que hay variables de instancia y de clase, también hay métodos de instancia y de clase. En el primer caso, un objeto llama a un método para realizar una determinada tarea, en el segundo, el método se llama desde la propia clase.

El proyecto

El proyecto consta de dos archivos, el primero contiene la clase *Rectangulo* que se guarda en el archivo *Rectangulo.java* y no tiene el método *main*. La última casilla del [asistente de creación de clases](#) **New Java File** debe de estar desactivada.

La otra clase, es la que describe la aplicación *RectanguloApp1* y se guarda en el archivo *RectanguloApp1.java*, esta clase tiene que tener el método *main*, por lo tanto, la última casilla del asistente de creación de clases **New Java File** debe de estar activada.

Un proyecto puede constar de varias clases (normalmente se sitúa cada clase en un archivo) pero solamente una tiene el método *main* y representa la aplicación. Para distinguir la clase que describe la aplicación de las demás le hemos añadido el sufijo **App**.



rectangulo1: [Rectangulo.java](#), [RectanguloApp1.java](#)

La clase

Para crear una clase se utiliza la palabra reservada **class** y a continuación el nombre de la clase. La definición de la clase se pone entre las llaves de apertura y cierre. El nombre de la clase empieza por letra mayúscula.

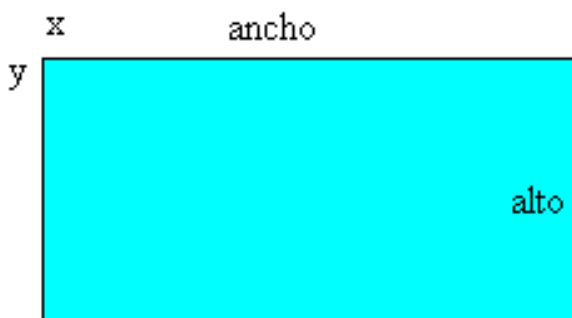
```
class Rectangulo{
    //miembros dato
    //funciones miembro
}
```

Los miembros dato

Los valores de los atributos se guardan en los miembros dato o variables de instancia. Los nombres de dichas variables comienzan por letra minúscula.

Vamos a crear una clase denominada *Rectangulo*, que describa las características comunes a estas figuras planas que son las siguientes:

- El origen del rectángulo: el origen o posición de la esquina superior izquierda del rectángulo en el plano determinado por dos números enteros x e y .
- Las dimensiones del rectángulo: *ancho* y *alto*, otros dos números enteros.



```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    //faltan las funciones miembro
}
```

Las funciones miembro

En el lenguaje C++ las funciones miembro se declaran, se definen y se llaman. En el lenguaje Java las funciones miembro o métodos solamente se definen y se llaman.

El nombre de las funciones miembro o métodos comienza por letra minúscula y deben sugerir acciones (mover, calcular, etc.). La definición de una función tiene el siguiente formato:

```
tipo nombreFuncion(tipo parm1, tipo parm2, tipo parm3){
    //...sentencias
}
```

Entre las llaves de apertura y cierre se coloca la definición de la función. *tipo* indica el tipo de dato que puede ser predefinido **int**, **double**, etc, o definido por el usuario, una clase cualquiera.

Para llamar a un función miembro o método se escribe

```
retorno=objeto.nombreFuncion(arg1, arg2, arg3);
```

Cuando se llama a la función, los argumentos *arg1*, *arg2*, *arg3* se copian en los parámetros *parm1*, *parm2*, *parm3* y se ejecutan las sentencias dentro de la función. La función finaliza cuando se llega al final de su bloque de definición o cuando encuentra una sentencia **return**.

Cuando se llama a la función, el valor devuelto mediante la sentencia **return** se asigna a la variable *retorno*.

Cuando una función no devuelve nada se dice de tipo **void**. Para llamar a la función, se escribe

```
objeto.nombreFuncion(arg1, arg2, arg3);
```

Estudiaremos más adelante con más detalle como se [definen las funciones](#).

Una función suele finalizar cuando llega al final del bloque de su definición

```
void funcion(...){
//sentencias...
}
```

Una función puede finalizar antes del llegar al final de su definición

```
void funcion(...){
//sentencias...
    if(condicion) return;
```

```
//sentencias..
}
```

Una función puede devolver un valor (un tipo de dato primitivo o un objeto).

```
double funcion(...){
    double suma=0.0;
//sentencias...
    return suma;
}
```

Cualquier variable declarada dentro de la función tiene una vida temporal, existiendo en memoria, mientras la función esté activa. Se trata de variables locales a la función. Por ejemplo:

```
void nombreFuncion(int parm){
    //...
    int i=5;
    //...
}
```

La variable *parm*, existe desde el comienzo hasta el final de la función. La variable local *i*, existe desde el punto de su declaración hasta el final del bloque de la función.

Se ha de tener en cuenta que las funciones miembro tienen acceso a los miembros dato, por tanto, es importante en el diseño de una clase decidir qué variables son miembros dato, qué variables son locales a las funciones miembro, y qué valores les pasamos a dichas funciones. Los ejemplos nos ayudarán a entender esta distinción.

Hemos definido los atributos o miembros dato de la clase *Rectangulo*, ahora le vamos añadir un comportamiento: los objetos de la clase *Rectangulo* o rectángulos sabrán calcular su área, tendrán capacidad para trasladarse a otro punto del plano, sabrán si contienen en su interior un punto determinado del plano.

La función que calcula el área realizará la siguiente tarea, calculará el producto del ancho por el alto del rectángulo y devolverá el resultado. La función devuelve un entero es por tanto, de tipo **int**. No es necesario pasarle datos ya que tiene acceso a los miembros dato *ancho* y *alto* que guardan la anchura y la altura de un rectángulo concreto.

```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    int calcularArea(){
        return (ancho*alto);
    }
}
```

A la función que desplaza el rectángulo horizontalmente en dx , y verticalmente en dy , le pasamos dichos desplazamientos, y a partir de estos datos actualizará los valores que guardan sus miembros dato x e y . La función no devuelve nada es de tipo **void**.

```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

La función que determina si un punto está o no en el interior del rectángulo, devolverá **true** si el punto se encuentra en el interior del rectángulo y devolverá **false** si no se encuentra, es decir, será una función del tipo **boolean**. La función necesitará conocer las coordenadas de dicho punto. Para que un punto de coordenadas $x1$ e $y1$ esté dentro de un rectángulo cuyo origen es x e y , y cuyas dimensiones son *ancho* y *alto*, se deberá cumplir a la vez cuatro condiciones

$x1 > x$ y a la vez $x1 < x + ancho$

También se debe cumplir

$y1 > y$ y a la vez $y1 < y + alto$

Como se tienen que cumplir las cuatro condiciones a la vez, se unen mediante el [operador lógico AND](#) simbolizado por **&&**.

```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    boolean estaDentro(int x1, int y1){
        if((x1 > x) && (x1 < x + ancho) && (y1 > y) && (y1 < y + alto)) {
            return true;
        }
        return false;
    }
}
```

En el lenguaje Java, si la primera condición es falsa no se evalúan las restantes expresiones ya que el resultado

es **false**. Ahora bien, si la primera es verdadera **true**, se pasa a evaluar la segunda, si ésta es falsa el resultado es **false**, y así sucesivamente.

Los constructores

Un objeto de una clase se crea llamando a una función especial denominada constructor de la clase. El constructor se llama de forma automática cuando se crea un objeto, para situarlo en memoria e inicializar los miembros declarados en la clase. El constructor tiene el mismo nombre que la clase. Lo específico del constructor es que no tiene tipo de retorno.

```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    Rectangulo(int x1, int y1, int w, int h){
        x=x1;
        y=y1;
        ancho=w;
        alto=h;
    }
}
```

El constructor recibe cuatro números que guardan los parámetros $x1$, $y1$, w y h , y con ellos inicializa los miembros x , y , $ancho$ y $alto$.

Una clase puede tener más de un constructor. Por ejemplo, el siguiente constructor crea un rectángulo cuyo origen está en el punto (0, 0).

```
class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    Rectangulo(int w, int h){
        x=0;
        y=0;
        ancho=w;
        alto=h;
    }
}
```

Este constructor crea un rectángulo de dimensiones nulas situado en el punto (0, 0),

```

class Rectangulo{
    int x;
    int y;
    int ancho;
    int alto;
    Rectangulo(){
        x=0;
        y=0;
        ancho=0;
        alto=0;
    }
}

```

Con estas porciones de código definimos la clase, y la guardamos en un archivo que tenga el mismo nombre que la clase *Rectangulo* y con extensión *.java*.

```

public class Rectangulo {
    int x;
    int y;
    int ancho;
    int alto;
    public Rectangulo() {
        x=0;
        y=0;
        ancho=0;
        alto=0;
    }
    public Rectangulo(int x1, int y1, int w, int h) {
        x=x1;
        y=y1;
        ancho=w;
        alto=h;
    }
    public Rectangulo(int w, int h) {
        x=0;
        y=0;
        ancho=w;
        alto=h;
    }
    int calcularArea(){
        return (ancho*alto);
    }
    void desplazar(int dx, int dy){
        x+=dx;

```

```

        y+=dy;
    }
    boolean estaDentro(int x1, int y1){
        if((x1>x)&&(x1<x+ancho)&&(y1>y)&&(y1<y+ancho)){
            return true;
        }
        return false;
    }
}

```

Los objetos

Para crear un objeto de una clase se usa la palabra reservada **new**.

Por ejemplo,

```
Rectangulo rect1=new Rectangulo(10, 20, 40, 80);
```

new reserva espacio en memoria para los miembros dato y devuelve una referencia que se guarda en la variable *rect1* del tipo *Rectangulo* que denominamos ahora objeto. Dicha sentencia, crea un objeto denominado *rect1* de la clase *Rectangulo* llamando al segundo constructor en el listado. El rectángulo estará situado en el punto de coordenadas $x=10$, $y=20$; tendrá una anchura de *ancho*=40 y una altura de *alto*=80.

```
Rectangulo rect2=new Rectangulo(40, 80);
```

Crea un objeto denominado *rect2* de la clase *Rectangulo* llamando al tercer constructor, dicho rectángulo estará situado en el punto de coordenadas $x=0$, $y=0$; y tendrá una anchura de *ancho*=40 y una altura de *alto*=80.

```
Rectangulo rect3=new Rectangulo();
```

Crea un objeto denominado *rect3* de la clase *Rectangulo* llamando al primer constructor, dicho rectángulo estará situado en el punto de coordenadas $x=0$, $y=0$; y tendrá una anchura de *ancho*=0 y una altura de *alto*=0.

Acceso a los miembros

Desde un objeto se puede acceder a los miembros mediante la siguiente sintaxis

objeto.miembro;

Por ejemplo, podemos acceder al miembro dato *ancho*, para cambiar la anchura de un objeto rectángulo.

```
rect1.ancho=100;
```

El rectángulo *rect1* que tenía inicialmente una anchura de 40, mediante esta sentencia se la cambiamos a 100.

Desde un objeto llamamos a las funciones miembro para realizar una determinada tarea. Por ejemplo, desde el rectángulo *rect1* llamamos a la función *calcularArea* para calcular el área de dicho rectángulo.

```
rect1.calcularArea();
```

La función miembro *area* devuelve un entero, que guardaremos en una variable entera *medidaArea*, para luego usar este dato.

```
int medidaArea=rect1.calcularArea();  
System.out.println("El área del rectángulo es "+medidaArea);
```

Para desplazar el rectángulo *rect2*, 10 unidades hacia la derecha y 20 hacia abajo, escribiremos

```
rect2.desplazar(10, 20);
```

Podemos verificar mediante el siguiente código si el punto (20, 30) está en el interior del rectángulo *rect1*.

```
if(rect1.estaDentro(20,30)){  
    System.out.println("El punto está dentro del rectángulo");  
}else{  
    System.out.println("El punto está fuera del rectángulo");  
}
```

rect1.dentro() devuelve **true** si el punto (20, 30) que se le pasa a dicha función miembro está en el interior del rectángulo *rect1*, ejecutándose la primera sentencia, en caso contrario se ejecuta la segunda.

Como veremos más adelante no siempre es posible acceder a los miembros, si establecemos controles de acceso a los mismos.


```

public class RectanguloApp1 {
    public static void main(String[] args) {
        Rectangulo rect1=new Rectangulo(10, 20, 40, 80);
        Rectangulo rect2=new Rectangulo(40, 80);
        Rectangulo rect3=new Rectangulo();
        int medidaArea=rect1.calcularArea();
        System.out.println("El área del rectángulo es "+medidaArea);

        rect2.desplazar(10, 20);

        if(rect1.estaDentro(20,30)){
            System.out.println("El punto está dentro del rectángulo");
        }else{
            System.out.println("El punto está fuera del rectángulo");
        }
    }
}

```

La vida de un objeto

En el lenguaje C++, los objetos que se crean con **new** se han de eliminar con **delete**. **new** reserva espacio en memoria para el objeto y **delete** libera dicha memoria. En el lenguaje Java no es necesario liberar la memoria reservada, el recolector de basura (garbage collector) se encarga de hacerlo por nosotros, liberando al programador de una de las tareas que más quebraderos de cabeza le producen, olvidarse de liberar la memoria reservada.

Veamos un ejemplo

```

public class UnaClase {
    public static void main(String[] args) {
        Image granImagen=creaImagen();
        mostrar(greImagen);

        while(condicion){
            calcular();
        }
    }
}

```

El objeto *granImagen*, continua en memoria hasta que se alcanza el final de la función *main*, aunque solamente es necesario hasta el bucle **while**. En C o en C++ eliminaríamos dicho objeto liberando la memoria que ocupa mediante **delete**. El equivalente en Java es el de asignar al objeto *granImagen* el valor **null**.

```
public class UnaClase {
    public static void main(String[] args) {
        Image granImagen=creaImagen();
        mostrar(graImagen);
        granImagen=null;

        while(condicion){
            calcular();
        }
    }
}
```

A partir de la sentencia marcada en letra negrita el recolector de basura se encargará de liberar la memoria ocupada por dicha imagen. Así pues, se asignará el valor **null** a las referencias a objetos temporales que ocupen mucha memoria tan pronto como no sean necesarios.

Creamos dos objetos de la clase rectángulo, del mismo modo que en el apartado anterior

```
Rectangulo rect1=new Rectangulo(10, 20, 40, 80);
Rectangulo rect3=new Rectangulo();
```

Si escribimos

```
rect3=rect1;
```

En *rect3* se guarda la referencia al objeto *rect1*. La referencia al objeto *rect3* se pierde. El recolector se encarga de liberar el espacio en memoria ocupado por el objeto *rect3*.

La destrucción de un objeto es una tarea (thread) de baja prioridad que lleva a cabo la Máquina Virtual Java (JVM). Por tanto, nunca podemos saber cuando se va a destruir un objeto.

Puede haber situaciones en las que es necesario realizar ciertas operaciones que no puede realizar el recolector de basura (garbage collector) cuando se destruye un objeto. Por ejemplo, se han abierto varios archivos durante la vida de un objeto, y se desea que los archivos estén cerrados cuando dicho objeto desaparece. Se puede definir en la clase un método denominado [*finalize*](#) que realice esta tarea. Este método es llamado por el recolector de basura inmeditamente antes de que el objeto sea destruído.

Identificadores

Cómo se escriben los nombres de las variables, de las clases, de las funciones, etc., es un asunto muy importante de cara a la comprensión y el mantenimiento de código. En la introducción a los fundamentos del lenguaje Java hemos tratado ya de los [identificadores](#).

El código debe de ser tanto más fácil de leer y de entender como sea posible. Alguien que lea el código, incluso después de cierto tiempo, debe ser capaz de entender lo que hace a primera vista, aunque los detalles internos, es decir, cómo lo hace, precise un estudio detallado.

Vemos primero un ejemplo que muestra un código poco legible y por tanto, muy difícil de mantener

```
public class Cuen{
    private int ba;

    Cuen(int ba){
        this.ba=ba;
    }
    public void dep(int i){
        ba+=i;
    }
    public boolean ret(int i){
        if(ba>=i){
            ba-=i;
            return true;
        }
        return false;
    }
    public int get(){
        return ba;
    }
}
```

Las abreviaciones empleadas solamente tienen significado para el programador en el momento de escribir el código, ya que puede olvidarse de su significado con el tiempo. Otros programadores del grupo tienen que descifrar el significado del nombre de cada variable o de cada función. El tiempo extra que se gasta en escribir con claridad el nombre de los diversos elementos que entran en el programa, se ahorra más adelante durante su desarrollo, depuración, y mejora, es decir, durante todo el ciclo de vida del programa.

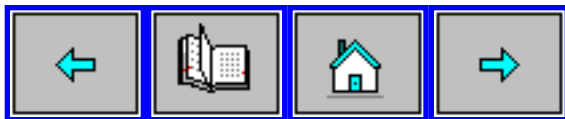
```
public class CuentaBancaria{
    private int balance;

    CuentaBancaria(int balance){
        this.balance=balance;
    }
    public void depositar(int cantidad){
        balance+=cantidad;
    }
    public boolean retirar(int cantidad){
        if(balance>=cantidad){
            balance-=cantidad;
            return true;
        }
        return false;
    }
    public int obtenerBalance(){
        return balance;
    }
}
```

Este es una programa sencillo de una cuenta bancaria. El tipo de dato puede ser entero (**int** o **long**), si la unidad monetaria tiene poco valor como la peseta, o un número decimal (**double**) si la unidad monetaria es de gran valor como el Euro y el Dólar.

El código de las funciones miembro es muy sencillo y su significado se hace evidente al leer el programa. La función *retirar* es de tipo **boolean**, ya que no (**false**) estamos autorizados a retirar una cantidad mayor que la existente en ese momento en el banco. Sin embargo, si (**true**) estamos autorizados a retirar una cantidad menor que la que tenemos en la cuenta.

Composición



Clases y objetos

[La clase *Punto*](#)

[La clase *Rectangulo*](#)

[Objetos de la clase *Rectangulo*](#)

Hay dos formas de reutilizar el código, mediante la composición y mediante la herencia. La composición significa utilizar objetos dentro de otros objetos. Por ejemplo, un applet es un objeto que contiene en su interior otros objetos como botones, etiquetas, etc. Cada uno de los controles está descrito por una clase.

Vamos a estudiar una nueva aproximación a la clase *Rectangulo* definiendo el origen, no como un par de coordenadas x e y (números enteros) sino como objetos de una nueva clase denominada *Punto*.



rectangulo: [Punto.java](#), [Rectangulo.java](#), [RectanguloApp.java](#)

La clase *Punto*

La clase *Punto* tiene dos miembros dato, la abscisa x y la ordenada y de un punto del plano. Definimos dos constructores uno por defecto que sitúa el punto en el origen, y otro constructor explícito que proporciona las coordenadas x e y de un punto concreto.

```
public class Punto {  
    int x;  
    int y;  
    //funciones miembro  
}
```

El constructor explícito de la clase *Punto* podemos escribirlo de dos formas

```
public Punto(int x1, int y1) {
    x = x1;
    y = y1;
}
```

Cuando el nombre de los parámetros es el mismo que el nombre de los miembros datos escribimos

```
public Punto(int x, int y) {
    this.x = x;
    this.y = y;
}
```

this.x que está a la izquierda y que recibe el dato *x* que se le pasa al constructor se refiere al miembro dato, mientras que *x* que está a la derecha es el parámetro. **this** es una palabra reservada que guarda una referencia al objeto propio, u objeto actual. Tendremos ocasión a lo largo del curso de encontrar esta palabra en distintas situaciones.

La función miembro *desplazar* simplemente cambia la posición del punto desde (x, y) a $(x+dx, y+dy)$. La función *desplazar* cuando es llamada recibe en sus dos parámetros *dx* y *dy* el desplazamiento del punto y actualiza las coordenadas *x* e *y* del punto. La función no retorna ningún valor

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Para crear un objeto de la clase *Punto* cuyas coordenadas *x* e *y* valgan repectivamente 10 y 23 escribimos

```
Punto p=new Punto(10, 23);
```

Para desplazar el punto *p* 10 unidades hacia la izquierda y 40 hacia abajo, llamamos desde el objeto *p* a la función *desplazar* y le pasamos el desplazamiento horizontal y vertical.

```
p.desplazar(-10, 40);
```

El código completo de la clase *Punto*, es el siguiente

```

public class Punto {
    int x = 0;
    int y = 0;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Punto() {
        x=0;
        y=0;
    }
    void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}

```

La clase *Rectangulo*

La clase *Rectangulo* tiene como miembros dato, el *origen* que es un objeto de la clase *Punto* y las dimensiones *ancho* y *alto*.

```

public class Rectangulo {
    Punto origen;
    int ancho ;
    int alto ;
    //funciones miembro
}

```

El constructor por defecto, crea un rectángulo situado en el punto 0,0 y con dimensiones nulas

```

    public Rectangulo() {
        origen = new Punto(0, 0);
        ancho=0;
        alto=0;
    }

```

El constructor explícito crea un rectángulo situado en un determinado punto *p* y con unas dimensiones

que se le pasan en el constructor

```
public Rectangulo(Punto p, int w, int h) {
    origen = p;
    ancho = w;
    alto = h;
}
```

Podemos definir otros constructores en términos del constructor explícito usando la palabra reservada **this**.

```
public Rectangulo(Punto p) {
    this(p, 0, 0);
}
public Rectangulo(int w, int h) {
    this(new Punto(0, 0), w, h);
}
```

El primero crea un rectángulo de dimensiones nulas situado en el punto p . El segundo, crea un rectángulo de unas determinadas dimensiones situándolo en el punto 0, 0. Dentro del cuerpo de cada constructor se llama al constructor explícito mediante **this** pasándole en sus parámetros los valores apropiados.

Para desplazar un rectángulo, trasladamos su origen (esquina superior izquierda) a otra posición, sin cambiar su anchura o altura. Desde el objeto *origen*, llamamos a la función *desplazar* miembro de la clase *Punto*

```
void desplazar(int dx, int dy) {
    origen.desplazar(dx, dy);
}
```

El código completo de la nueva clase *Rectangulo*, es el siguiente.


```

public class Rectangulo {
    Punto origen;
    int ancho ;
    int alto ;

    public Rectangulo() {
        origen = new Punto(0, 0);
        ancho=0;
        alto=0;
    }
    public Rectangulo(Punto p) {
        this(p, 0, 0);
    }
    public Rectangulo(int w, int h) {
        this(new Punto(0, 0), w, h);
    }
    public Rectangulo(Punto p, int w, int h) {
        origen = p;
        ancho = w;
        alto = h;
    }
    void desplazar(int dx, int dy) {
        origen.desplazar(dx, dy);
    }
    int calcularArea() {
        return ancho * alto;
    }
}

```

Objetos de la clase *Rectangulo*

Para crear un rectángulo *rect1* situado en el punto (0, 0) y cuyas dimensiones son 100 y 200 escribimos

```
Rectangulo rect1=new Rectangulo(100, 200);
```

Para crear un rectángulo *rect2*, situado en el punto de coordenadas 44, 70 y de dimensiones nulas escribimos

```
Punto p=new Punto(44, 70);
Rectangulo rect2=new Rectangulo(p);
```

O bien, en una sólo línea

```
Rectangulo rect2=new Rectangulo(new Punto(44, 70));
```

Para desplazar el rectángulo *rect1* desde el punto (100, 200) a otro punto situado 40 unidades hacia la derecha y 20 hacia abajo, sin modificar sus dimensiones, escribimos

```
rect1.desplazar(40, 20);
```

Para hallar y mostrar el área del rectángulo *rect1* podemos escribir

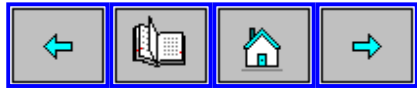
```
System.out.println("el área es "+rect1.calcularArea());
```

Para hallar el área de un rectángulo de 100 unidades de largo y 50 de alto y guardar el resultado en la variable entera *areaRect*, escribimos en una sólo línea.

```
int areaRect=new Rectangulo(100, 50).calcularArea();
```

```
public class RectanguloApp {
    public static void main(String[] args) {
        Rectangulo rect1=new Rectangulo(100, 200);
        Rectangulo rect2=new Rectangulo(new Punto(44, 70));
        Rectangulo rect3=new Rectangulo();
        rect1.desplazar(40, 20);
        System.out.println("el área es "+rect1.calcularArea());
        int areaRect=new Rectangulo(100, 50).calcularArea();
        System.out.println("el área es "+areaRect);
    }
}
```

La clase *String*

[Clases y objetos](#)[Las clases del lenguaje Java](#)[La clase *String*](#)[Cómo se obtiene información acerca del string](#)[Comparación de strings](#)[Extraer un substring de un string](#)[Convertir un número a string](#)[Convertir un string en número](#)

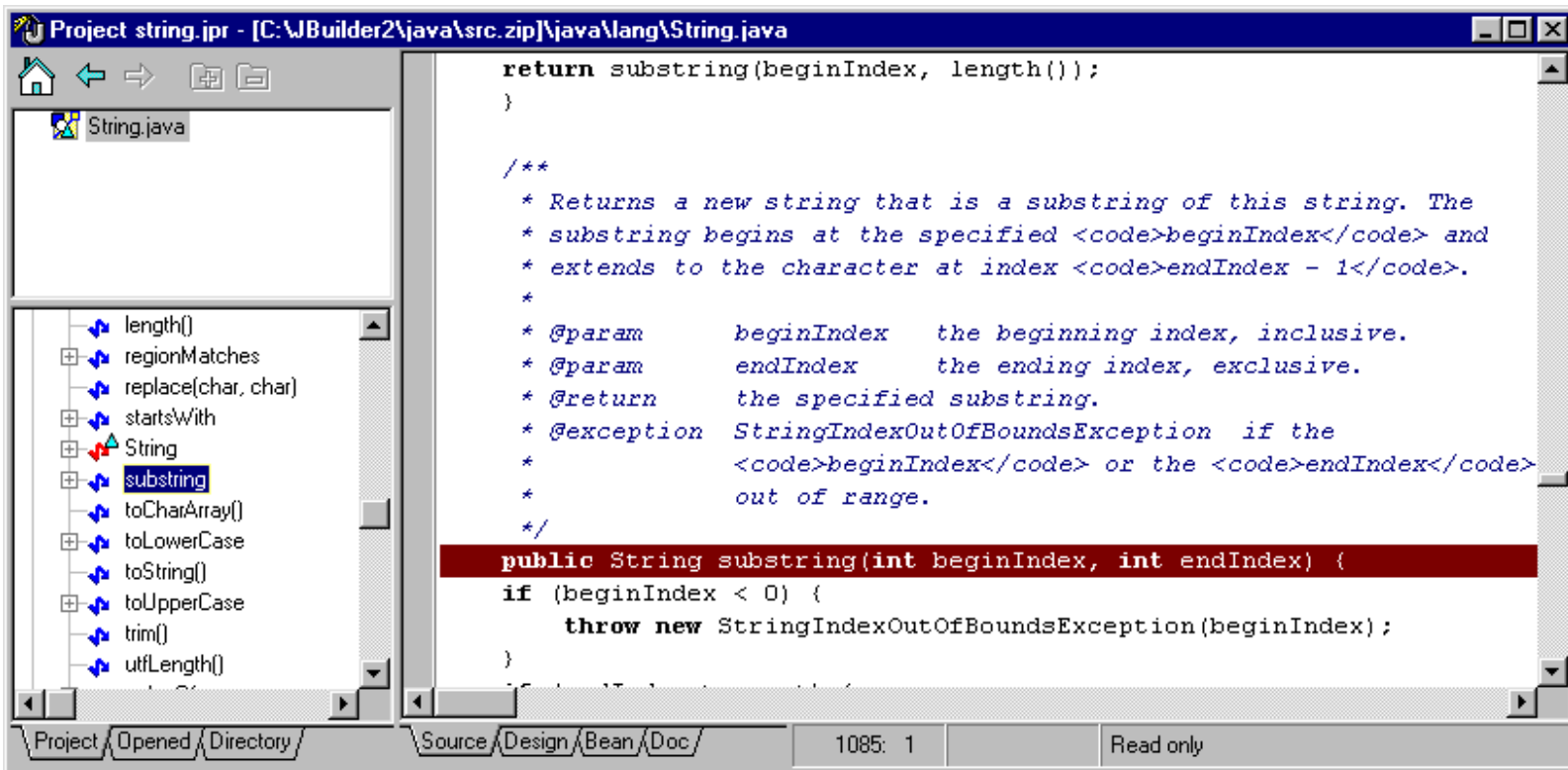
Hemos aprendido a diferenciar entre clase y objetos, a acceder desde un objeto a los miembros datos y a las funciones miembro. Vamos a utilizar clases importantes en el lenguaje Java y a crear objetos de dichas clases. Empezaremos por la clase *String* una de las más importantes del lenguaje Java. Más adelante, volveremos a estudiar otros ejemplos para que el lector se acostumbre a crear sus propias clases.

Las clases del lenguaje Java

Como habremos observado, y apreciaremos aún más en la parte correspondiente a la creación de applets, el IDE JBuilder proporciona un sistema de ayuda a medida que vamos escribiendo el código. También, podemos situar el cursor sobre el nombre de una clase, al pulsar el botón derecho del ratón, aparece un menú flotante. Seleccionando el primer elemento del menú, **Browse symbol at cursor**, aparece la definición de la clase. Los paneles cambian, podemos seleccionar la función miembro que nos interesa en el panel de estructura (inferior izquierda) y ver su código fuente en el panel de contenido (a la derecha).

En la figura podemos ver la clase *String* y en el panel de contenido la definición de una de las funciones miembro *substring* que hemos seleccionado en el panel de estructura. Por encima de la definición aparece la documentación relativa a dicha función.

Para volver al código fuente de nuestra clase pulsamos en el botón "home" encima del panel de navegación.



La clase *String*

string: [StringApp.java](#)

Dentro de un objeto de la clases *String* o *StringBuffer*, Java crea un array de caracteres de una forma similar a como lo hace el lenguaje C++. A este array se accede a través de las funciones miembro de la clase.

Los strings u objetos de la clase *String* se pueden crear explícitamente o implícitamente. Para crear un string implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, cuando se escribe

```
System.out.println("El primer programa");
```

Java crea un objeto de la clase *String* automáticamente.

Para crear un string explícitamente escribimos

```
String str=new String("El primer programa");
```

También se puede escribir, alternativamente

```
String str="El primer programa";
```

Para crear un string nulo se puede hacer de estas dos formas

```
String str="";
String str=new String();
```

Un string nulo es aquél que no contiene caracteres, pero es un objeto de la clase *String*. Sin embargo,

```
String str;
```

está declarando un objeto *str* de la clase *String*, pero aún no se ha creado ningún objeto de esta clase.

Cómo se obtiene información acerca del string

Una vez creado un objeto de la clase *String* podemos obtener información relevante acerca del objeto a través de las funciones miembro.

Para obtener la longitud, número de caracteres que guarda un string se llama a la función miembro *length*.

```
String str="El primer programa";
int longitud=str.length();
```

Podemos conocer si un string comienza con un determinado prefijo, llamando al método *startsWith*, que devuelve **true** o **false**, según que el string comience o no por dicho prefijo

```
String str="El primer programa";
boolean resultado=str.startsWith("El");
```

En este ejemplo la variable resultado tomará el valor **true**.

De modo similar, podemos saber si un string finaliza con un conjunto dado de caracteres, mediante la función miembro *endsWith*.

```
String str="El primer programa";
boolean resultado=str.endsWith("programa");
```

Si se quiere obtener la posición de la primera ocurrencia de la letra p, se usa la función *indexOf*.

```
String str="El primer programa";
int pos=str.indexOf('p');
```

Para obtener las sucesivas posiciones de la letra p, se llama a otra versión de la misma función

```
pos=str.indexOf('p', pos+1);
```


El segundo argumento le dice a la función *indexOf* que empiece a buscar la primera ocurrencia de la letra p a partir de la posición *pos+1*.

Otra versión de *indexOf* busca la primera ocurrencia de un substring dentro del string.

```
String str="El primer programa";
int pos=str.indexOf("pro");
```

Vemos que una clase puede definir varias funciones miembro con el mismo nombre pero que tienen distinto número de parámetros o de distinto tipo.

Comparación de strings

 equals: [EqualsApp.java](#)

La comparación de strings nos da la oportunidad de distinguir entre el operador lógico == y la función miembro *equals* de la clase *String*. En el

siguiente código

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
if(str1==str2){
    System.out.println("Los mismos objetos");
}else{
    System.out.println("Distintos objetos");
}
if(str1.equals(str2)){
    System.out.println("El mismo contenido");
}else{
    System.out.println("Distinto contenido");
}
```

Esta porción de código devolverá que *str1* y *str2* son distintos objetos pero con el mismo contenido. *str1* y *str2* ocupan posiciones distintas en memoria pero guardan los mismos datos.

Cambiemos la segunda sentencia y escribamos

```
String str1="El lenguaje Java";
String str2=str1;
System.out.println("Son el mismo objeto "+(str1==str2));
```

Los objetos *str1* y *str2* guardan la misma referencia al objeto de la clase *String* creado. La expresión (*str1==str2*) devolverá **true**.

Así pues, el método *equals* compara un string con un objeto cualquiera que puede ser otro string, y devuelve **true** cuando dos strings son iguales o **false** si son distintos.

```
String str="El lenguaje Java";
boolean resultado=str.equals("El lenguaje Java");
```

La variable *resultado* tomará el valor **true**.

La función miembro *compareTo* devuelve un entero menor que cero si el objeto string es menor (en orden alfabético) que el string dado, cero si son iguales, y mayor que cero si el objeto string es mayor que el string dado.

```
String str="Tomás";
int resultado=str.compareTo("Alberto");
```

La variable entera *resultado* tomará un valor mayor que cero, ya que Tomás está después de Alberto en orden alfabético.

```
String str="Alberto";
int resultado=str.compareTo("Tomás");
```

La variable entera *resultado* tomará un valor menor que cero, ya que Alberto está antes que Tomás en orden alfabético.

Extraer un substring de un string

En muchas ocasiones es necesario extraer una porción o substring de un string dado. Para este propósito hay una función miembro de la clase *String* denominada *substring*.

Para extraer un substring desde una posición determinada hasta el final del string escribimos

```
String str="El lenguaje Java";
```

La clase `String`

```
String subStr=str.substring(12);
```

Se obtendrá el substring "Java".

Una segunda versión de la función miembro *substring*, nos permite extraer un substring especificando la posición de comienzo y la el final.

```
String str="El lenguaje Java";  
String subStr=str.substring(3, 11);
```

Se obtendrá el substring "lenguaje". Recuérdese, que las posiciones se empiezan a contar desde cero.

Convertir un número a string

Para convertir un número en string se emplea la [función miembro estática](#) *valueOf* (más adelante explicaremos este tipo de funciones).

```
int valor=10;  
String str=String.valueOf(valor);
```

La clase *String* proporciona versiones de *valueOf* para convertir los datos primitivos: **int**, **long**, **float**, **double**.

Esta función se emplea mucho cuando programamos applets, por ejemplo, cuando queremos mostrar el resultado de un cálculo en el área de trabajo de la ventana o en un control de edición.

Convertir un string en número

Cuando introducimos caracteres en un control de edición a veces es inevitable que aparezcan espacios ya sea al comienzo o al final. Para eliminar estos espacios tenemos la función miembro *trim*

```
String str=" 12 ";  
String str1=str.trim();
```

Para convertir un string en número entero, primero quitamos los espacios en blanco al principio y al final y luego, llamamos a la función miembro estática *parseInt* de la clase *Integer* (clase envolvente que describe los números enteros)

```
String str=" 12 ";  
int numero=Integer.parseInt(str.trim());
```

Para convertir un string en número decimal (**double**) se requieren dos pasos: convertir el string en un objeto de la clase envolvente *Double*, mediante la función miembro estática *valueOf*, y a continuación convertir el objeto de la clase *Double* en un tipo primitivo **double** mediante la función *doubleValue*

```
String str="12.35 ";  
double numero=Double.valueOf(str).doubleValue();
```

Se puede hacer el mismo procedimiento para convertir un string a número entero

```
String str="12";  
int numero=Integer.valueOf(str).intValue();
```

La clase *StringBuffer*

En la sección dedicada a los operadores hemos visto que es posible [concatenar cadenas de caracteres](#), es, decir, objetos de la clase *String*. Ahora bien, los objetos de la clase *String* son constantes lo cual significa que por defecto, solamente se pueden crear y leer pero no se pueden modificar.

Imaginemos una función miembro a la cual se le pasa un array de cadenas de caracteres. Los [arrays](#) se estudiarán en la siguiente página.

```
public class CrearMensaje{
    public String getMensaje(String[] palabras){
        String mensaje="";
        for(int i=0; i<palabras.length; i++){
            mensaje+=" "+palabras[i];
        }
        return mensaje;
    }
}

//...
}
```

Cada vez que se añade una nueva palabra, se reserva una nueva porción de memoria y se desecha la vieja porción de memoria que es más pequeña (una palabra menos) para que sea liberada por el [recolector de basura](#) (garbage collector). Si el bucle se realiza 1000 veces, habrá 1000 porciones de memoria que el recolector de basura ha de identificar y liberar.

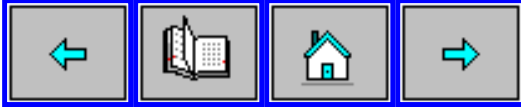
Para evitar este trabajo extra al recolector de basura, se puede emplear la clase *StringBuffer* que nos permite crear objetos dinámicos, que pueden modificarse.

```
public class CrearMensaje{
    public String getMensaje(String[] palabras){
        StringBuffer mensaje=new StringBuffer();
        for(int i=0; i<palabras.length; i++){
            mensaje.append(" ");
            mensaje.append(palabras[i]);
        }
        return mensaje.toString();
    }
}

//...
}
```

La función *append* incrementa la memoria reservada para el objeto *mensaje* con una palabra más sin crear nueva memoria, cada vez que se ejecuta el bucle. La función [toString](#), que veremos más adelante, convierte un objeto en una cadena de caracteres.

Los arrays

[Clases y objetos](#)[Declarar y crear un array](#)[Inicializar y usar el array](#)[Arrays multidimensionales](#)[El código fuente](#)

Un array es un medio de guardar un conjunto de objetos de la misma clase. Se accede a cada elemento individual del array mediante un número entero denominado índice. 0 es el índice del primer elemento y $n-1$ es el índice del último elemento, siendo n , la dimensión del array. Los arrays son objetos en Java y como tales vamos a ver los pasos que hemos de seguir para usarlos convenientemente

- Declarar el array
- Crear el array
- Inicializar los elementos del array
- Usar el array

Declarar y crear un array

Para declarar un array se escribe

```
tipo_de_dato[] nombre_del_array;
```

Para declarar un array de enteros escribimos

```
int[] numeros;
```

Para crear un array de 4 número enteros escribimos

```
numeros=new int[4];
```

La declaración y la creación del array se puede hacer en una misma línea.

```
int[] numeros =new int[4];
```

Inicializar y usar los elementos del array

Para inicializar el array de 4 enteros escribimos

```
numeros[0]=2;  
numeros[1]=-4;  
numeros[2]=15;  
numeros[3]=-25;
```

Se pueden inicializar en un bucle **for** como resultado de alguna operación

```
for(int i=0; i<4; i++){  
    numeros[i]=i*i+4;  
}
```

No necesitamos recordar el número de elementos del array, su miembro dato *length* nos proporciona la dimensión del array. Escribimos de forma equivalente

```
for(int i=0; i<numeros.length; i++){  
    numeros[i]=i*i+4;  
}
```

Los arrays se pueden declarar, crear e inicializar en una misma línea, del siguiente modo

```
int[] numeros={2, -4, 15, -25};  
String[] nombres={"Juan", "José", "Miguel", "Antonio"};
```

Para imprimir a los elementos de array *nombres* se escribe

```
for(int i=0; i<nombres.length; i++){  
    System.out.println(nombres[i]);  
}
```

Java verifica que el índice no sea mayor o igual que la dimensión del array, lo que facilita mucho el trabajo al programador.

Para crear un array de tres objetos de la clase *Rectangulo* se escribe

- Declarar

```
Rectangulo[] rectangulos;
```

- Crear el array

```
rectangulos=new Rectangulo[3];
```

- Inicializar los elementos del array

```
rectangulos[0]=new Rectangulo(10, 20, 30, 40);
rectangulos[1]=new Rectangulo(30, 40);
rectangulos[2]=new Rectangulo(50, 80);
```

O bien, en una sola línea

```
Rectangulo[] rectangulos={new Rectangulo(10, 20, 30, 40),
    new Rectangulo(30, 40), new Rectangulo(50, 80)};
```

- Usar el array

Para calcular y mostrar el área de los rectángulos escribimos

```
for(int i=0; i<rectangulos.length; i++){
    System.out.println(rectangulos[i].calcularArea());
}
```

Arrays multidimensionales

Una matriz bidimensional puede tener varias filas, y en cada fila no tiene por qué haber el mismo número de elementos o columnas. Por ejemplo, podemos declarar e inicializar la siguiente matriz bidimensional

```
double[][] matriz={{1,2,3,4},{5,6},{7,8,9,10,11,12},{13}};
```

- La primer fila tiene cuatro elementos {1,2,3,4}
- La segunda fila tiene dos elementos {5,6}
- La tercera fila tiene seis elementos {7,8,9,10,11,12}
- La cuarta fila tiene un elemento {13}

Para mostrar los elementos de este array bidimensional escribimos el siguiente código

```
for (int i=0; i < matriz.length; i++) {
```

```

        for (int j=0; j < matriz[i].length; j++) {
            System.out.print(matriz[i][j]+"\\t");
        }
        System.out.println(" ");
    }
}

```

Como podemos apreciar, *matriz.length* nos proporciona el número de filas (cuatro), y *matriz[i].length*, nos proporciona el número de elementos en cada fila.

Mostramos los elementos de una fila separados por un tabulador usando la función *print*. Una vez completada una fila se pasa a la siguiente mediante *println*.

Los arrays bidimensionales nos permiten guardar los elementos de una matriz. Queremos crear y mostrar una matriz cuadrada unidad de dimensión 4. Recordaremos que una matriz unidad es aquella cuyos elementos son ceros excepto los de la diagonal principal $i=j$, que son unos. Mediante un doble bucle **for** recorreremos los elementos de la matriz especificando su fila *i* y su columna *j*. En el siguiente programa

- Se crea una matriz cuadrada de dimensión cuatro
- Se inicializa los elementos de la matriz (matriz unidad)
- Se muestra la matriz una fila debajo de la otra separando los elementos de una fila por tabuladores.

```

public class MatrizUnidadApp {
    public static void main (String[] args) {
        double[][] mUnidad= new double[4][4];

        for (int i=0; i < mUnidad.length; i++) {
            for (int j=0; j < mUnidad[i].length; j++) {
                if (i == j) {
                    mUnidad[i][j]=1.0;
                }else {
                    mUnidad[i][j] = 0.0;
                }
            }
        }

        for (int i=0; i < mUnidad.length; i++) {
            for (int j=0; j < mUnidad[i].length; j++) {
                System.out.print(mUnidad[i][j]+"\\t");
            }
            System.out.println(" ");
        }
    }
}

```

Un ejemplo del uso de [break](#) con etiqueta y arrays multidimensionales

```
int[][] matriz={ {32, 87, 3, 589},
                 {12, -30, 190, 0},
                 {622, 127, 981, -3, -5}};

int numero=12;
int i=0, j=0;

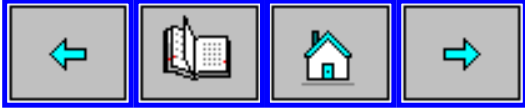
buscado:
    for(i=0; i<matriz.length; i++){
        for(j=0; j<matriz[i].length; j++){
            if(matriz[i][j]==numero){
                break buscado;
            }
        }
    }
    System.out.println("buscado: matriz("+ i+", "+j+")="+matriz[i][j]);
```

El código fuente



[MatrizUnidadApp.java](#)

Los paquetes

[Clases y objetos](#)[El paquete \(**package**\)](#)[La palabra reservada **import**](#)[Paquetes estándar](#)

El paquete (**package**)

- Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.
- Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.
- Las clases tienen ciertos privilegios de acceso a los miembros dato y a las funciones miembro de otras clases dentro de un mismo paquete.

En el Entorno Integrado de Desarrollo (IDE) JBuilder de Borland, un proyecto nuevo se crea en un subdirectorio que tiene el nombre del proyecto. A continuación, se crea la aplicación, un archivo .java que contiene el código de una clase cuyo nombre es el mismo que el del archivo. Se pueden agregar nuevas clases al proyecto, todas ellas contenidas en archivos .java situadas en el mismo subdirectorio. La primera sentencia que encontramos en el código fuente de las distintas clases que forman el proyecto es **package** o del nombre del paquete.

```
//archivo MiApp.java

package nombrePaquete;

public class MiApp{
    //miembros dato
    //funciones miembro
}
```

```
//archivo MiClase.java

package nombrePaquete;

public class MiClase{
    //miembros dato
    //funciones miembro
}
```

La palabra reservada *import*

Para importar clases de un paquete se usa el comando **import**. Se puede importar una clase individual

```
import java.awt.Font;
```

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un asterisco (*) para reemplazar los nombres de clase individuales.

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

Para crear un objeto *fuente* de la clase *Font* podemos seguir dos alternativas

```
import java.awt.Font;
Font fuente=new Font("Monospaced", Font.BOLD, 36);
```

O bien, sin poner la sentencia **import**

```
java.awt.Font fuente=new java.awt.Font("Monospaced", Font.BOLD, 36);
```

Normalmente, usaremos la primera alternativa, ya que es la más económica en código, si tenemos que crear varias fuentes de texto.

Se pueden combinar ambas formas, por ejemplo, en la definición de la clase *BarTexto*

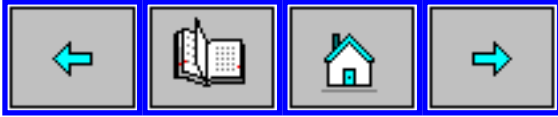
```
import java.awt.*;
public class BarTexto extends Panel implements java.io.Serializable{
    //...
}
```

Panel es una clase que está en el paquete *java.awt*, y *Serializable* es un [interface](#) que está en el paquete *java.io*

Los paquetes estándar

| Paquete | Descripción |
|-------------|--|
| java.applet | Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador |
| java.awt | Contiene clases para crear una aplicación GUI independiente de la plataforma |
| java.io | Entrada/Salida. Clases que definen distintos flujos de datos |
| java.lang | Contiene clases esenciales, se importa implícitamente sin necesidad de una sentencia import . |
| java.net | Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red. |
| java.util | Contiene otras clases útiles que ayudan al programador |

Los números aleatorios




[Clases y objetos](#)

[La clase *Random*](#)

[Comprobación de la uniformidad de los números aleatorios](#)

[Secuencias de números aleatorios](#)

La clase *Random*

 **azar:** [AzarApp.java](#)

Del mismo modo que hemos visualizado el [código fuente de la clase *String*](#), también la podemos obtener el de la clase *Random*. Situamos el cursor en el nombre de la clase y seleccionamos el primer elemento del menú flotante **Browse symbol at cursor** que aparece cuando se pulsa el botón derecho del ratón.

La clase *Random* proporciona un generador de números aleatorios que es más flexible que la función estática [random](#) de la clase *Math*.

Para crear una secuencia de números aleatorios tenemos que seguir los siguientes pasos:

1. Proporcionar a nuestro programa información acerca de la clase *Random*. Al principio del programa escribiremos la siguiente sentencia.

```
import java.util.Random;
```

2. Crear un objeto de la clase *Random*
3. Llamar a una de las funciones miembro que generan un número aleatorio
4. Usar el número aleatorio.

Constructores

La clase dispone de dos constructores, el primero crea un generador de números aleatorios cuya semilla es inicializada en base al instante de tiempo actual.

```
Random rnd = new Random();
```

El segundo, inicializa la semilla con un número del tipo **long**.

```
Random rnd = new Random(3816L);
```

El sufijo L no es necesario, ya que aunque 3816 es un número **int** por defecto, es promocionado automáticamente a **long**.

Aunque no podemos predecir que números se generarán con una semilla particular, podemos sin embargo, duplicar una serie de números aleatorios usando la misma semilla. Es decir, cada vez que creamos un objeto de la clase *Random* con la misma semilla obtendremos la misma secuencia de números aleatorios. Esto no es útil en el caso de loterías, pero puede ser útil en el caso de juegos, exámenes basados en una secuencia de preguntas aleatorias, las mismas para cada uno de los estudiantes, simulaciones que se repitan de la misma forma una y otra vez, etc.

Funciones miembro

Podemos cambiar la semilla de los números aleatorios en cualquier momento, llamando a la función miembro *setSeed*.

```
rnd.setSeed(3816);
```

Podemos generar números aleatorios en cuatro formas diferentes:

```
rnd.nextInt();
```

genera un número aleatorio entero de tipo **int**

```
rnd.nextLong();
```

genera un número aleatorio entero de tipo **long**

```
rnd.nextFloat();
```

genera un número aleatorio de tipo **float** entre 0.0 y 1.0, aunque siempre menor que 1.0

```
rnd.nextDouble();
```

genera un número aleatorio de tipo **double** entre 0.0 y 1.0, aunque siempre menor que 1.0

Casi siempre usaremos esta última versión. Por ejemplo, para generar una secuencia de 10 números aleatorios entre 0.0 y 1.0 escribimos

```
for (int i = 0; i < 10; i++) {
    System.out.println(rnd.nextDouble());
}
```

Para crear una secuencia de 10 números aleatorios enteros comprendidos entre 0 y 9 ambos incluidos escribimos

```
int x;
for (int i = 0; i < 10; i++) {
    x = (int)(rnd.nextDouble() * 10.0);
    System.out.println(x);
}
```

(int) transforma un número decimal **double** en entero **int** eliminando la parte decimal.

Comprobación de la uniformidad de los números aleatorios

Podemos comprobar la uniformidad de los números aleatorios generando una secuencia muy grande de números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive. Contamos cuantos ceros aparecen en la secuencia, cuantos unos, ... cuantos nueves, y guardamos estos datos en los elementos de un array.

Primero creamos un array *ndigitos* de 10 de elementos que son enteros.

```
int[] ndigitos = new int[10];
```

Inicializamos los elementos del array a cero.

```
for (int i = 0; i < 10; i++) {
    ndigitos[i] = 0;
}
```

Creamos una secuencia de 100000 números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive (véase el apartado anterior)

```
for (long i=0; i < 100000L; i++) {
    n = (int)(rnd.nextDouble() * 10.0);
    ndigitos[n]++;
}
```

Si n sale cero suma una unidad al contador de ceros, *ndigitos[0]*. Si n sale uno, suma una unidad al contador de unos, *ndigitos[1]*, y así sucesivamente.

Finalmente, se imprime el resultado, los números que guardan cada uno de los elementos del array *ndigitos*

```
for (int i = 0; i < 10; i++) {
    System.out.println(i+": " + ndigitos[i]);
}
```

Observaremos en la consola que cada número 0, 1, 2...9 aparece aproximadamente 10000 veces.

Secuencias de números aleatorios

En la siguiente porción de código, se imprime dos secuencias de cinco números aleatorios uniformemente distribuidos entre [0, 1), separando los números de cada una de las secuencias por un carácter tabulador.

```
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");
```

Comprobaremos que los números que aparecen en las dos secuencias son distintos.

En la siguiente porción de código, se imprime dos secuencias iguales de números aleatorios uniformemente distribuidos entre $[0, 1)$. Se establece la semilla de los números aleatorios con la función miembro *setSeed*.

```
rnd.setSeed(3816);
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

rnd.setSeed(3816);
System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");
```

```
package azar;

import java.util.Random;

public class AzarApp {
    public static void main (String[] args) {
        int[] ndigitos = new int[10];
        int n;

        Random rnd = new Random();

        // Inicializar el array
        for (int i = 0; i < 10; i++) {
            ndigitos[i] = 0;
        }

        // verificar que los números aleatorios están uniformemente distribuidos
        for (long i=0; i < 100000L; i++) {
            // genera un número aleatorio entre 0 y 9
            n = (int)(rnd.nextDouble() * 10.0);
            //Cuenta las veces que aparece un número
            ndigitos[n]++;
        }
    }
}
```

```
// imprime los resultados
for (int i = 0; i < 10; i++) {
    System.out.println(i+": " + ndigitos[i]);
}

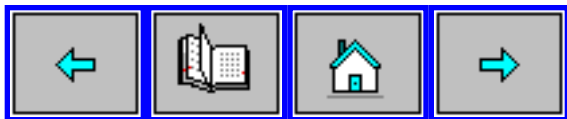
//Dos secuencias de 5 número (distinta semilla)
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

//Dos secuencias de 5 número (misma semilla)
rnd.setSeed(3816L);
System.out.println("Primera secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");

rnd.setSeed(3816);
System.out.println("Segunda secuencia");
for (int i = 0; i < 5; i++) {
    System.out.print("\t"+rnd.nextDouble());
}
System.out.println("");
}
}
```

Creación de clases



Clases y objetos

Definición de la clase *Lista*

Objetos de la clase *Lista*

Crear una clase denominada *Lista* cuyo miembro dato sea un array de números enteros y cuyas funciones miembro realicen las siguientes tareas:

- Hallar y devolver el valor mayor
- Hallar y devolver el valor menor
- Hallar y devolver el valor medio
- Ordenar los números enteros de menor a mayor
- Mostrar la lista ordenada separando los elementos por un tabulador



lista: [Lista.java](#), [ListaApp.java](#)

Definición de la clase *Lista*

Empezamos la definición de la clase escribiendo la palabra reservada **class** y a continuación el nombre de la clase *Lista*.

Los miembros dato

Los miembros dato de la clase *Lista* serán un array de enteros x , y opcionalmente la dimensión del array n .

```
public class Lista {  
    int[] x;        //array de datos  
    int n;          //dimensión
```

El constructor

Al constructor de la clase *Lista* se le pasará un array de enteros para inicializar los miembros dato

```
public Lista(int[] x) {
    this.x=x;
    n=x.length;
}
```

Como apreciamos basta una simple asignación para inicializar el miembro dato x que es un array de enteros, con el array de enteros x que se le pasa al constructor. Por otra parte, cuando se le pasa a una función un array se le pasa implícitamente la dimensión del array, que se puede obtener a partir de su miembro dato *length*.

Las funciones miembro

Las funciones miembro tienen acceso a los miembros dato, el array de enteros x y la dimensión del array n .

- El valor medio

Para hallar el valor medio, se suman todos los elementos del array y se divide el resultado por el número de elementos.

```
double valorMedio(){
    int suma=0;
    for(int i=0; i<n; i++){
        suma+=x[i];
    }
    return (double)suma/n;
}
```

Para codificar esta función se ha de tener algunas precauciones. La suma de todos los elementos del array se guarda en la variable local *suma*. Dicha variable local ha de ser inicializada a cero, ya que una variable local contrariamente a lo que sucede a los miembros dato o variables de instancia es inicializada con cualquier valor en el momento en que es declarada.

La división de dos enteros *suma* y n (número de elementos del array) es un número entero. Por tanto, se ha de [promocionar](#) el entero *suma* de **int** a **double** para efectuar la división y devolver el resultado de

esta operación.

- El valor mayor

```
int valorMayor(){
    int mayor=x[0];
    for(int i=1; i<n; i++){
        if(x[i]>mayor)    mayor=x[i];
    }
    return  mayor;
}
```

Se compara cada elemento del array con el valor de la variable local *mayor*, que inicialmente tiene el valor del primer elemento del array, si un elemento del array es mayor que dicha variable auxiliar se guarda en ella el valor de dicho elemento del array. Finalmente, se devuelve el valor *mayor* calculado

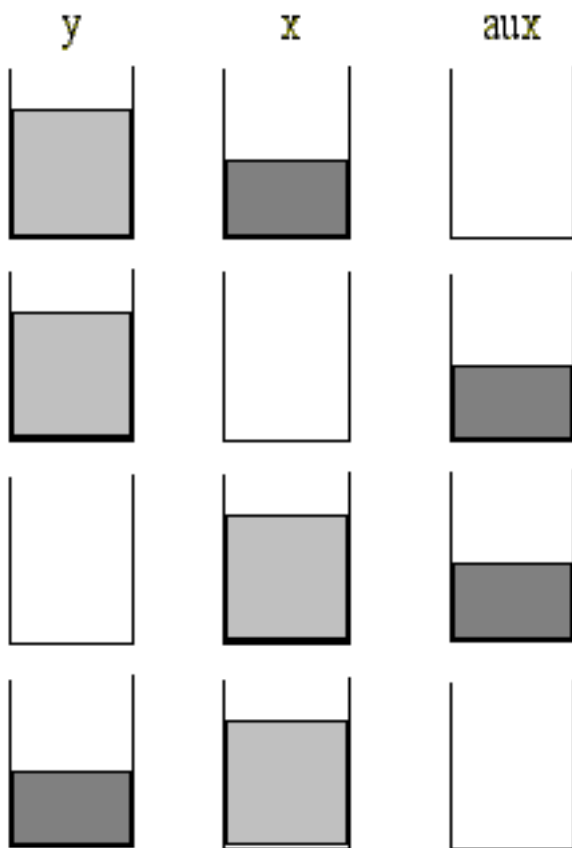
- El valor menor

```
int valorMenor(){
    int menor=x[0];
    for(int i=1; i<n; i++){
        if(x[i]<menor)    menor=x[i];
    }
    return  menor;
}
```

El código es similar a la función *valorMayor*

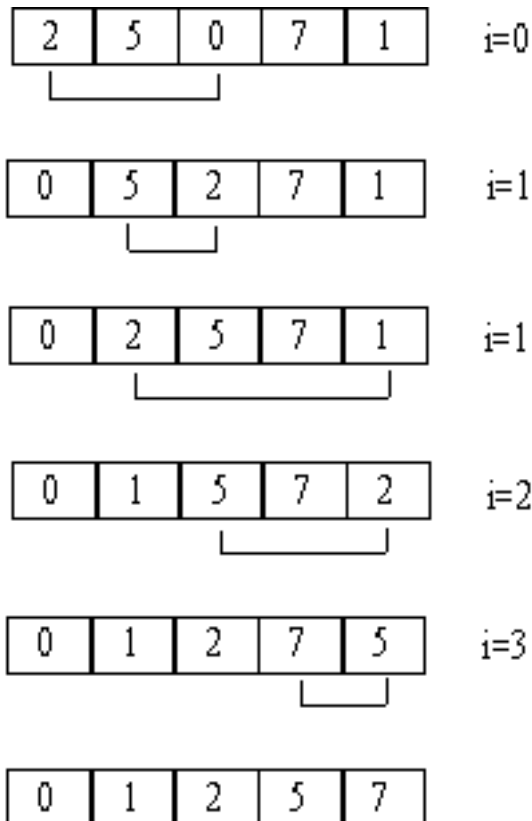
- Ordenar un conjunto de números

En el proceso de ordenación se ha de intercambiar los valores que guardan elementos del array. Veamos como sería el código correspondiente al intercambio de los valores que guardan dos variables *x* e *y*.



Para intercambiar el contenido de dos recipientes *x* e *y* sin que se mezclen, precisamos de un recipiente auxiliar *aux* vacío. Se vuelca el contenido del recipiente *x* en el recipiente *aux*, el recipiente *y* se vuelca en *x*, y por último, el recipiente *aux* se vuelca en *y*. Al final del proceso, el recipiente *aux* vuelve a estar vacío como al principio. En la figura se esquematiza este proceso.

```
aux=x;
x=y;
y=aux;
```



Para ordenar una lista de números emplearemos el método de la burbuja, un método tan simple como poco eficaz. Se compara el primer elemento, índice 0, con todos los demás elementos de la lista, si el primer elemento es mayor que el elemento *j*, se intercambian sus valores, siguiendo el procedimiento explicado en la figura anterior. Se continua este procedimiento con todos los elementos del array menos el último. La figura explica de forma gráfica este procedimiento.

```
void ordenar(){
    int aux;
    for(int i=0; i<n-1; i++){
        for(int j=i+1; j<n; j++){
            if(x[i]>x[j]){
                aux=x[j];
                x[j]=x[i];
                x[i]=aux;
            }
        }
    }
}
```

}

Caben ahora algunas mejoras en el programa, así la función *ordenar* la podemos utilizar para hallar el valor mayor, y el valor menor. Si tenemos una lista ordenada en orden ascendente, el último elemento de la lista será el valor mayor y el primero, el valor menor. De este modo, podemos usar una función en otras funciones, lo que resulta en un ahorro de código, y en un aumento de la legibilidad del programa.

```
int valorMayor(){
    ordenar();
    return x[n-1];
}
```

```
int valorMenor(){
    ordenar();
    return x[0];
}
```

- Imprimir la lista ordenada

Imprimimos la lista ordenada separando sus elementos por un tabulador. Primero, se llama a la función *ordenar*, y después se imprime un elemento a continuación del otro mediante *System.out.print*. Recuerdese, que *System.out.println* imprime y a continuación pasa a la siguiente línea.

```
void imprimir(){
    ordenar();
    for(int i=0; i<n; i++){
        System.out.print("\t"+x[i]);
    }
    System.out.println("");
}
```

El código completo de la clase *Lista*, es el siguiente

```

public class Lista {
    int[] x;        //array de datos
    int n;          //dimensión
    public Lista(int[] x) {
        this.x=x;
        n=x.length;
    }
    double valorMedio(){
        int suma=0;
        for(int i=0; i<n; i++){
            suma+=x[i];
        }
        return (double)suma/n;
    }
    int valorMayor(){
        int mayor=x[0];
        for(int i=1; i<n; i++){
            if(x[i]>mayor)    mayor=x[i];
        }
        return  mayor;
    }
    int valorMenor(){
        int menor=x[0];
        for(int i=1; i<n; i++){
            if(x[i]<menor)    menor=x[i];
        }
        return  menor;
    }
    void ordenar(){
        int aux;
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[i]>x[j]){
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;
                }
            }
        }
    }
    void imprimir(){
        ordenar();
        for(int i=0; i<n; i++){

```

```

        System.out.print("\t"+x[i]);
    }
    System.out.println(" ");
}
}

```

Los objetos de la clase *Lista*

A partir de un array de enteros podemos crear un objeto *lista* de la clase *Lista*.

```

int[] valores={10, -4, 23, 12, 16};
Lista lista=new Lista(valores);

```

Estas dos sentencias las podemos convertir en una

```

Lista lista=new Lista(new int[] {10, -4, 23, 12, 16});

```

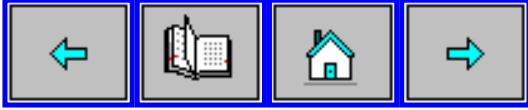
En el resto del código, el objeto *lista* llama a las funciones miembro

```

System.out.println("Valor mayor "+lista.valorMayor());
System.out.println("Valor menor "+lista.valorMenor());
System.out.println("Valor medio "+lista.valorMedio());
lista.imprimir();

```

Miembros estáticos

[Clases y objetos](#)[Variables de instancia, variables de clase](#)[Miembros estáticos](#)

Variables de instancia, variables de clase



circulo: [Circulo.java](#), [CirculoApp.java](#)

Ya mencionamos la diferencia entre [variables de instancia, de clase y locales](#). Consideremos de nuevo la clase *Circulo*, con dos miembros dato, el *radio* específico para cada círculo y el número *PI* que tiene el mismo valor para todos los círculos. La primera es una variable de instancia y la segunda es una variable de clase.

Para indicar que el miembro *PI* es una variable de clase se le antepone el modificador **static**. El modificador **final** indica que es una constante que no se puede modificar, una vez que la variable *PI* ha sido inicializada.

Definimos también una función miembro denominada *calcularArea* que devuelva el área del círculo

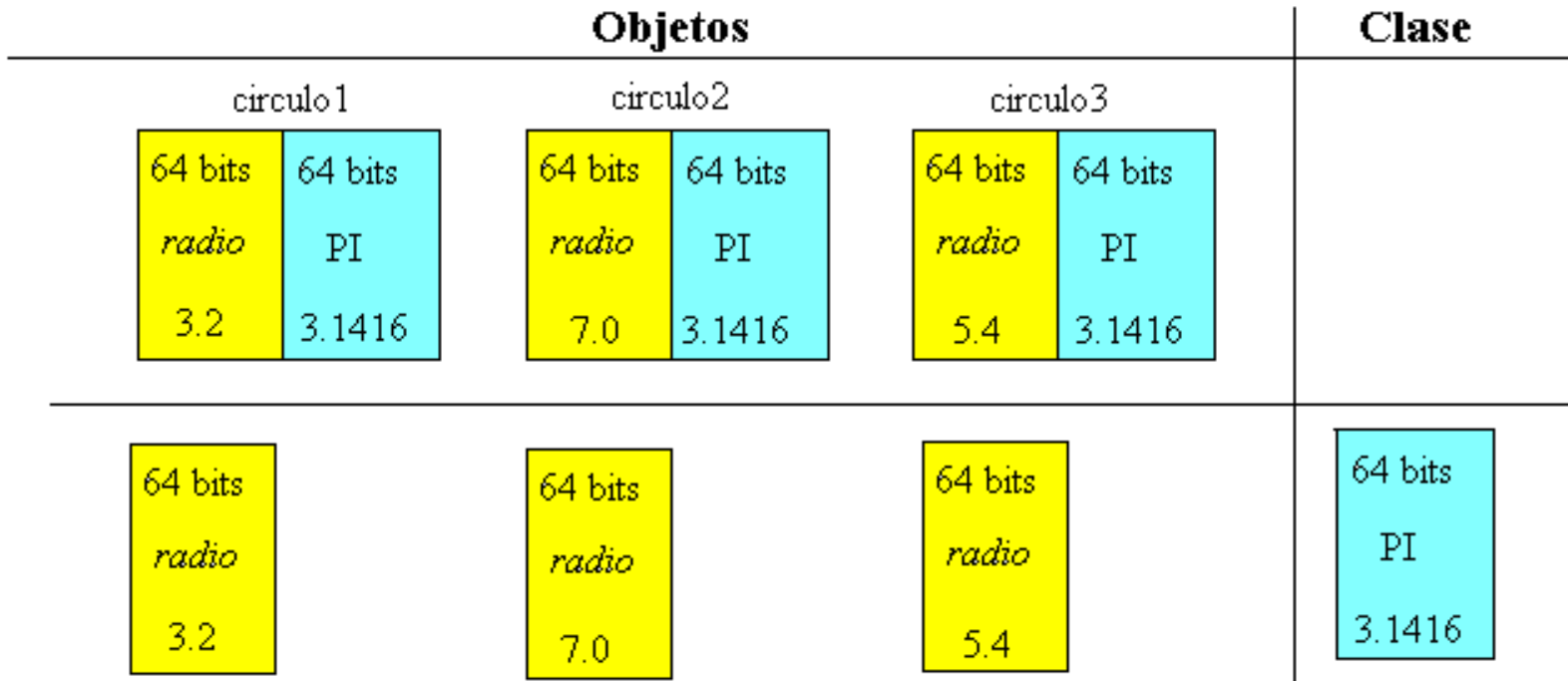
```
public class Circulo{
    static final double PI=3.1416;
    double radio;
    public Circulo(double radio){
        this.radio=radio;
    }
    double calcularArea(){
        return (PI*radio*radio);
    }
}
```

Para calcular el área de un círculo, creamos un objeto *circulo* de la clase *Circulo* dando un valor al radio. Desde este objeto llamamos a la función miembro *calcularArea*.

```
Circulo circulo=new Circulo(2.3);
```

```
System.out.println("área: "+circulo.calcularArea());
```

Veamos ahora las ventajas que supone declarar la constante PI como miembro estático.



Si PI y *radio* fuesen variables de instancia

```
public class Circulo{
    double PI=3.1416;
    double radio;
    //....
}
```

Creamos tres objetos de la clase *Circulo*, de radios 3.2, 7.0, y 5.4

```
Circulo circulo1=new Circulo(3.2);
Circulo circulo2=new Circulo(7.0);
Circulo circulo3=new Circulo(5.4);
```

Al crearse cada objeto se reservaría espacio para el dato *radio* (64 bits), y para el dato PI (otros 64 bits). Véase la sección [tipos de datos primitivos](#). Como vemos en la parte superior de la figura, se desperdicia la memoria del ordenador, guardando tres veces el mismo dato PI.

```
public class Circulo{
    static double PI=3.1416;
    double radio;
    //....
}
```

}

Declarando PI estático (**static**), la variable PI queda ligada a la clase *Circulo*, y se reserva espacio en memoria una sólo vez, tal como se indica en la parte inferior de la figura. Si además la variable PI no cambia, es una constante, le podemos anteponer la palabra **final**.

```
public class Circulo{
    static final double PI=3.1416;
    double radio;
    //....
}
```

Miembros estáticos



alumno: [Alumno.java](#), [AlumnoApp.java](#)

Las variables de clase o miembros estáticos son aquellos a los que se antepone el modificador **static**. Vamos a comprobar que un miembro dato estático guarda el mismo valor en todos los objetos de dicha clase.

Sea una clase denominada *Alumno* con dos miembros dato, la nota de selectividad, y un miembro estático denominado nota de corte. La nota es un atributo que tiene un valor distinto para cada uno de los alumnos u objetos de la clase *Alumno*, mientras que la nota de corte es un atributo que tiene el mismo valor para a un conjunto de alumnos. Se define también en dicha clase una función miembro que determine si está (**true**) o no (**false**) admitido.

```
public class Alumno {
    double nota;
    static double notaCorte=6.0;
    public Alumno(double nota) {
        this.nota=nota;
    }
    boolean estaAdmitido(){
        return (nota>=notaCorte);
    }
}
```

Creamos ahora un array de cuatro alumnos y asignamos a cada uno de ellos una nota.

```
Alumno[] alumnos={new Alumno(5.5), new Alumno(6.3),
    new Alumno(7.2), new Alumno(5.0)};
```

Contamos el número de alumnos que están admitidos


```

int numAdmitidos=0;
for(int i=0; i<alumnos.length; i++){
    if (alumnos[i].estaAdmitido()){
        numAdmitidos++;
    }
}
System.out.println("admitidos "+numAdmitidos);

```

Accedemos al miembro dato *notaCorte* desde un objeto de la clase *Alumno*, para cambiarla a 7.0

```
alumnos[1].notaCorte=7.0;
```

Comprobamos que todos los objetos de la clase *Alumno* tienen dicho miembro dato estático *notaCorte* cambiado a 7.0

```

for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}

```

El miembro dato *notaCorte* tiene el modificador **static** y por tanto está ligado a la clase más que a cada uno de los objetos de dicha clase. Se puede acceder a dicho miembro con la siguiente sintaxis

```
Nombre_de_la_clase.miembro_estático
```

Si ponemos

```

Alumno.notaCorte=6.5;
for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}

```

Veremos que todos los objetos de la clase *Alumno* habrán cambiado el valor del miembro dato estático *notaCorte* a 6.5.

Un miembro dato estático de una clase se puede acceder desde un objeto de la clase, o mejor, desde la clase misma.

La clase *Math*



[Clases y objetos](#)

[Miembros dato constantes](#)

[Funciones miembro](#)

[Cálculo del número irracional \$\pi\$](#)

La clase *Math* tiene miembros dato y funciones miembro estáticas, vamos a conocer algunas de estas funciones, cómo se llaman y qué tarea realizan.

 matemáticas: [MatematicasApp.java](#)

Miembros dato constantes

La clase *Math* define dos constantes muy útiles, el número π y el número e.

```
public final class Math {  
    public static final double E = 2.7182818284590452354;  
    public static final double PI = 3.14159265358979323846;  
    //...  
}
```

El modificador **final** indica que los valores que guardan no se pueden cambiar, son valores constantes

Se accede a estas constantes desde la clase *Math*, de la siguiente forma

```
System.out.println("Pi es " + Math.PI);  
System.out.println("e es " + Math.E);
```

Funciones miembro

La clase *Math* define muchas funciones y versiones distintas de cada función.

Por ejemplo, para hallar el valor absoluto de un número define las siguientes funciones. Se llama a una u otra dependiendo del tipo de dato que se le pasa en su único argumento.

```
public final class Math {
    public static int abs(int a) {
        return (a < 0) ? -a : a;
    }
    public static long abs(long a) {
        return (a < 0) ? -a : a;
    }
    public static float abs(float a) {
        return (a < 0) ? -a : a;
    }
    public static double abs(double a) {
        return (a < 0) ? -a : a;
    }
    //...
}
```

Por ejemplo, hallar el valor absoluto de los siguientes números

```
int i = -9;
double x = 0.3498;
System.out.println("|" + i + "|" es " + Math.abs(i));
System.out.println("|" + x + "|" es " + Math.abs(x));
```

Math.abs(i), llama a la primera versión, y *Math.abs(x)* llama a la última versión.

Funciones trigonométricas

En las funciones trigonométricas los argumentos se expresan en radianes. Por ejemplo, el ángulo 45° se convierte en radianes y luego se halla el seno, el coseno y la tangente

```
double angulo = 45.0 * Math.PI/180.0;
System.out.println("cos(" + angulo + ") es " + Math.cos(angulo));
System.out.println("sin(" + angulo + ") es " + Math.sin(angulo));
System.out.println("tan(" + angulo + ") es " + Math.tan(angulo));
```

Para pasar de coordenadas rectangulares a polares es útil la función *atan2*, que admite dos argumentos, la ordenada y la abscisa del punto. Devuelve el ángulo en radianes.

```
double y=-6.2; //ordenada
double x=1.2; //abscisa
```

```
System.out.println("atan2(" + y+" , "+x + ") es " + Math.atan2(y, x));
```

Funciones exponencial y logarítmica

La función exponencial *exp* devuelve el número *e* elevado a una potencia

```
System.out.println("exp(1.0) es " + Math.exp(1.0));
System.out.println("exp(10.0) es " + Math.exp(10.0));
System.out.println("exp(0.0) es " + Math.exp(0.0));
```

La función *log* calcula el logaritmo natural (de base *e*) de un número

```
System.out.println("log(1.0) es " + Math.log(1.0));
System.out.println("log(10.0) es " + Math.log(10.0));
System.out.println("log(Math.E) es " + Math.log(Math.E));
```

Función potencia y raíz cuadrada

Para elevar un número *x* a la potencia *y*, se emplea *pow*(*x*, *y*)

```
System.out.println("pow(10.0, 3.5) es " + Math.pow(10.0,3.5));
```

Para hallar la raíz cuadrada de un número, se emplea la función *sqrt*

```
System.out.println("La raíz cuadrada de " + x + " is " + Math.sqrt(x));
```

Aproximación de un número decimal

Para expresar un número real con un número especificado de números decimales empleamos la función *round*. Por ejemplo, para expresar los números *x* e *y* con dos cifras decimales escribimos

```
double x = 72.3543;
double y = 0.3498;
System.out.println(x + " es aprox. " + (double)Math.round(x*100)/100);
System.out.println(y + " es aprox. " + (double)Math.round(y*100)/100);
```

Se obtiene 72.35 y 0.35 como cabría esperar. Fijarse que *round* devuelve un número entero **int** que es necesario promocionar a **double** para efectuar la división entre 100.

Si empleamos la función *floor* en vez de *round* obtendríamos

```
System.out.println(x + " es aprox. " + Math.floor(x*100)/100);
System.out.println(y + " es aprox. " + Math.floor(y*100)/100);
```

Se obtiene 72.35 y 0.34. La aproximación del primero es correcta ya que la tercera cifra decimal es 4 inferior a 5. La

aproximación del segundo es incorrecta ya que la tercera cifra decimal es 9 mayor que 5. En la mayor parte de los cálculos se cometen errores, por lo que la diferencia entre *floor* y *round* no es significativa.

El mayor y el menor de dos números

Para hallar el mayor y el menor de dos números se emplean las funciones *min* y *max* que comparan números del mismo tipo.


```
int i = 7;
int j = -9;
double x = 72.3543;
double y = 0.3498;
// para hallar el menor de dos número
System.out.println("min(" + i + "," + j + ") es " + Math.min(i,j));
System.out.println("min(" + x + "," + y + ") es " + Math.min(x,y));
// Para hallar el mayor de dos números
System.out.println("max(" + i + "," + j + ") es " + Math.max(i,j));
System.out.println("max(" + x + "," + y + ") es " + Math.max(x,y));
```

Números aleatorios

La clase *Math* define una función denominada *random* que devuelve un número pseudoaleatorio comprendido en el intervalo [0.0, 1.0). Existe otra alternativa, se pueden generar números pseudoaleatorios a partir de un objeto de la [clase Random](#), que llame a la función miembro *nextDouble*.

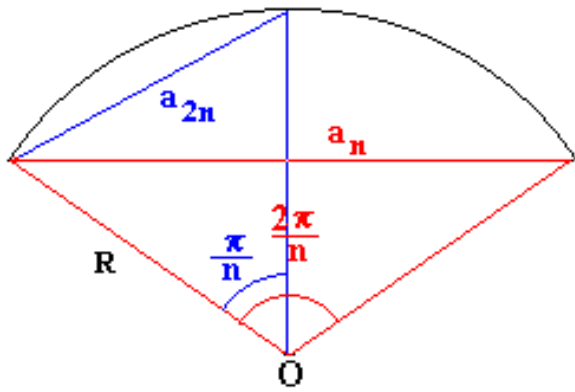
```
System.out.println("Número aleatorio: " + Math.random());
System.out.println("Otro número aleatorio: " + Math.random());
```

Cálculo del número irracional π

 [pi: PiApp.java](#)

Para hallar la longitud de una circunferencia de radio R , primero se calcula el perímetro de un triángulo equilátero (3 lados) inscrito en dicha circunferencia, luego, de un hexágono (6 lados), un dodecágono (12 lados), y así sucesivamente. El límite de la sucesión de perímetros es precisamente la longitud de la circunferencia $2\pi R$.

Si tomamos una circunferencia de radio unidad, al dividir entre dos los valores de los perímetros iremos obteniendo las sucesivas aproximaciones del número irracional π .



A partir de la figura, podemos calcular la longitud del lado a_n un polígono regular de n lados inscrito en la circunferencia de radio R , (en color rojo).

$$a_n = 2R \operatorname{sen} \frac{\pi}{n}$$

Del mismo modo, obtenemos la longitud del lado de un polígono regular inscrito de $2n$ lados (en color azul).

$$a_{2n} = 2R \operatorname{sen} \frac{\pi}{2n}$$

Teniendo en cuenta que $\operatorname{sen} \frac{\alpha}{2} = \sqrt{\frac{1 - \cos \alpha}{2}}$

Establecemos la relación entre a_n y a_{2n} y por tanto, entre el perímetro P_n del polígono regular de n lados y el perímetro P_{2n} del polígono regular de $2n$ lados.

$$P_{2n} = 2nR \sqrt{2 - \sqrt{4 - \frac{P_n^2}{R^2 n^2}}}$$

Tomando como radio R , la unidad

- Para un triángulo, $n=3$, la longitud del lado es $a_3=2\operatorname{sen}60^\circ$, y el perímetro $P_3 = 3\sqrt{3}$
- Para un hexágono, $n=6$, la longitud del lado es $a_6=2\operatorname{sen}30^\circ=1$, y el perímetro $P_6=6$.
- y así sucesivamente.

Para obtener las sucesivas aproximaciones del número irracional π mediante la fórmula anterior procedemos del siguiente modo

1. Partimos del valor del perímetro P de un triángulo equilátero inscrito en una circunferencia de radio unidad, el valor de n es 3.
2. Calculamos el perímetro P de un polígono de $2n$ lados a partir del valor del perímetro de un polígono regular de n lados.
3. El valor obtenido P será el valor del perímetro de un polígono regular de $n=2n$ lados.
4. Se imprime el valor de P dividido entre dos (aproximación de π)
5. Se vuelve al paso 2.

Ahora, hemos de trasladar las fórmulas matemáticas a código, y aquí es donde podemos llevarnos algunas sorpresas.

En primer lugar, hemos de tener en cuenta que la expresión $\frac{(P \cdot P)}{(n \cdot n)}$ es matemáticamente equivalente a $\left(\frac{P}{n}\right)\left(\frac{P}{n}\right)$ pero no lo es cuando trabajamos con números en el ordenador.

Por ejemplo si n es tipo de dato **int**. Al evaluar el denominador en la primera expresión obtenemos el cuadrado de n que crece muy rápidamente con n , sobrepasándose (overflow) el [valor máximo que puede guardar una variable entera](#) dado por *Integer.MAX_VALUE*. *Integer* es la clase que describe los números enteros. Por tanto, al realizar los cálculos en el ordenador es aconsejable emplear la segunda expresión en vez de la primera, incluso si cambiamos el tipo de dato de n de **int** a **long**.

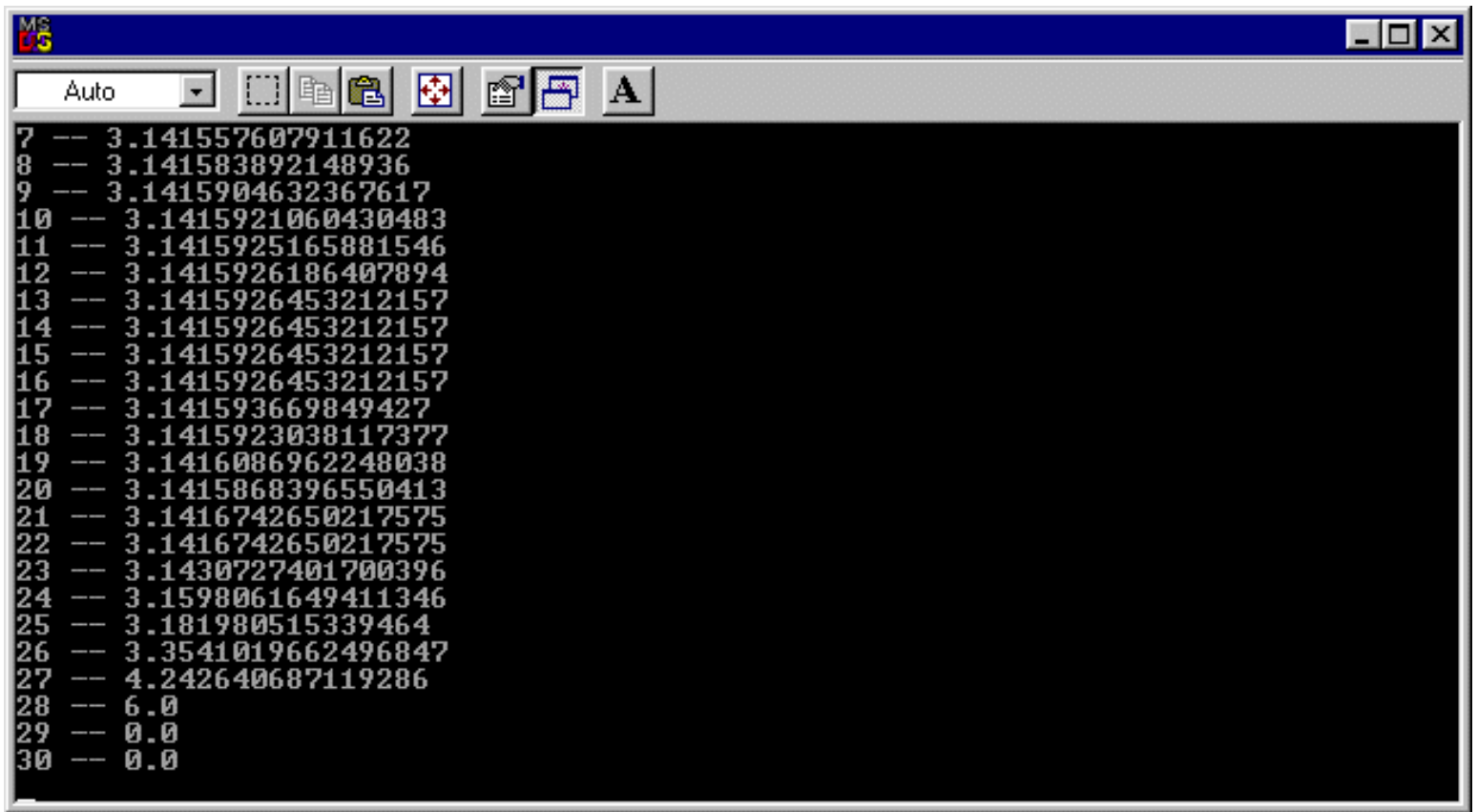
El cálculo de π implica un número infinito de iteracciones, ya que como hemos visto no es posible al sobrepasarse el valor máximo que puede guardar una variable entera, nuestra primera intención sería la programar un bucle que realice el máximo número de iteracciones

```
double perimetro=3*Math.sqrt(3);           //triángulo equilátero inscrito
long n=3;
int i=0;      //número de iteracciones
while(n<Long.MAX_VALUE){
    perimetro=2*n*Math.sqrt(2.0-Math.sqrt(4.0-(perimetro/n)*(perimetro/n)));
    n=2*n;
    i++;
    System.out.println(i+" -- "+perimetro/2);
}
```

Con cierta sorpresa observamos la salida del programa cuando se ha completado el bucle, se imprime un cero, en vez de 3.14159265358979323846.

Si observamos las 30 primeras iteracciones vemos, tal como se muestra en la figura inferior, que la valor más próximo a π se obtiene en las iteracciones 13, 14, 15, y 16.

```
while(i<30){
//...
}
```



```
7 -- 3.141557607911622
8 -- 3.141583892148936
9 -- 3.1415904632367617
10 -- 3.1415921060430483
11 -- 3.1415925165881546
12 -- 3.1415926186407894
13 -- 3.1415926453212157
14 -- 3.1415926453212157
15 -- 3.1415926453212157
16 -- 3.1415926453212157
17 -- 3.141593669849427
18 -- 3.1415923038117377
19 -- 3.1416086962248038
20 -- 3.1415868396550413
21 -- 3.1416742650217575
22 -- 3.1416742650217575
23 -- 3.1430727401700396
24 -- 3.1598061649411346
25 -- 3.181980515339464
26 -- 3.3541019662496847
27 -- 4.242640687119286
28 -- 6.0
29 -- 0.0
30 -- 0.0
```

La conclusión final, es que hemos de tener mucho cuidado al trasladar las fórmulas matemáticas a código. Los datos de tipo predefinido solamente pueden guardar valores entre un máximo y un mínimo, tal como [hemos visto en su definición](#). Por otra parte, una variable de tipo **double** tiene una precisión limitada por lo que no representa a todos los números reales sino a un conjunto finito de éstos.

Una clase con funciones estáticas



[Clases y objetos](#)

[El valor absoluto de un número](#)

[La potencia de exponente entero de un número entero](#)

[El factorial de un número](#)

[Determinar si un número es o no primo](#)

[La clase *Matematicas*](#)

[Llamada a las funciones miembro](#)

El propósito de este capítulo es definir un conjunto de funciones estáticas dentro de una clase denominada *Matematicas*. Al definir el concepto de clase estudiamos las [funciones miembro](#) o métodos. En esta página, vamos a ver cómo se define una función para realizar una determinada tarea.

Se deberá tener en cuenta que no se puede usar una variable de instancia por una función estática. Por ejemplo

```
class UnaClase{
    int i=0;
    static void funcion(){
        System.out.println(i);
    }
    //otras funciones miembro
}
```

Al compilar se produce un error ya que la variable *i* es de instancia y no se puede usar en un contexto estático. El lector puede probar esto con cualquier clase que describe una aplicación. Declara una variable de instancia en la clase y la trata de usar en la función estática *main*.



mates: [Matematicas.java](#), [MatesApp.java](#)

El valor absoluto de un número.

- Nombre de la función: `absoluto`
- Parámetros: un entero
- Devuelve: un número entero positivo.

El valor absoluto de un número x es x si el número es positivo y $-x$ si el número es negativo. Una función *absoluto* que calcula el valor absoluto de un número recibirá el dato del número, y devolverá el valor absoluto del mismo. Supongamos que dicho dato es entero, la definición de la función será:

```
int absoluto(int x){
    int abs;
    if(x>0) abs=x;
    else abs=-x;
    return abs;
}
```

Podemos simplificar el código sin necesidad de declarar una variable temporal *abs*.

```
int absoluto(int x){
    if(x>0) return x;
    else return -x;
}
```

O bien,

```
int absoluto(int x){
    if(x>0) return x;
    return -x;
}
```

Una función similar nos hallará el valor absoluto de un dato de tipo **double**.

```
double absoluto(double x){
    if(x<0) return -x;
    return x;
}
```

La potencia de exponente entero de un número entero

- Nombre de la función: potencia
- Parámetros: la base y el exponente, ambos enteros
- Devuelve: el resultado, un número del tipo **long**.

Para hallar la potencia de un número se multiplica tantas veces la base como indica el exponente. Por ejemplo, para hallar la quinta potencia de 3, se escribirá

$$3^5 = 3 * 3 * 3 * 3 * 3$$

Podemos realizar este producto en un bucle **for**, de manera que en la variable *resultado* se vaya acumulando el resultado de los sucesivos productos, tal como se recoge en la Tabla, entre paréntesis figura el valor de *resultado* en la iteración previa, el valor inicial es 1.

| iteración | valor de <i>resultado</i> |
|-----------|---------------------------|
| 1ª | (1)*3 |
| 2ª | (1*3)*3 |
| 3ª | (1*3*3)*3 |
| 4ª | (1*3*3*3)*3 |
| 5ª | (1*3*3*3*3)*3 |

```
long resultado=1;
for(int i=0; i<5; i++)
    resultado*=3;
```

El siguiente paso es la generalización del proceso a un exponente positivo cualquiera y a una base entera cualesquiera. La variable *resultado* es de tipo **long**, porque al hallar la potencia de un número entero se puede sobrepasar el rango de dichos números.

```
long resultado=1;
for(int i=0; i<exponente; i++)
    resultado*=base;
```

Por último, le pondremos una etiqueta a esta tarea, asociaremos esta rutina al nombre de una función.

```

long potencia(int base, int exponente){
    long resultado=1;
    for(int i=0; i<exponente; i++){
        resultado*=base;
    }
    return resultado;
}

```

El factorial de un número

- Nombre de la función: factorial
- Parámetros: un entero **int**
- Devuelve: el resultado, un número positivo del tipo **long**.

Como ejemplos de los [sentencias iterativas](#) **for** y **while** ya tuvimos la ocasión de calcular el factorial de un número. Ahora se trata de poner las líneas de código dentro de una función para que pueda ser reutilizada.

```

long resultado=1;
while(numero>0){
    resultado*=numero;
    numero--;
}

```

Es mucho más lógico definir una función que calcule el factorial del número *num*,

```

long factorial(int num){
    long resultado=1;
    while(num>0){
        resultado*=num;
        num--;
    }
    return resultado;
}

```

que repetir tres veces el código del cálculo del factorial para hallar el número combinatorio m sobre n , de acuerdo a la siguiente fórmula.

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

El numerador y el denominador del número combinatorio m sobre n se obtiene del siguiente modo:

```
long num=factorial(m);
long den=factorial(n)*factorial(m-n);
```

Determinar si un número es o no primo

- Nombre de la función: `esPrimo`
- Parámetros: un entero **int**
- Devuelve: **true** o **false**, un dato del tipo **boolean**

Como ejemplos de los [sentencias iterativas](#) **for** y **break** ya tuvimos la ocasión de calcular los números primos comprendidos entre 3 y 100. Recuerdese que

- un número par no es primo, excepto el 2
- un número impar es primo si no es divisible por los números impares comprendidos entre 1 y $\text{numero}/2$

```
boolean bPrimo=true;
for(int i=3; i<numero/2; i+=2){
    if(numero%i==0){
        bPrimo=false;
        break;
    }
}
```

Ahora, se trata de poner las líneas de código dentro de una función que determine si un número es primo (devuelve **true**) o no es primo (devuelve **false**), para que pueda ser reutilizada. El código de la función *esPrimo* es la siguiente

```
boolean esPrimo(int numero){
    if(numero==2)        return true;
    if(numero%2==0)      return false;
    boolean bPrimo=true;
    for(int i=3; i<numero/2; i+=2){
        if(numero%i==0){
            bPrimo=false;
            break;
        }
    }
    return bPrimo;
}
```

```
}
```

Podemos ahorrarnos la variable *bPrimo*, escribiendo la función *esPrimo* de la siguiente forma equivalente

```
boolean esPrimo(int numero){
    if((numero!=2)&&(numero%2==0))        return false;
    for(int i=3; i<numero/2; i+=2){
        if(numero%i==0){
            return false;
        }
    }
    return true;
}
```

La clase *Matematicas*

Ahora solamente nos queda poner todas las funciones en el interior de una clase que le llamaremos *Matematicas*, anteponiéndole a cada una de las funciones la palabra reservada **static**.

```
public class Matematicas {
    static long factorial(int num){
        //...
    }
    static long potencia(int base, int exponente){
        //...
    }
    static boolean esPrimo(int numero){
        //...
    }
    static double absoluto(double x){
        //...
    }
    static int absoluto(int x){
        //...
    }
}
```

Llamada a las funciones miembro

Para llamar a las funciones miembro se escribe

```
Nombre_de_la_clase.función_miembro(...);
```

- Hallar el número combinatorio m sobre n

```
int m=8, n=3;
System.out.print("El número combinatorio "+m+" sobre "+n);
long numerador=Matematicas.factorial(m);
long denominador=Matematicas.factorial(m-n)*Matematicas.factorial(n);
System.out.println(" vale "+numerador+" / "+denominador);
```

- Encontrar los números primos comprendidos entre 100 y 200

```
for(int num=100; num<200; num++){
    if(Matematicas.esPrimo(num)){
        System.out.print(num+" - ");
    }
}
```

- Hallar la potencia de un número entero

```
System.out.print("La potencia 5 elevado a 4 ");
System.out.print("vale "+Matematicas.potencia(5, 4));
```

- Hallar el valor absoluto de un número

```
double y=-2.5;
System.out.print("El valor absoluto de "+y);
System.out.println(" vale "+Matematicas.absoluto(y));
```

La clase *Fraccion*



Clases y objetos

[Los miembros dato](#)

[Las funciones miembro](#)

[La clase *Fraccion*](#)

[Uso de la clase *Fraccion*](#)

[Modificadores de acceso](#)

[Mejora de la clase *Lista*](#)

En esta página vamos a definir una clase denominada *Fraccion* con dos miembros dato: el numerador y el denominador, y varias funciones miembro que realizan las operaciones entre fracciones. El lenguaje Java no tiene la característica de la sobrecarga de operadores como el lenguaje C++. En este lenguaje es posible sobrecargar los operadores aritméticos, como funciones miembro o como funciones amigas (friend) para que se realicen las operaciones entre entidades definidas por el usuario tal como las pensamos o las escribimos en un papel. Por ejemplo, si a y b son dos fracciones (objetos de la clase *Fraccion*) podemos escribir

$$c=a+b;$$

para obtener la fracción c resultado de la suma de a y b .

Definiremos las operaciones en Java de un modo similar al lenguaje C, pero como en Java no existen funciones que no sean miembros de una clase, definiremos las operaciones como funciones estáticas de una clase que denominamos *Fraccion*.



fraccion1: [Fraccion.java](#), [FraccionApp1.java](#)

Los miembros dato

Consideremos la clase que describe una fracción que denominaremos *Fraccion*. Consta de dos miembros ambos enteros, el numerador *num*, y del denominador *den*.

```
public class Fraccion {
    int num;
    int den;
    //...
}
```

Las funciones miembro

Además de los constructores definiremos varias funciones miembro que codifican las operaciones que se realizan con fracciones: suma de dos fracciones, diferencia de dos fracciones, producto, cociente, fracción inversa de una dada, y simplificar dos fracciones. Finalmente, redefiniremos la función *toString* para obtener una representación en forma de texto de una fracción.

Los constructores

Definiremos dos constructores, el constructor por defecto, que da al numerador el valor cero, y al denominador el valor uno, y el constructor explícito.

```
public Fraccion() {
    num=0;
    den=1;
}
public Fraccion(int x, int y) {
    num=x;
    den=y;
}
```

Suma de dos fracciones

Se tratará de definir una función denominada *sumar*, que realice las operación de sumar dos fracciones. Por tanto, la función *sumar* tendrá dos parámetros que son dos fracciones *a* y *b*, y devolverá una fracción,

su declaración será

```
Fraccion sumar(Fraccion a, Fraccion b){
    //...
}
```

Para codificar la función plantearemos el procedimiento de sumar dos fracciones a y b , cuyos numeradores son $a.num$ y $b.num$, y cuyos denominadores son $a.den$ y $b.den$, respectivamente. El resultado se guarda en la fracción c . El numerador $c.num$ y el denominador $c.den$ se obtienen del siguiente modo:

$$\frac{c.num}{c.den} = \frac{a.num * b.den + a.den * b.num}{a.den * b.den}$$

La suma de dos fracciones es otra fracción c que tiene por numerador $c.num$.

```
c.num=a.num*b.den+b.num*a.den;
```

y por denominador $c.den$

```
c.den=a.den*b.den;
```

Una vez efectuada la suma, la función *sumar* devuelve la fracción c

```
return c;
```

El código completo de la función *sumar* es

```
Fraccion sumar(Fraccion a, Fraccion b){
    Fraccion c=new Fraccion();
    c.num=a.num*b.den+b.num*a.den;
    c.den=a.den*b.den;
    return c;
}
```

Diferencia de dos fracciones

La función *restar* es semejante a la función *sumar* y no requiere más explicación.

```
Fraccion restar(Fraccion a, Fraccion b){
```

```

    Fraccion c=new Fraccion();
    c.num=a.num*b.den-b.num*a.den;
    c.den=a.den*b.den;
    return c;
}

```

Producto de dos fracciones

Cuando se multiplican dos fracciones a y b , se obtiene otra fracción c cuyo numerador es el producto de los numeradores, y cuyo denominador es el producto de sus denominadores respectivos.

$$\frac{c.num}{c.den} = \frac{a.num*b.num}{a.den*b.den}$$

```

Fraccion multiplicar(Fraccion a, Fraccion b){
    Fraccion c=new Fraccion();
    c.num=a.num*b.num;
    c.den=a.den*b.den;
    return c;
}

```

Podemos ahorrarnos la fracción temporal c , y escribir

```

Fraccion multiplicar(Fraccion a, Fraccion b){
    return new Fraccion(a.num*b.num, a.den*b.den);
}

```

Inversa de una fracción

La función *inversa*, recibe una fracción en su único argumento y devuelve una fracción cuyo numerador es el denominador de la fracción argumento, y cuyo denominador es el numerador de dicha fracción.

$$\frac{1}{a.num/a.den} = \frac{a.den}{a.num}$$

```

public static Fraccion inversa(Fraccion a){
    return new Fraccion(a.den, a.num);
}

```

Cociente de dos fracciones

Cuando se dividen dos fracciones a y b , se obtiene otra fracción c cuyo numerador es el producto del numerador de la primera por el denominador de la segunda, y cuyo denominador es el producto del denominador de la primera por el numerador de la segunda.

$$\frac{c.num}{c.den} = \frac{a.num * b.den}{a.den * b.num}$$

```
Fraccion dividir(Fraccion a, Fraccion b){
    return new Fraccion(a.num*b.den, a.den*b.num);
}
```

La operación división de dos fracciones es equivalente a multiplicar la fracción a por la inversa de b , de este modo aprovechamos el código de la función *inversa*.

```
Fraccion dividir(Fraccion a, Fraccion b){
    return multiplicar(a, inversa(b));
}
```

Simplificar una fracción

Para simplificar una fracción primero hay que hallar el máximo común divisor del numerador y del denominador. la función *mcd* se encarga de esta tarea. Para ello emplea el algoritmo de Euclides, cuyo funcionamiento se muestra en el siguiente ejemplo. Sea $u=1260$ y $v=231$,

- En la primera iteración, se halla el resto r de dividir el primero u entre el segundo v . Se asigna a u el divisor v , y se asigna a v el resto r .
- En la segunda iteración, se halla el resto r de dividir u entre v . Se asigna a u el divisor v , y se asigna a v el resto r .
- Se repite el proceso hasta que el resto r sea cero. El máximo común divisor será el último valor de v .

$$1260=231*5+105$$

$$231=105*2+21$$

$$105=21*5+0$$

el máximo común divisor es 21.

Definimos en la clase *Fraccion* una función *mcd* que calcula y devuelve el máximo común divisor del numerador y del denominador.

```
int mcd() {
    int u=Math.abs(num);
    int v=Math.abs(den);
    if(v==0){
        return u;
    }
    int r;
    while(v!=0){
        r=u%v;
        u=v;
        v=r;
    }
    return u;
}
```

A continuación definimos la función simplificar, de modo que al aplicarlo sobre una fracción, dicha fracción se reduzca a la fracción equivalente más simple. Para ello, se divide numerador y denominador por el máximo común divisor de ambos números, y devuelve la fracción simplificada.

```
Fraccion simplificar(){
    int dividir=mcd();
    num/=dividir;
    den/=dividir;
    return this;
}
```

Aquí tenemos otro ejemplo del uso de la palabra reservada **this**. Los miembros dato cambian al dividirlos entre el máximo común divisor y la función devuelve el objeto actual, **this**.

La función miembro *toString*

Para mostrar una fracción podemos definir una función miembro denominada *imprimir*

```
public void imprimir(){
```

```
System.out.println(num+" / "+den);
```

```
}
```

Un objeto de la clase *Fraccion* llama a la función miembro *imprimir* para mostrar en la consola (una ventana DOS) los valores que guardan sus miembros *dato*, *num* y *den*, el numerador y el denominador. La función *imprimir* así definida no nos servirá cuando la clase *Fraccion* se emplee en un contexto gráfico. Ahora bien, como vamos a ver a continuación el lenguaje Java nos proporciona una solución a este problema.

Aunque no se define explícitamente, la clase *Fraccion* deriva de la [clase base *Object*](#) (la estudiaremos en el siguiente capítulo) y redefine la función miembro pública *toString*, cuya tarea es la de dar una representación en forma de texto de la fracción.

```
public String toString(){
    String texto=num+" / "+den;
    return texto;
}
```

En la definición de *toString* vemos que el operador + se usa para concatenar strings (el lenguaje Java convierte automáticamente un dato primitivo en su representación textual cuando se concatena con un string).

Para mostrar en la consola el numerador y el denominador de una fracción (objeto de la clase *Fraccion*) *a* basta escribir

```
System.out.println(a);
```

Lo que equivale a la llamada explícita

```
System.out.println(a.toString());
```

Si queremos mostrar la fracción *a* en un contexto gráfico *g* de un applet o de un canvas escribimos

```
g.drawString("fracción: "+a, 20, 30);
```

donde 20, 30 son las coordenadas de la línea base del primer carácter. Esta sentencia equivale a la llamada explícita

```
g.drawString("fracción: "+a.toString(), 20, 30);
```

La redefinición de la función *toString* devuelve un string un objeto de la clase *String* que guarda la

representación en forma de texto de los objetos de una determinada clase. De este modo, una clase que redefina *toString* puede emplearse en cualquier ámbito.

La clase *Fraccion*

Ahora ponemos las funciones miembro dentro de la clase *Fraccion*, anteponiendo en las funciones que representan operaciones la palabra reservada **static**.

```
public class Fraccion {
    private int num;
    private int den;
    public Fraccion() {
        num=0;
        den=1;
    }
    public Fraccion(int x, int y) {
        num=x;
        den=y;
    }
    public static Fraccion sumar(Fraccion a, Fraccion b){
        Fraccion c=new Fraccion();
        c.num=a.num*b.den+b.num*a.den;
        c.den=a.den*b.den;
        return c;
    }
    public static Fraccion restar(Fraccion a, Fraccion b){
        Fraccion c=new Fraccion();
        c.num=a.num*b.den-b.num*a.den;
        c.den=a.den*b.den;
        return c;
    }
    public static Fraccion multiplicar(Fraccion a, Fraccion b){
        return new Fraccion(a.num*b.num, a.den*b.den);
    }
    public static Fraccion inversa(Fraccion a){
        return new Fraccion(a.den, a.num);
    }
    public static Fraccion dividir(Fraccion a, Fraccion b){
        return multiplicar(a, inversa(b));
    }
}
```

```

private int mcd(){
    int u=Math.abs(num);
    int v=Math.abs(den);
    if(v==0){
        return u;
    }
    int r;
    while(v!=0){
        r=u%v;
        u=v;
        v=r;
    }
    return u;
}
public Fraccion simplificar(){
    int dividir=mcd();
    num/=dividir;
    den/=dividir;
    return this;
}
public String toString(){
    String texto=num+" / "+den;
    return texto;
}
}

```

Uso de la clase *Fraccion*

Como vemos en la definición de la clase *Fraccion* tenemos funciones estáticas y no estáticas. Vamos a ver la diferencia entre las llamadas a funciones estáticas y no estáticas.

- Crear un objeto de la clase *Fraccion* o una fracción

```
Fraccion x=new Fraccion(2,3);
```

- Mostrar una fracción

```
System.out.println("x--> "+x);
```


Cuando se pone una fracción x como argumento de la función *println* o se concatena con un string se llama automáticamente a la función miembro *toString*, lo que equivale a la siguiente llamada

```
System.out.println("x--> "+x.toString());
```

- Suma de dos fracciones

```
Fraccion x=new Fraccion(2,3);
Fraccion y=new Fraccion(4,3);
System.out.println("x+y= "+Fraccion.sumar(x, y));
```

- Producto de dos fracciones

```
Fraccion x=new Fraccion(2,3);
Fraccion y=new Fraccion(4,3);
System.out.println("x*y= "+Fraccion.multiplicar(x, y));
```

- Operaciones combinadas

Primero suma las fracciones x e y y luego hace el producto con la fracción z

```
Fraccion x=new Fraccion(2,3);
Fraccion y=new Fraccion(4,3);
Fraccion z=new Fraccion(1,2);
Fraccion resultado=Fraccion.multiplicar(Fraccion.sumar(x,y), z);
System.out.println("(x+y)*z= "+resultado);
```

- Simplificar una fracción

```
System.out.println(resultado.simplificar());
```

Modificadores de acceso

Este ejemplo ilustra una faceta importante de los lenguajes de Programación Orientada a Objetos denominada encapsulación. El acceso a los miembros de una clase está controlado. Para usar una clase, solamente necesitamos saber que funciones miembro se pueden llamar y a qué datos podemos acceder, no necesitamos saber como está hecha la clase, como son sus detalles internos. Una vez que la clase está

depurada y probada, la clase es como una caja negra. Los objetos de dicha clase guardan unos datos, y están caracterizados por una determinada conducta. Este ocultamiento de la información niega a la entidades exteriores el acceso a los miembros privados de un objeto. De este modo, las entidades exteriores acceden a los datos de una manera controlada a través de algunas funciones miembro. Para acceder a un miembro público (dato o función) basta escribir.

```
objeto_de_la_clase_Fraccion.miembro_público_no_estático
clase_Fraccion.miembro_público_estático
```

Delante de los miembros dato, como podemos ver en el listado hemos puesto las palabras reservadas **public** y **private**.

- Miembros públicos

Los miembros públicos son aquellos que tienen delante la palabra **public**, y se puede acceder a ellos sin ninguna restricción.

- Miembros privados

Los miembros privados son aquellos que tienen delante la palabra **private**, y se puede acceder a ellos solamente dentro del ámbito de la clase.

Los miembros dato *num* y *den* son privados, y también la función que calcula el máximo común divisor *mcd*, que es una función auxiliar de la función miembro publica *simplificar*. El usuario solamente precisa saber que dispone de una función pública que le permite simplificar una fracción, pero no necesita saber cuál es el procedimiento empleado para simplificar fracciones. Así declaramos la función *mcd* como privada y *simplificar* como pública.

- Por defecto (a nivel de paquete)

Cuando no se pone ningún modificador de acceso delante de los miembros, se dice que son accesibles dentro del mismo [paquete \(package\)](#). Esto es lo que hemos hecho en los ejemplos estudiados hasta esta sección.

package es la primera sentencia que se pone en un archivo .java. El nombre del paquete es el mismo que el nombre del subdirectorío que contiene los archivos .java. Cada archivo .java contiene habitualmente una clase. Si tiene más de una solamente una de ellas es pública. El nombre de dicha clase coincide con el nombre del archivo.

Como el lector se habrá dado cuenta [hay una correspondencia](#) entre archivos y clases, entre paquetes y subdirectoríos. El Entorno Integrado de Desarrollo (IDE) en el que creamos los programas facilita esta

tarea sin que el usuario se aperciba de ello.

Mejora de la clase *Lista*



list1: [Lista.java](#), [ListaApp1.java](#)

Veamos un ejemplo más, [la clase *Lista*](#), y hagamos uso de la función miembro *ordenar* para hallar el *valorMenor* y el *valorMayor*. Si tenemos una lista ordenada en orden creciente, el valor menor es el primer elemento de la lista $x[0]$, y el valor mayor es el último elemento de la lista $x[n-1]$. Podemos escribir el siguiente código

```
public int valorMayor(){
    ordenar();
    return x[n-1];
}
public int valorMenor(){
    ordenar();
    return x[0];
}
```

Podemos llamar una sólo vez a la función miembro *ordenar* en el constructor, después de haber creado el array, y evitar así la reiteración de llamadas a dicha función en *valorMayor*, *valorMenor* e *imprimir*.

```
public Lista(int[] x) {
    this.x=x;
    n=x.length;
    ordenar();
}
public int valorMayor(){
    return x[n-1];
}
public int valorMenor(){
    return x[0];
}
```

La función miembro *ordenar*, es una función auxiliar de las otras funciones miembro públicas, por tanto, podemos ponerle delante el modificador de acceso **private**. El usuario solamente está interesado en el valor medio, el valor mayor y menor de un conjunto de datos, pero no está interesado en el

procedimiento que permite *ordenar* el conjunto de datos. Como ocurre en la vida moderna usamos muchos aparatos pero no tenemos por que conocer sus detalles internos y cómo funcionan por dentro. Una clase es como uno de estos aparatos modernos, el usuario solamente tiene que conocer qué hace la clase, a qué miembros tiene acceso, pero no como está implementada en software.

La función miembro *toString*

Si los miembros dato de la clase *Lista* son privados (**private**) hemos de definir una función que hemos denominado *imprimir* para mostrar los valores que guardan los miembros dato de los objetos de la clase *Lista*.

```
public class Lista {
    private int[] x;        //array de datos
    private int n;          //dimensión
    //...
    public void imprimir(){
        for(int i=0; i<n; i++){
            System.out.print("\t"+x[i]);
        }
        System.out.println(" ");
    }
}
```

La llamada a esta función miembro se efectúa desde un objeto de la clase *Lista*

```
Lista lista=new Lista(new int[]{60, -4, 23, 12, -16});
System.out.println("Mostrar la lista");
lista.imprimir();
```

Sustituímos la función miembro *imprimir* por la redefinición de *toString*. Para [redefinir una función](#), tiene que tener el mismo nombre, los mismos modificadores, el mismo tipo de retorno y los mismos parámetros y del mismo tipo en la clase base y en la clase derivada. Para evitar errores, el mejor procedimiento es el de ir al código de la clase base *Object*, copiar la línea de la declaración de *toString*, pegarla en la definición de nuestra clase, y a continuación definir dicha función.

```
public class Lista {
    private int[] x;        //array de datos
    private int n;          //dimensión
    //...
    public String toString(){
        String texto=" ";
    }
```

```

        for(int i=0; i<n; i++){
            texto+="\t"+x[i];
        }
        return texto;
    }

```

La llamada a la función *toString* se realiza implícitamente en el argumento de la función *System.out.println*, o bien, al concatenar un string y un objeto de la clase *Lista*.

```

Lista lista=new Lista(new int[]{60, -4, 23, 12, -16});
System.out.println("Mostrar la lista");
System.out.println(lista);

```

```

public class Lista {
    private int[] x;        //array de datos
    private int n;          //dimensión
    public Lista(int[] x) {
        this.x=x;
        n=x.length;
        ordenar();
    }
    public double valorMedio(){
        int suma=0;
        for(int i=0; i<n; i++){
            suma+=x[i];
        }
        return (double)suma/n;
    }
    public int valorMayor(){
        return x[n-1];
    }
    public int valorMenor(){
        return x[0];
    }
    private void ordenar(){
        int aux;
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[i]>x[j]){
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;
                }
            }
        }
    }
}

```

```
        }  
    }  
}  
  
public String toString(){  
    String texto="";  
    for(int i=0; i<n; i++){  
        texto+="\t"+x[i];  
    }  
    return texto;  
}  
}
```

La clase base y la clase derivada



La herencia y el polimorfismo

[Introducción](#)

[La clase base](#)

[La clase derivada](#)

[Modificadores de acceso](#)

[La clase base *Object*](#)

Introducción

La herencia es una propiedad esencial de la Programación Orientada a Objetos que consiste en la creación de nuevas clases a partir de otras ya existentes. Este término ha sido prestado de la Biología donde afirmamos que un niño tiene la cara de su padre, que ha heredado ciertas facetas físicas o del comportamiento de sus progenitores.

La herencia es la característica fundamental que distingue un lenguaje orientado a objetos, como el C++ o Java, de otro convencional como C, BASIC, etc. Java permite heredar a las clases características y conductas de una o varias clases denominadas base. Las clases que heredan de clases base se denominan derivadas, estas a su vez pueden ser clases bases para otras clases derivadas. Se establece así una clasificación jerárquica, similar a la existente en Biología con los animales y las plantas.

La herencia ofrece una ventaja importante, permite la reutilización del código. Una vez que una clase ha sido depurada y probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad se puede cambiar derivando una nueva clase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y dinero, ya que solamente tiene que verificar la nueva conducta que proporciona la clase derivada.

La programación en los entornos gráficos, en particular Windows, con el lenguaje C++, es un ejemplo ilustrativo. Los compiladores como los de Borland y Microsoft proporcionan librerías cuyas clases describen el aspecto y la conducta de las ventanas, controles, menús, etc. Una de estas clases denominada *TWindow* describe el aspecto y la conducta de una ventana, tiene una función miembro denominada *Paint*, que no dibuja nada en el área de trabajo de la misma. Definiendo una clase derivada de *TWindow*, podemos redefinir en ella la función *Paint* para que dibuje una figura. Aprovechamos de este modo la ingente cantidad y complejidad del código necesario para

crear una ventana en un entorno gráfico. Solamente, tendremos que añadir en la clase derivada el código necesario para dibujar un rectángulo, una elipse, etc.

En el lenguaje Java, todas las clases derivan implícitamente de la clase base *Object*, por lo que heredan las funciones miembro definidas en dicha clase. Las clases derivadas pueden redefinir algunas de estas funciones miembro como [*toString*](#) y definir otras nuevas.

Para crear un applet, solamente tenemos que definir una clase derivada de la clase base *Applet*, redefinir ciertas funciones como *init* o *paint*, o definir otras como las respuestas a las acciones sobre los controles.

Los programadores crean clases base:

1. Cuando se dan cuenta que diversos tipos tienen algo en común, por ejemplo en el juego del ajedrez peones, alfiles, rey, reina, caballos y torres, son piezas del juego. Creamos, por tanto, una clase base y derivamos cada pieza individual a partir de dicha clase base.
2. Cuando se precisa ampliar la funcionalidad de un programa sin tener que modificar el código existente.

La clase base



ventana: [Ventana.java](#), [VentanaTitulo.java](#), [VentanaApp.java](#)

Vamos a poner un ejemplo del segundo tipo, que simule la utilización de librerías de clases para crear un interfaz gráfico de usuario como Windows 3.1 o Windows 95.

Supongamos que tenemos una clase que describe la conducta de una ventana muy simple, aquella que no dispone de título en la parte superior, por tanto no puede desplazarse, pero si cambiar de tamaño actuando con el ratón en los bordes derecho e inferior.

La clase *Ventana* tendrá los siguientes miembros dato: la posición *x* e *y* de la ventana, de su esquina superior izquierda y las dimensiones de la ventana: *ancho* y *alto*.

```
public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;
    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
}
```



```

    }
    //...
}

```

Las funciones miembros, además del constructor serán las siguientes: la función *mostrar* que simula una ventana en un entorno gráfico, aquí solamente nos muestra la posición y las dimensiones de la ventana.

```

public void mostrar(){
    System.out.println("posición      : x="+x+", y="+y);
    System.out.println("dimensiones   : w="+ancho+", h="+alto);
}

```

La función *cambiarDimensiones* que simula el cambio en la anchura y altura de la ventana.

```

public void cambiarDimensiones(int dw, int dh){
    ancho+=dw;
    alto+=dh;
}

```

El código completo de la clase base *Ventana*, es el siguiente

```

package ventana;

public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;
    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
    public void mostrar(){
        System.out.println("posición      : x="+x+", y="+y);
        System.out.println("dimensiones   : w="+ancho+", h="+alto);
    }
    public void cambiarDimensiones(int dw, int dh){
        ancho+=dw;
        alto+=dh;
    }
}

```

Objetos de la clase base

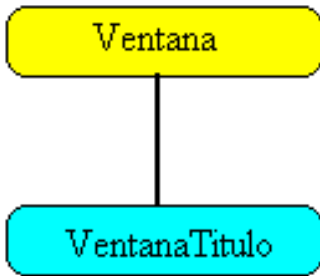
Como vemos en el código, el constructor de la clase base inicializa los cuatro miembros dato. Llamamos al constructor creando un objeto de la clase *Ventana*

```
Ventana ventana=new Ventana(0, 0, 20, 30);
```

Desde el objeto *ventana* podemos llamar a las funciones miembro públicas

```
ventana.mostrar();
ventana.cambiarDimensiones(10, 10);
ventana.mostrar();
```

La clase derivada



Incrementamos la funcionalidad de la clase *Ventana* definiendo una clase derivada denominada *VentanaTitulo*. Los objetos de dicha clase tendrán todas las características de los objetos de la clase base, pero además tendrán un título, y se podran desplazar (se simula el desplazamiento de una ventana con el ratón).

La clase derivada heredar  los miembros dato de la clase base y las funciones miembro, y tendr  un miembro dato m s, el t tulo de la ventana.

```
public class VentanaTitulo extends Ventana{
    protected String titulo;
    public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h);
        titulo=nombre;
    }
}
```

extends es la palabra reservada que indica que la clase *VentanaTitulo* deriva, o es una subclase, de la clase *Ventana*.

La primera sentencia del constructor de la clase derivada es una llamada al constructor de la clase base mediante la palabra reservada **super**. La llamada

```
super(x, y, w, h);
```

inicializa los cuatro miembros dato de la clase base *Ventana*: *x*, *y*, *ancho*, *alto*. A continuación, se inicializa los miembros dato de la clase derivada, y se realizan las tareas de inicialización que sean necesarias. Si no se llama explícitamente al constructor de la clase base Java lo realiza por nosotros, llamando al constructor por defecto si existe.

La función miembro denominada *desplazar* cambia la posición de la ventana, añadiéndoles el desplazamiento.

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Redefine la función miembro *mostrar* para mostrar una ventana con un título.

```
public void mostrar(){
    super.mostrar();
    System.out.println("titulo      : "+titulo);
}
```

En la clase derivada se define una función que tiene el mismo nombre y los mismos parámetros que la de la clase base. Se dice que redefinimos la función *mostrar* en la clase derivada. La función miembro *mostrar* de la clase derivada *VentanaTitulo* hace una llamada a la función *mostrar* de la clase base *Ventana*, mediante

```
super.mostrar();
```

De este modo aprovechamos el código ya escrito, y le añadimos el código que describe la nueva funcionalidad de la ventana por ejemplo, que muestre el título.

Si nos olvidamos de poner la palabra reservada **super** llamando a la función *mostrar*, tendríamos una función recursiva. La función *mostrar* llamaría a *mostrar* indefinidamente.

```
public void mostrar(){ //¡ojo!, función recursiva
    System.out.println("titulo      : "+titulo);
    mostrar();
}
```

La definición de la clase derivada *VentanaTitulo*, será la siguiente.

```
package ventana;

public class VentanaTitulo extends Ventana{
    protected String titulo;
    public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h);
        titulo=nombre;
    }
    public void mostrar(){
        super.mostrar();
        System.out.println("titulo      : "+titulo);
    }
    public void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

Objetos de la clase derivada

Creamos un objeto *ventana* de la clase derivada *VentanaTitulo*

```
VentanaTitulo ventana=new VentanaTitulo(0, 0, 20, 30, "Principal");
```

Mostramos la ventana con su título, llamando a la función *mostrar*, redefinida en la clase derivada

```
ventana.mostrar();
```

Desde el objeto *ventana* de la clase derivada llamamos a las funciones miembro definidas en dicha clase

```
ventana.desplazar(4, 3);
```

Desde el objeto *ventana* de la clase derivada podemos llamar a las funciones miembro definidas en la clase base.

```
ventana.cambiarDimensiones(10, -5);
```

Para mostrar la nueva ventana desplazada y cambiada de tamaño escribimos

```
ventana.mostrar();
```

Modificadores de acceso

Ya hemos visto el significado de los [modificadores de acceso](#) **public** y **private**, así como el control de acceso por defecto a nivel de paquete, cuando no se especifica nada. En la herencia, surge un nuevo control de acceso denominado **protected**.

Hemos puesto **protected** delante de los miembros dato x e y de la clase base *Ventana*

```
public class Ventana {
    protected int x;
    protected int y;
    //...
}
```

En la clase derivada la función miembro *desplazar* accede a dichos miembros dato

```
public class VentanaTitulo extends Ventana{
    //...
    public void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

Si cambiamos el modificador de acceso de los miembros x e y de la clase base *Ventana* de **protected** a **private**, veremos que el compilador se queja diciendo que los miembros x e y no son accesibles.

Los miembros *ancho* y *alto* se pueden poner con acceso **private** sin embargo, es mejor dejarlos como **protected** ya que podrían ser utilizados por alguna función miembro de otra clase derivada de *VentanaTitulo*. Dentro de una jerarquía pondremos un miembro con acceso **private**, si estamos seguros de que dicho miembro solamente va a ser usado por dicha clase.

Como vemos hay cuatro modificadores de acceso a los miembros dato y a los métodos: **private**, **protected**, **public** y **default** (por defecto, o en ausencia de cualquier modificador). La herencia complica aún más el problema de acceso, ya que las clases dentro del mismo paquete tienen diferentes accesos que las clases de distinto paquete

Los siguientes cuadros tratan de aclarar este problema

| Clases dentro del mismo paquete | | |
|---------------------------------|----------|-----------|
| Modificador de acceso | Heredado | Accesible |

| | | |
|-------------------------------|----|----|
| Por defecto (sin modificador) | Si | Si |
| private | No | No |
| protected | Si | Si |
| public | Si | Si |

| Clases en distintos paquetes | | |
|-------------------------------|----------|-----------|
| Modificador de acceso | Heredado | Accesible |
| Por defecto (sin modificador) | No | No |
| private | No | No |
| protected | Si | No |
| public | Si | Si |

Desde el punto de vista práctico, cabe reseñar que no se heredan los miembros privados, ni aquellos miembros (dato o función) cuyo nombre sea el mismo en la clase base y en la clase derivada.

La clase base *Object*

La clase *Object* es la clase raíz de la cual derivan todas las clases. Esta derivación es implícita.

La clase *Object* define una serie de funciones miembro que heredan todas las clases. Las más importantes son las siguientes

```
public class Object {
    public boolean equals(Object obj) {
        return (this == obj);
    }
    protected native Object clone() throws CloneNotSupportedException;
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
    protected void finalize() throws Throwable { }
    //otras funciones miembro...
}
```

Igualdad de dos objetos:

Hemos visto que el método [*equals*](#) de la clase *String* cuando compara un string y cualquier otro objeto. El método *equals* de la clase *Object* compara dos objetos uno que llama a la función y otro es el argumento de dicha función.

Representación en forma de texto de un objeto

El método *toString* imprime por defecto el nombre de la clase a la que pertenece el objeto y su código (hash). Esta función miembro se redefine en la clase derivada para mostrar la información que nos interese acerca del objeto. La [*clase Fraccion*](#) redefine *toString* para mostrar el numerador y el denominador separados por la barra de dividir. En la misma página, hemos [*mejorado la clase Lista*](#) para mostrar los datos que se guardan en los objetos de dicha clase, redefiniendo *toString*.

La función *toString* se llama automáticamente siempre que pongamos un objeto como argumento de la función *System.out.println* o concatenado con otro string.

Duplicación de objetos

El [*método clone crea un objeto duplicado*](#) (clónico) de otro objeto. Más adelante estudiaremos en detalle la redefinición de esta función miembro y pondremos ejemplos que nos muestren su utilidad.

Finalización

El método *finalize* se llama cuando va a ser [*liberada la memoria que ocupa el objeto*](#) por el recolector de basura (garbage collector). Normalmente, no es necesario redefinir este método en las clases, solamente en contados casos especiales. La forma en la que se redefine este método es el siguiente.

```
class CualquierClase{
    //..

    protected void finalize() throws Throwable{
        super.finalize();
        //código que libera recursos externos
    }
}
```

La primera sentencia que contenga la redefinición de *finalize* ha de ser una llamada a la función del mismo nombre de la clase base, y a continuación le añadimos cierta funcionalidad, habitualmente, la liberación de recursos, cerrar un archivo, etc.

La jerarquía de clases



[La herencia y el polimorfismo](#)

[La jerarquía de clases que describen las figuras planas](#)

[Uso de la jerarquía de clases](#)

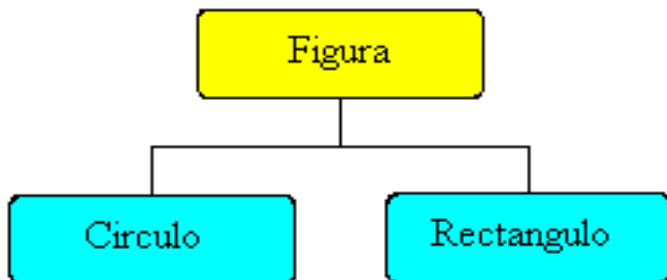
[Enlace dinámico](#)

[El polimorfismo en acción](#)

La jerarquía de clases que describen las figuras planas

 **figura:** [Figura.java](#), [FiguraApp.java](#)

Consideremos las figuras planas cerradas como el rectángulo, y el círculo. Tales figuras comparten características comunes como es la posición de la figura, de su centro, y el área de la figura, aunque el procedimiento para calcular dicha área sea completamente distinto. Podemos por tanto, diseñar una jerarquía de clases, tal que la clase base denominada *Figura*, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la figura



La clase *Figura* es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene área. Esto lo expresamos declarando *Figura* como una clase abstracta, declarando la función miembro *area* **abstract**.

Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta. Sin embargo, se pueden declarar variables de dichas clases.

En el juego del ajedrez podemos definir una clase base denominada *Pieza*, con las características comunes a todas las piezas, como es su posición en el tablero, y derivar de ella las características específicas de cada pieza particular. Así pues, la clase *Pieza* será una clase abstracta con una función **abstract** denominada *mover*, y cada tipo de pieza definirá dicha función de acuerdo a las reglas de su movimiento sobre el tablero.

- La clase *Figura*

La definición de la clase abstracta *Figura*, contiene la posición x e y de la figura particular, de su centro, y la función *area*, que se va a definir en las clases derivadas para calcular el área de cada figura en particular.

```
public abstract class Figura {
    protected int x;
    protected int y;
    public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}
```

- La clase *Rectangulo*

Las clases derivadas heredan los miembros dato x e y de la clase base, y definen la función *area*, declarada **abstract** en la clase base *Figura*, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada *Rectangulo*, tiene como datos, aparte de su posición (x , y) en el plano, sus dimensiones, es decir, su anchura *ancho* y altura *alto*.

```

class Rectangulo extends Figura{
    protected double ancho, alto;
    public Rectangulo(int x, int y, double ancho, double alto){
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public double area(){
        return ancho*alto;
    }
}

```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada **super**. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto x e y . Después inicializa sus miembros dato *ancho* y *alto*.

En la definición de la función *area*, se calcula el área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado

- La clase *Circulo*

```

class Circulo extends Figura{
    protected double radio;
    public Circulo(int x, int y, double radio){
        super(x,y);
        this.radio=radio;
    }
    public double area(){
        return Math.PI*radio*radio;
    }
}

```

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabra reservada **super**. Posteriormente, se inicializa el miembro dato *radio*, de la clase derivada *Circulo*.

En la definición de la función *area*, se calcula el área del círculo mediante la conocida fórmula $\pi*r^2$, o bien $\pi*r*r$. La [constante Math.PI](#) es una aproximación decimal del número irracional π .

Uso de la jerarquía de clases

Creamos un objeto *c* de la clase *Circulo* situado en el punto (0, 0) y de 5.5 unidades de radio. Calculamos y mostramos el valor de su área.

```
Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());
```

Creamos un objeto *r* de la clase *Rectangulo* situado en el punto (0, 0) y de dimensiones 5.5 de anchura y 2 unidades de largo. Calculamos y mostramos el valor de su área.

```
Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+r.area());
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por **new** al crear objetos de las clases derivadas en una variable *f* del tipo *Figura* (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+f.area());
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

Enlace dinámico

En el lenguaje C, los identificadores de la función están asociados siempre a direcciones físicas antes de la ejecución del programa, esto se conoce como enlace temprano o estático. Ahora bien, el lenguaje C++ y Java permiten decidir a que función llamar en tiempo de ejecución, esto se conoce como enlace tardío o dinámico. Vamos a ver un ejemplo de ello.

Podemos crear un array de la clase base *Figura* y guardar en sus elementos los valores devueltos por **new** al crear objetos de las clases derivadas.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
```

```
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
```

La sentencia

```
fig[i].area();
```

¿a qué función *area* llamará?. La respuesta será, según sea el índice *i*. Si *i* es cero, el primer elemento del array guarda una referencia a un objeto de la clase *Rectangulo*, luego llamará a la función miembro *area* de *Rectangulo*. Si *i* es uno, el segundo elemento del array guarda una referencia un objeto de la clase *Circulo*, luego llamará también a la función *area* de *Circulo*, y así sucesivamente. Pero podemos introducir el valor del índice *i*, a través del teclado, o seleccionando un control en un applet, en el momento en el que se ejecuta el programa. Luego, la decisión sobre qué función *area* se va a llamar se retrasa hasta el tiempo de ejecución.

El polimorfismo en acción

Supongamos que deseamos saber la figura que tiene mayor área independientemente de su forma. Primero, programamos una función que halle el mayor de varios números reales positivos.

```
double valorMayor(double[] x){
    double mayor=0.0;
    for (int i=0; i<x.length; i++)
        if(x[i]>mayor){
            mayor=x[i];
        }
    return mayor;
}
```

Ahora, la llamada a la función *valorMayor*

```
double numeros[]={3.2, 3.0, 5.4, 1.2};
System.out.println("El valor mayor es "+valorMayor(numeros));
```

La función *figuraMayor* que compara el área de figuras planas es semejante a la función *valorMayor* anteriormente definida, se le pasa el array de objetos de la clase base *Figura*. La función devuelve una referencia al objeto cuya área es la mayor.

```

static Figura figuraMayor(Figura[] figuras){
    Figura mFigura=null;
    double areaMayor=0.0;
    for(int i=0; i<figuras.length; i++){
        if(figuras[i].area()>areaMayor){
            areaMayor=figuras[i].area();
            mFigura=figuras[i];
        }
    }
    return mFigura;
}

```

La clave de la definición de la función está en las líneas

```

    if(figuras[i].area()>areaMayor){
        areaMayor=figuras[i].area();
        mFigura=figuras[i];
    }

```

En la primera línea, se llama a la versión correcta de la función *area* dependiendo de la referencia al tipo de objeto que guarda el elemento *figuras[i]* del array. En *areaMayor* se guarda el valor mayor de las áreas calculadas, y en *mFigura*, la figura cuya área es la mayor.

La principal ventaja de la definición de esta función estriba en que la función *figuraMayor* está definida en términos de variable *figuras* de la clase base *Figura*, por tanto, trabaja no solamente para una colección de círculos y rectángulos, sino también para cualquier otra figura derivada de la clase base *Figura*. Así si se deriva *Triangulo* de *Figura*, y se añade a la jerarquía de clases, la función *figuraMayor* podrá manejar objetos de dicha clase, sin modificar para nada el código de la misma.

Veamos ahora la llamada a la función *figuraMayor*

```

Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);

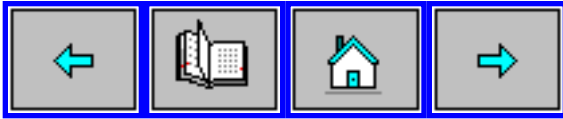
Figura fMayor=figuraMayor(fig);
System.out.println("El área mayor es "+fMayor.area());

```

Pasamos el array *fig* a la función *figuraMayor*, el valor que retorna lo guardamos en *fMayor*. Para conocer el valor del área, desde *fMayor* se llamará a la función miembro *area*. Se llamará a la versión correcta dependiendo de la referencia al tipo de objeto que guarde por *fMayor*. Si *fMayor* guarda una referencia a un objeto de la clase *Circulo*, llamará a la función *area* definida en dicha clase. Si *fMayor* guarda una referencia a un objeto de la clase *Rectangulo*, llamará a la función *area* definida en dicha clase, y así sucesivamente.

La combinación de herencia y enlace dinámico se denomina polimorfismo. El polimorfismo es, por tanto, la técnica que permite pasar un objeto de una clase derivada a funciones que conocen el objeto solamente por su clase base.

Ampliación de la jerarquía de clases



[La herencia y el polimorfismo](#)

[Añadiendo nuevas clases a la jerarquía](#)

[El polimorfismo en acción](#)

[El operador *instanceof*](#)

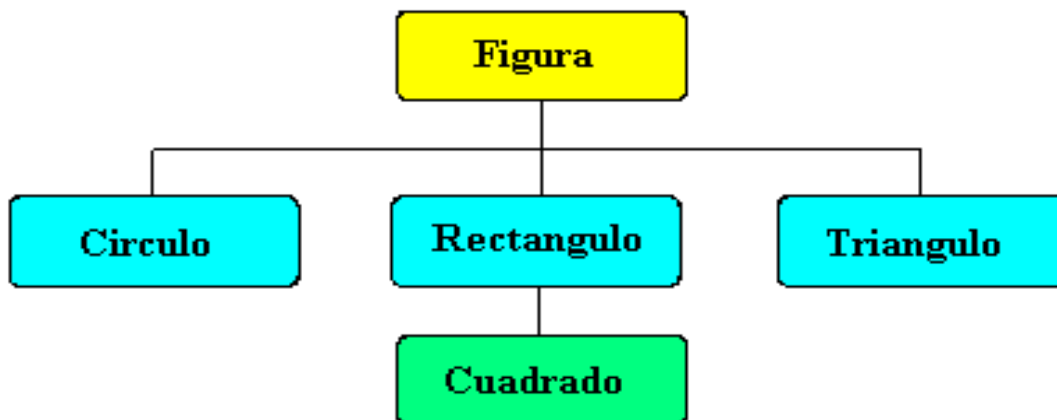
[Resumen](#)

Añadiendo nuevas clases a la jerarquía



figura1: [Figura.java](#), [FiguraApp1.java](#)

Ampliamos el árbol jerárquico de las clases que describen las figuras planas regulares, para acomodar a dos clases que describen las figuras planas, triángulo y cuadrado. La relación jerárquica se muestra en la figura.



- La clase *Cuadrado*

La clase *Cuadrado* es una clase especializada de *Rectangulo*, ya que un cuadrado tiene los lados iguales.

El constructor solamente precisa de tres argumentos los que corresponden a la posición de la figura y a la longitud del lado

```
class Cuadrado extends Rectangulo{
    public Cuadrado(int x, int y, double dimension){
        super(x, y, dimension, dimension);
    }
}
```

El constructor de la clase derivada llama al constructor de la clase base y le pasa la posición x e y de la figura, el ancho y alto que tienen el mismo valor. No es necesario redefinir una nueva función *area*. La clase *Cuadrado* hereda la función *area* definida en la clase *Rectangulo*.

- La clase *Triangulo*

La clase derivada *Triángulo*, tiene como datos, aparte de su posición (x, y) en el plano, la base y la altura del triángulo.

```
class Triangulo extends Figura{
    protected double base, altura;
    public Triangulo(int x, int y, double base, double altura){
        super(x, y);
        this.base=base;
        this.altura=altura;
    }
    public double area(){
        return base*altura/2;
    }
}
```

El constructor de la clase *Triangulo* llama al constructor de la clase *Figura*, le pasa las coordenadas x e y de su centro, y luego inicializa los miembros dato *base* y *altura*.

En la definición de la función *area*, se calcula el área del triángulo como producto de la *base* por la *altura* y dividido por dos.

El polimorfismo en acción

Veamos ahora la llamada a la [función *figuraMayor*](#). Primero, creamos un array del tipo *Figura*, guardando en sus elementos las direcciones devueltas por **new** al crear cada uno de los objetos.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 2.0);
fig[1]=new Circulo(0,0, 3.0);
fig[2]=new Cuadrado(0, 0, 5.0);
fig[3]=new Triangulo(0,0, 7.0, 12.0);

Figura fMayor=figuraMayor(fig);
System.out.println("El área mayor es "+fMayor.area());
```

Pasamos el array *fig* a la función *figuraMayor*, el valor que retorna lo guardamos en *fMayor*. Para conocer el valor del área, desde *fMayor* se llamará a la función miembro *area*. Se llamará a la versión correcta dependiendo de la referencia al tipo de objeto que guarda *fMayor*.

Si *fMayor* guarda una referencia a un objeto de la clase *Circulo*, llamará a la función *area* definida en dicha clase. Si *fMayor* guarda una referencia a un objeto de la clase *Triangulo*, llamará a la función *area* definida en dicha clase, y así sucesivamente.

El operador *instanceof*

El operador **instanceof** tiene dos operandos: un objeto en el lado izquierdo y una clase en el lado derecho. Esta expresión devuelve **true** o **false** dependiendo de que el objeto situado a la izquierda sea o no una instancia de la clase situada a la derecha o de alguna de sus clases derivadas.

Por ejemplo.

```
Rectangulo rect=new Rectangulo(0, 0, 5.0, 2.0);
rect instanceof String           //false
rect instanceof Rectangulo       //true
```

El objeto *rect* de la clase *Rectangulo* no es un objeto de la clase *String*. El objeto *rect* si es un objeto de la clase *Rectangulo*.

Veamos la relación entre *rect* y las clases de la jerarquía

```
rect instanceof Figura           //true
rect instanceof Cuadrado         //false
```

rect es un objeto de la clase base *Figura* pero no es un objeto de la clase derivada *Cuadrado*

Resumen

La herencia es la propiedad que permite la creación de nuevas clases a partir de clases ya existentes. La clase derivada hereda los datos y las funciones miembro de la clase base, y puede redefinir algunas de las funciones miembro o definir otras nuevas, para ampliar la funcionalidad que ha recibido de la clase base.

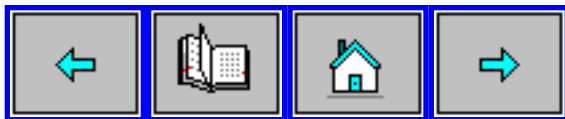
Para crear un objeto de la clase derivada se llama primero al constructor de la clase base mediante la palabra reservada **super**. Luego, se inicializa los miembros dato de dicha clase derivada

El polimorfismo se implementa por medio de las funciones abstractas, en las clases derivadas se declara y se define una función que tiene el mismo nombre, el mismo número de parámetros y del mismo tipo que en la clase base, pero que da lugar a un comportamiento distinto, específico de los objetos de la clase derivada.

Enlace dinámico significa que la decisión sobre la función a llamar se demora hasta el tiempo de ejecución del programa.

No se pueden crear objetos de una clase abstracta pero si se pueden declarar referencias en las que guardamos el valor devuelto por **new** al crear objetos de las clases derivadas. Esta peculiaridad nos permite pasar un objeto de una clase derivada a una función que conoce el objeto solamente por su clase base. De este modo podemos ampliar la jerarquía de clases sin modificar el código de las funciones que manipulan los objetos de las clases de la jerarquía.

Clases y métodos finales



[La herencia y el polimorfismo](#)

[Clases finales](#)

[Métodos finales](#)

Clases finales

Se puede declarar una clase como **final**, cuando no nos interesa crear clases derivadas de dicha clase. La clase [Cuadrado](#) se puede declarar como **final**, ya que no se espera que ningún programador necesite crear clases derivadas de *Cuadrado*.

```
final class Cuadrado extends Rectangulo{  
    public Cuadrado(int x, int y, double dimension){  
        super(x, y, dimension, dimension);  
    }  
}
```

Uno de los mecanismos que tienen los hackers para dañar o para obtener información privada en los sistemas es la de crear una clase derivada y sustituir dicha clase por la original. La clase derivada actúa exactamente igual que la original pero también puede hacer otras cosas, normalmente dañinas. Para prevenir los posibles daños, se declara la clase como **final**, impidiendo a cualquier programador la creación de clases derivadas de ésta. Por ejemplo, la clase *String* que es una de las más importantes en la programación en lenguaje Java, está declarada como **final**. El lenguaje Java garantiza que siempre que se utilice un string, es un objeto de la clase *String* que se encuentra en el paquete *java.lang.String*, y no cualquier otro string.

Métodos finales

Como se ha comentado al introducir la herencia, una de las formas de aprovechar el código existente, es la de crear una clase derivada y redefinir algunos de los métodos de la clase base.

```
class Base{
//...
    final public void funcionFinal(){
//...
    }
    public void dibujar(Graphics g){
    }
}

class Derivada{
//...
    public void dibujar(Graphics g){
//dibujar algunas figuras
    }
}
```

La clase *Base* define una función miembro pública *dibujar*, que no dibuja nada en el contexto gráfico *g*. La clase *Derivada* redefine la función miembro *dibujar*, para dibujar algunas figuras en el contexto gráfico *g*. La función que se redefine tiene que tener la misma declaración en la clase *Base* y en la clase *Derivada*.

Para evitar que las clase derivadas redefinan una función miembro de una clase base, se le antepone la palabra clave **final**. La función miembro *funcionFinal* de la clase *Base* no se puede redefinir en la clase *Derivada*, pero si se puede redefinir la función miembro *dibujar*.

Interfaces



[La herencia y el polimorfismo](#)

[¿Qué es un interface?](#)

[Diferencias entre un interface y una clase abstracta](#)

[Los interfaces y el polimorfismo](#)

¿Qué es un interface?

Un interface es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes.

Runnable es un ejemplo de interface en el cual se declara, pero no se implementa, una función miembro *run*.

```
public interface Runnable {  
    public abstract void run();  
}
```

Las clases que implementen (**implements**) el interface *Runnable* han de definir obligatoriamente la función *run*.

```
class Animacion implements Runnable{  
    //..  
    public void run(){  
        //define la función run  
    }  
}
```

El papel del interface es el de describir algunas de las características de una clase. Por ejemplo, el hecho de que una persona sea un futbolista no define su personalidad completa, pero hace que tenga ciertas características que las distinguen de otras.

Clases que no están relacionadas pueden implementar el interface *Runnable*, por ejemplo, una clase que describa una animación, y también puede implementar el interface *Runnable* una clase que realice un cálculo intensivo.

Diferencias entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.

Ahora bien, la diferencia es mucho más profunda. Imaginemos que *Runnable* fuese una clase abstracta. Un applet descrito por la clase *MiApplet* que moviese una figura por su área de trabajo, derivaría a la vez de la clase base *Applet* (que describe la funcionalidad mínima de un applet que se ejecuta en un navegador) y de la clase *Runnable*. Pero el lenguaje Java no tiene herencia múltiple.

En el lenguaje Java la clase *MiApplet* deriva de la clase base *Applet* e implementa el interface *Runnable*

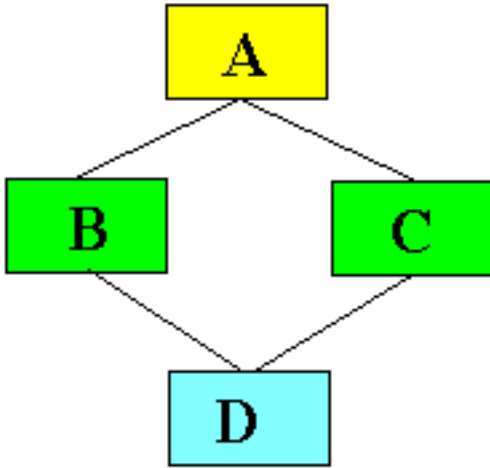
```
class MiApplet extends Applet implements Runnable{
//...
//define la función run del interface
    public void run(){
        //...
    }
//redefine paint de la clase base Applet
    public void paint(Graphics g){
        //...
    }
//define otras funciones miembro
}
```

Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**.

El lenguaje Java no fuerza por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares.

Los interfaces y el polimorfismo

En el lenguaje C++, es posible la herencia múltiple, pero este tipo de herencia presenta dificultades. Por ejemplo, cuando dos clases B y C derivan de una clase base A, y a su vez una clase D deriva de B y C. Este problema es conocido con el nombre de diamante.



En el lenguaje Java solamente existe la herencia simple, pero las clases pueden implementar interfaces. Vamos a ver en este apartado que la importancia de los interfaces no estriba en resolver los problemas inherentes a la herencia múltiple sin forzar relaciones jerárquicas, sino es el de incrementar el polimorfismo del lenguaje más allá del que proporciona la herencia simple.

Para explicar este aspecto importante y novedoso del lenguaje Java adaptaremos los ejemplos que aparecen en el artículo del Bill Venners "Designing with interfaces" publicado en Java World (www.javaWorld.com) en Diciembre de 1998. Comparemos la herencia simple mediante un ejemplo similar al de [la jerarquía de las figuras planas](#), con los interfaces.

Herencia simple



polimorfismo: [Animal.java](#), [PoliApp.java](#)

Creamos una clase abstracta denominada *Animal* de la cual deriva las clases *Gato* y *Perro*. Ambas clases redefinen la función *habla* declarada abstracta en la clase base *Animal*.

```

public abstract class Animal {
    public abstract void habla();
}

class Perro extends Animal{
    public void habla(){
        System.out.println("¡Guau!");
    }
}

class Gato extends Animal{
    public void habla(){
        System.out.println("¡Miau!");
    }
}

```

El polimorfismo nos permite pasar la referencia a un objeto de la clase *Gato* a una función *hazleHablar* que conoce al objeto por su clase base *Animal*

```

public class PoliApp {
    public static void main(String[] args) {
        Gato gato=new Gato();
        hazleHablar(gato);
    }

    static void hazleHablar(Animal sujeto){
        sujeto.habla();
    }
}

```

El compilador no sabe exactamente que objeto se le pasará a la función *hazleHablar* en el momento de la ejecución del programa. Si se pasa un objeto de la clase *Gato* se imprimirá ¡Miau!, si se pasa un objeto de la clase *Perro* se imprimirá ¡Guau!. El compilador solamente sabe que se le pasará un objeto de alguna clase derivada de *Animal*. Por tanto, el compilador no sabe que función *habla* será llamada en el momento de la ejecución del programa.

El polimorfismo nos ayuda a hacer el programa más flexible, por que en el futuro podemos añadir nuevas clases derivadas de *Animal*, sin que cambie para nada el método *hazleHablar*. Como ejercicio, se sugiere al lector añadir la clase *Pajaro* a la jerarquía, y pasar un objeto de dicha clase a la función *hazleHablar* para que se imprima ¡pio, pio, pio ..!.

Interfaces



polimorfismo1: [Parlanchin.java](#), [Animal.java](#), [Reloj.java](#), [PoliApp.java](#)

Vamos a crear un interface denominado *Parlanchin* que contenga la declaración de una función denominada *habla*.

```
public interface Parlanchin {
    public abstract void habla();
}
```

Hacemos que la jerrarquía de clases que deriva de *Animal* implemente el interface *Parlanchin*

```
public abstract class Animal implements Parlanchin{
    public abstract void habla();
}

class Perro extends Animal{
    public void habla(){
        System.out.println("¡Guau!");
    }
}

class Gato extends Animal{
    public void habla(){
        System.out.println("¡Miau!");
    }
}
```

Ahora veamos otra jerarquía de clases completamente distinta, la que deriva de la clase base *Reloj*. Una de las clases de dicha jerarquía *Cucu* implementa el interface *Parlanchin* y por tanto, debe de definir obligatoriamente la función *habla* declarada en dicho interface.

```
public abstract class Reloj {
}

class Cucu extends Reloj implements Parlanchin{
    public void habla(){
        System.out.println("¡Cucu, cucu, ..!");
    }
}
```

Definamos la función *hazleHablar* de modo que conozca al objeto que se le pasa no por una clase base, sino por el interface *Parlanchin*. A dicha función le podemos pasar cualquier objeto que implemente el interface *Parlanchin*, este o no en la misma jerarquía de clases.

```
public class PoliApp {

    public static void main(String[] args) {
        Gato gato=new Gato();
        hazleHablar(gato);
        Cucu cucu=new Cucu();
        hazleHablar(cucu);
    }

    static void hazleHablar(Parlanchin sujeto){
        sujeto.habla();
    }
}
```

Al ejecutar el programa, veremos que se imprime en la consola ¡Miau!, por que a la función *hazleHablar* se le pasa un objeto de la clase *Gato*, y después ¡Cucu, cucu, ..! por que a la función *hazleHablar* se le pasa un objeto de la clase *Cucu*.

Si solamente hubiese herencia simple, *Cucu* tendría que derivar de la clase *Animal* (lo que no es lógico) o bien no se podría pasar a la función *hazleHablar*. Con interfaces, cualquier clase en cualquier familia puede implementar el interface *Parlanchin*, y se podrá pasar un objeto de dicha clase a la función *hazleHablar*. Esta es la razón por la cual los interfaces proporcionan más polimorfismo que el que se puede obtener de una simple jerarquía de clases.

Las excepciones estándar



Las excepciones

Las excepciones

Captura de las excepciones

Manejando varias excepciones

Los programadores de cualquier lenguaje se esfuerzan por escribir programas libres de errores, sin embargo, es muy difícil que los programas reales se vean libres de ellos. En Java las situaciones que pueden provocar un fallo en el programa se denominan excepciones.

Java lanza una excepción en respuesta a una situación poco usual. El programador también puede lanzar sus propias excepciones. Las excepciones en Java son objetos de clases derivadas de la clase base *Exception*. Existen también los errores internos que son objetos de la clase *Error* que no estudiaremos. Ambas clases *Error* y *Exception* son clases derivadas de la clase base *Throwable*.

Existe toda una jerarquía de clases derivada de la clase base *Exception*. Estas clases derivadas se ubican en dos grupos principales:

Las excepciones en tiempo de ejecución ocurren cuando el programador no ha tenido cuidado al escribir su código. Por ejemplo, cuando se sobrepasa la dimensión de un array se lanza una excepción *ArrayIndexOutOfBoundsException*. Cuando se hace uso de una referencia a un objeto que no ha sido creado se lanza la excepción *NullPointerException*. Estas excepciones le indican al programador que tipos de fallos tiene el programa y que debe arreglarlo antes de proseguir.

El segundo grupo de excepciones, es el más interesante, ya que indican que ha sucedido algo inesperado o fuera de control.

Las excepciones

En la página dedicada al estudio de la clase *String*, mencionamos una función que convierte un [string en un número](#). Esta función es muy usada cuando creamos applets. Introducimos el número en un control de edición, se obtiene el texto y se guarda en un string. Luego, se convierte el string en número entero mediante la función estática *Integer.parseInt*, y finalmente, usamos dicho número.

```
String str=" 12 ";  
int numero=Integer.parseInt(str);
```

Si se introducen caracteres no numéricos, o no se quitan los espacios en blanco al principio y al final del string, [mediante la función *trim*](#), se lanza una excepción *NumberFormatException*.

```
AppAccelerator(tm) 1.1.034 for Java (JDK 1.1), x86 version.
Copyright (c) 1998 Borland International. All Rights Reserved.
```

```
java.lang.NumberFormatException: 12
    at java.lang.Integer.parseInt(Compiled Code)
    at java.lang.Integer.parseInt(Integer.java:390)
    at excepcion.ExcepcionApp.main(ExcepcionApp.java:8)
```

Para ver este texto, [se selecciona el elemento del menú del IDE *Run/Parematers*](#). En el cuadro de diálogo que aparece activar el botón de radio *Set run output to Execution Log*. Luego, se corre la aplicación *Run/Run*. Se selecciona el elemento del menú *View/Execution Log*, para que se muestre en una ventana la salida del programa. Para restaurar la salida a la consola, la ventana DOS, se selecciona de nuevo *Run/Parematers*. En el cuadro de diálogo que aparece, se activa el botón de radio *Set run output to Console window*.

El mensaje que aparece en la ventana nos indica el tipo de excepción *NumberFormatException*, la función que la ha lanzado *Integer.parseInt*, que se llama dentro de *main*.

Objeto no inicializado

Habitualmente, cuando llamamos desde un objeto no inicializado, a una función miembro.

```
public static void main(String[] args) {
    String str;
    str.length();
    //...
}
```

El compilador se queja con el siguiente mensaje "variable str might not have been initilized". En otras ocasiones, se lanza una excepción del tipo *NulPointerException*. Fijarse que en la porción de código que sigue, *grafico* es una variable de instancia que es inicializada por defecto a **null**.

```
class MiCanvas....{
    Grafico grafico;
    public void paint(...){
        grafico.dibuja();
        //...
    }
    //...
}
```

Como vemos en la porción de código, si al llamarse a la función *paint*, el objeto *grafico* no ha sido inicializado con el valor devuelto por **new** al crear un objeto de la clase *Grafico* o de alguna de sus clases derivadas, se lanza la excepción *NullPointerException* apareciendo en la consola el siguiente texto.

```
Exception occurred during event dispatching:
java.lang.NullPointerException
    at grafico1.MiCanvas.paint(MiCanvas.java:43)
    at sun.awt.windows.WComponentPeer.handleEvent(Compiled Code)
    at java.awt.Component.dispatchEventImpl(Compiled Code)
```

```
at java.awt.Component.dispatchEvent(Compiled Code)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:82)
```

Entrada/salida

En otras situaciones el mensaje de error aparece en el momento en el que se compila el programa. Así, cuando intentamos leer un carácter del teclado, llamamos a la la función

```
System.in.read();
```

Cuando compilamos el programa, nos aparece un mensaje de error que no nos deja proseguir.

```
unreported exception: java.io.IOException; must be caught or declared to be thrown
```

Captura de las excepciones

Empecemos por solucionar el error que se produce en el programa durante la compilación. Tal como indica el mensaje que genera el compilador, se ha de poner la sentencia *System.in.read()*; en un bloque **try...catch**, del siguiente modo.

```
try {
    System.in.read();
}catch (IOException ex) { }
```

Para solucionar el error que se produce en el programa durante su ejecución, se debe poner la llamada a *Integer.parseInt* en el siguiente bloque **try...catch**.

```
String str="  12 ";
int numero;
try{
    numero=Integer.parseInt(str);
}catch(NumberFormatException ex){
    System.out.println("No es un número");
}
```

En el caso de que el string *str* contenga caracteres no numéricos como es éste el caso, el número 12 está acompañado de espacios en blanco, se produce una excepción del tipo *NumberFormatException* que es capturada y se imprime el mensaje "No es un número".

En vez de un mensaje propio se puede imprimir el objeto *ex* de la clase *NumberFormatException*

```
try{
    //...
}catch(NumberFormatException ex){
    System.out.println(ex);
}
```

}

La clase base *Throwable* de todas las clases que describen las excepciones, redefine la función *toString*, que devuelve el nombre de la clase que describe la excepción acompañado del mensaje asociado, que en este caso es el propio string *str*.

```
java.lang.NumberFormatException: 12
```

Podemos extraer dicho mensaje mediante la función miembro *getMessage*, del siguiente modo

```
try{
    //...
}catch(NumberFormatException ex){
    System.out.println(ex.getMessage());
}
```

Manejando varias excepciones



excepcion: [ExcepcionApp.java](#)

Vamos a crear un programa que divida dos números. Supongamos que los números se introducen en dos controles de edición. Se obtiene el texto de cada uno de los controles de edición que se guardan en dos strings. En esta situación se pueden producir dos excepciones *NumberFormatException*, si se introducen caracteres no numéricos y *ArithmeticException* si se divide entre cero.

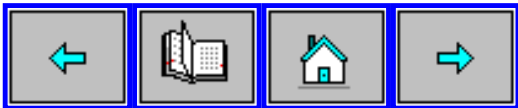
```
public class ExcepcionApp {
    public static void main(String[] args) {
        String str1="12";
        String str2="0";
        String respuesta;
        int numerador, denominador, cociente;
        try{
            numerador=Integer.parseInt(str1);
            denominador=Integer.parseInt(str2);
            cociente=numerador/denominador;
            respuesta=String.valueOf(cociente);
        }catch(NumberFormatException ex){
            respuesta="Se han introducido caracteres no numéricos";
        }catch(ArithmeticException ex){
            respuesta="División entre cero";
        }
        System.out.println(respuesta);
    }
}
```

Como vemos las sentencias susceptibles de lanzar una excepción se sitúan en un bloque **try...catch**. Si el denominador es cero, se produce una excepción de la clase *ArithmeticException* en la expresión que halla el cociente, que es inmediatamente capturada en el bloque **catch** que maneja dicha excepción, ejecutándose las sentencias que hay en dicho bloque. En este caso se guarda en el string *respuesta* el texto "División entre cero".

Hay veces en las que se desea estar seguro de que un bloque de código se ejecute se produzcan o no excepciones. Se puede hacer esto añadiendo un bloque **finally** después del último **catch**. Esto es importante cuando accedemos a archivos, para asegurar que se cerrará siempre un archivo se produzca o no un error en el proceso de lectura/escritura.

```
try{
    //Este código puede generar una excepción
}catch(Exception ex){
    //Este código se ejecuta cuando se produce una excepción
}finally{
    //Este código se ejecuta se produzca o no una excepción
}
```

Las excepciones propias



Las excepciones

[La clase que describe la excepción](#)

[El método que puede lanzar una excepción](#)

[Captura de las excepciones](#)

[Una función que puede lanzar varias excepciones](#)

El lenguaje Java proporciona las clases que manejan casi cualquier tipo de excepción. Sin embargo, podemos imaginar situaciones en la que producen excepciones que no están dentro del lenguaje Java. Siguiendo el ejemplo de la página anterior estudiaremos una situación en la que el usuario introduce un valor fuera de un determinado intervalo, el programa lanza una excepción, que vamos a llamar *ExcepcionIntervalo*.

La clase que describe la excepción



excepcion3: [ExcepcionIntervalo.java](#), [ExcepcionApp3.java](#)

Para crear y lanzar una excepción propia tenemos que definir la clase *ExcepcionIntervalo* [derivada de la clase base Exception](#).

```
public class ExcepcionIntervalo extends Exception {  
    public ExcepcionIntervalo(String msg) {  
        super(msg);  
    }  
}
```

La definición de la clase es muy simple. Se le pasa un string *msg*, que contiene un mensaje, en el único parámetro que tiene el constructor de la clase derivada y éste se lo pasa a la clase base mediante **super**.

El método que puede lanzar una excepción

La función miembro que lanza una excepción tiene la declaración habitual que cualquier otro método pero se le añade a continuación la palabra reservada **throws** seguido de la excepción o excepciones que puede lanzar.

```
static void rango(int num, int den)throws ExcepcionIntervalo{
    if((num>100)|| (den<-5)){
        throw new ExcepcionIntervalo("Números fuera del intervalo");
    }
}
```

Cuando el numerador es mayor que 100 y el denominador es menor que 5 se lanza **throw** una excepción, un objeto de la clase *ExcepcionIntervalo*. Dicho objeto se crea llamando al constructor de dicha clase y pasándole un string que contiene el mensaje "Números fuera del intervalo".

Captura de las excepciones

Al programa estudiado en la página anterior, añadimos la llamada a la función *rango* que verifica si los números están dentro del intervalo dado, y el bloque **catch** que captura la excepción que puede lanzar dicha función si los números no están en el intervalo especificado.

```
public class ExcepcionApp3 {
    public static void main(String[] args) {
        String str1="120";
        String str2="3";
        String respuesta;
        int numerador, denominador, cociente;
        try{
            numerador=Integer.parseInt(str1);
            denominador=Integer.parseInt(str2);
            rango(numerador, denominador);
            cociente=numerador/denominador;
            respuesta=String.valueOf(cociente);
        }catch(NumberFormatException ex){
            respuesta="Se han introducido caracteres no numéricos";
        }catch(ArithmeticException ex){
            respuesta="División entre cero";
        }catch(ExcepcionIntervalo ex){
            respuesta=ex.getMessage();
        }
        System.out.println(respuesta);
    }
}
```

```

        static void rango(int num, int den)throws ExcepcionIntervalo{
            if((num>100)|| (den<-5)){
                throw new ExcepcionIntervalo("Números fuera de rango");
            }
        }
    }
}

```

Como vemos el numerador que vale 120 tiene un valor fuera del intervalo especificado en la función *rango*, por lo que se lanza una excepción cuando se llega a la llamada a dicha función en el bloque **try**. Dicha excepción es capturada por el bloque **catch** correspondiente a dicha excepción, y se ejecutan las sentencias de dicho bloque. En concreto, se obtiene mediante *getMessage* el texto del mensaje que guarda el objeto *ex* de la clase *ExcepcionIntervalo*.

El ciclo de vida de una excepción se puede resumir del siguiente modo:

1. Se coloca la llamada a la función susceptible de producir una excepción en un bloque **try...catch**
2. En dicha función se crea mediante **new** un objeto de la clase *Exception* o derivada de ésta
3. Se lanza mediante **throw** el objeto recién creado
4. Se captura en el correspondiente bloque **catch**
5. En este bloque se notifica al usuario esta eventualidad imprimiendo el mensaje asociado a dicha excepción, o realizando una tarea específica.

Una función que puede lanzar varias excepciones



excepcion4: [ExcepcionIntervalo.java](#), [ExcepcionApp4.java](#)

Hay otra alternativa para el ejercicio anterior, que es la de definir una función denominada *calcular*, que devuelva el cociente entre el numerador y el denominador, cuando se le pasa los strings obtenidos de los respectivos controles de edición. La función *calcular*, convierte los strings en números enteros, verifica el rango, calcula y devuelve el cociente entre el numerador y el denominador,

```

public class ExcepcionApp4 {
    public static void main(String[] args) {
        String str1="20";
        String str2="2";
        String respuesta;
        int numerador, denominador, cociente;
        try{
            cociente=calcular(str1, str2);
            respuesta=String.valueOf(cociente);
        }catch(NumberFormatException ex){
            respuesta="Se han introducido caracteres no numéricos";
        }catch(ArithmeticException ex){
            respuesta="División entre cero";
        }catch(ExcepcionIntervalo ex){
            respuesta=ex.getMessage();
        }
        System.out.println(respuesta);
    }
    static int calcular(String str1, String str2)throws ExcepcionIntervalo,
        NumberFormatException, ArithmeticException{
        int num=Integer.parseInt(str1);
        int den=Integer.parseInt(str2);
        if((num>100)|| (den<-5)){
            throw new ExcepcionIntervalo("Números fuera del intervalo");
        }
        return (num/den);
    }
}

```

Vemos que la función *calcular* puede lanzar, **throws**, tres tipos de excepciones. En el cuerpo de la función se crea, **new**, y se lanza, **throw**, explícitamente un objeto de la clase *ExcepcionIntervalo*, definida por el usuario, e implícitamente se crea y se lanza objetos de las clases *NumberFormatException* y *ArithmeticException* definidas en el lenguaje Java.

La sentencia que llama a la función *calcular* dentro del bloque **try** puede producir alguna de las tres excepciones que es capturada por el correspondiente bloque **catch**.

Podemos simplificar algo el código ahorrándonos la variable temporal *cociente*, escribiendo en vez de las dos sentencias

```

cociente=calcular(str1, str2);
respuesta=String.valueOf(cociente);

```

Una única sentencia

```
respuesta=String.valueOf(calcular(str1, str2));
```

Paso por valor



Pasando datos a una función

Pasando datos de tipo básico

Pasando objetos

En el lenguaje C++ hay tres formas de pasar datos a una función: por valor, por dirección y por referencia.

En el lenguaje Java, solamente se pasa por valor, lo que significa que se efectúa una copia local de las entidades que se están pasando. Vamos a explicar mediante ejemplos, el significado de la expresión pasar por valor:

- datos del tipo básico: **int**, **double**, **char**, etc.
- objetos de una determinada clase

Pasando datos de tipo básico

 valor: [ValorApp.java](#)

Sea la función

```
void funcion(int x){  
    x=5;  
    System.out.println("dentro de la función: a="+x);  
}
```

Sea la variable *a* que toma inicialmente el valor de 3. ¿Cuál será el valor de *a* después de la llamada a la función *funcion*?

```
int a=3;
funcion(a);
System.out.println("después de la llamada: a="+a);
```

En primer lugar, recordaremos que la función denominada *funcion* tiene un único parámetro x , cuyo alcance es desde la llave de apertura hasta la llave de cierre de la función. La variable x deja de existir una vez que la función retorna.

Vamos a ver en este ejemplo el significado de "paso por valor". La variable a toma el valor inicial de 3. Cuando se llama a la función se pasa el valor de a en su único argumento, el valor de a se copia en el parámetro x , la variable x toma el valor de 3. En el curso de la llamada a la función, el valor de x cambia a 5, pero cuando la función retorna, la variable x ha dejado de existir. La variable a no se ha modificado en el curso de la llamada a la función, y sigue valiendo 3.

Durante el curso de la llamada a la función *funcion*, existe la variable a y su copia x , pero son dos variables distintas, aunque inicialmente guarden el mismo valor.

```
public class ValorApp {
    public static void main(String[] args) {
        int a=3;
        System.out.println("antes de la llamada: a="+a);
        funcion(a);
        System.out.println("después de la llamada: a="+a);
    }

    public static void funcion(int x){
        x=5;
        System.out.println("dentro de la función: a="+x);
    }
}
```

Pasando objetos



valor2: [Entero.java](#), [ValorApp2.java](#)

Creamos una clase *Entero* muy sencilla que tiene como miembro dato un número entero *valor*, y un constructor que inicializa dicho miembro público al crearse un objeto de la clase *Entero*.

```
public class Entero {
    public int valor;
    public Entero(int valor){
        this.valor=valor;
    }
}
```

El valor devuelto por **new** al crear un objeto es una referencia a un objeto en memoria, que hemos denominado objeto. Creamos un objeto *aInt* de la clase *Entero* para guardar el número tres.

```
Entero aInt=new Entero(3);
funcion(aInt);
```

El valor devuelto por **new** lo guardamos en *aInt*, y se lo pasamos a la función denominada *funcion*.

```
void funcion(Entero xInt){
    xInt.valor=5;
}
```

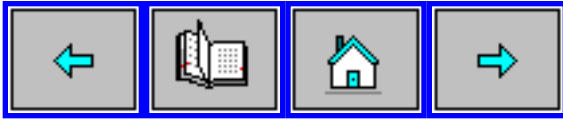
Dentro de la función denominada *funcion* disponemos en *xInt* de la referencia al objeto *aInt*. Como el argumento *aInt* y el parámetro *xInt* guardan la referencia al mismo objeto, es posible modificar dicho objeto en el curso de la llamada a la función, accediendo a sus miembros públicos. Desde *xInt* accedemos al miembro público *valor* para modificarlo. Cuando la función retorna, el objeto *aInt* habrá modificado su estado.

```
public class ValorApp2 {
    public static void main(String[] args) {
        Entero aInt=new Entero(3);
        System.out.println("Antes de llamar a la función");
        System.out.println("objeto.miembro "+aInt.valor);
        funcion(aInt);
        System.out.println("Después de llamar a la función");
        System.out.println("objeto.miembro "+aInt.valor);
    }

    public static void funcion(Entero xInt){
        xInt.valor=5;
    }
}
```

Estos dos ejemplos, nos ponen de manifiesto el significado de la frase "pasar por valor un dato a una función", y el distinto comportamiento de los tipos básicos de datos, que no se pueden modificar en el curso de la llamada a la función, de los objetos que si se pueden modificar. La referencia a un objeto se pasa por valor a la función. Dentro de la función, desde esta referencia podemos acceder a los miembros públicos de dicho objeto para modificar su estado. Cuando la función retorna el objeto estará modificado.

Duplicación de objetos (objetos)



Pasando datos a una función

El interface *Cloneable*

Duplicación de un objeto simple

Duplicación de un objeto compuesto

En algunas situaciones deseamos que un objeto que se pasa a una función, o bien, que un objeto que llama a una función miembro no se modifiquen en el curso de la llamada. Por ejemplo, un objeto de la [clase *Lista*](#) al llamar a la función miembro *ordenar* modifica la posición de los datos en el array. Podríamos estar interesados en mantener la misma secuencia original no ordenada de datos.

Al hallar el determinante de una matriz, efectuamos una serie de transformaciones sobre la matriz original que la convierte en una matriz triangular. Prodríamos estar interesados en mantener la matriz original para realizar otras operaciones.

En todos estos casos, puede ser muy útil para el programador realizar una copia del objeto original y realizar las transformaciones en la copia dejando intacto el original.



clonico: [Punto.java](#), [Rectangulo.java](#), [ClonicoApp.java](#)

El interface *Cloneable*

Un [interface](#) como hemos estudiado declara un conjunto de funciones, pero sin implementarlas. El interface *Cloneable* es muy simple ya que no define ninguna función.

```
public interface Cloneable {
}
```

Una clase que implemente este interface le indica al método *clone* de la [clase base *Object*](#) que puede hacer una copia miembro a miembro de las instancias de dicha clase. Si una clase no implementa esta interface, e intenta hacer una duplicación del objeto a través de la llamada al método *clone* de la clase base *Object*, da como resultado una excepción del tipo *CloneNotSupportedException*.

Duplicación de un objeto simple

La [clase base *Object*](#) de todas las clases en el lenguaje Java, tiene una función miembro denominada *clone*, que se redefine en la clase derivada para realizar una duplicación de un objeto de dicha clase.

Sea la [clase *Punto*](#) ya estudiada en páginas anteriores. Para hacer una copia de un objeto de esta clase, se ha de agregar a la misma el siguiente código:

- se ha de implementar el interface *Cloneable*
- se ha de redefinir la función miembro *clone* de la clase base *Object*

```
public class Punto implements Cloneable{
    private int x;
    private int y;
    //constructores ...

    public Object clone(){
        Object obj=null;
        try{
            obj=super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        return obj;
    }
    //otras funciones miembro
}
```

En la redefinición de *clone*, se llama a la versión *clone* de la clase base desde **super**. Esta llamada se ha de hacer forzosamente dentro de un bloque **try... catch**, para [capturar la excepción](#) *CloneNotSuportedException* que nunca se producirá si la clase implementa el interface *Cloneable*. Como vemos la llamada a la versión *clone* de la clase base devuelve un objeto de la clase base *Object*, que es a su vez devuelto por la versión *clone* de la clase derivada.

Para crear un objeto *pCopia* que es una copia de otro objeto *punto* se escribe.

```
Punto punto=new Punto(20, 30);
Punto pCopia=(Punto)punto.clone();
```

La promoción (casting) es necesaria ya que *clone* devuelve un objeto de la clase base *Object* que ha de ser promocionado a la clase *Punto*.

Si hemos redefinido en la clase *Punto* la función miembro *toString* de la clase base *Object*, podemos comprobar que los objetos *punto* y *pCopia* guardan los mismos valores en sus miembros dato.

```
System.out.println("punto "+ punto);
System.out.println("copia "+ pCopia);
```

```
public class Punto implements Cloneable{
    private int x;
    private int y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Punto() {
        x=0;
        y=0;
    }
    public Object clone(){
        Object obj=null;
        try{
            obj=super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        return obj;
    }
    public void trasladar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
    public String toString(){
        String texto="origen: (" +x+" , "+y+" )";
        return texto;
    }
}
```

}

Duplicación de un objeto compuesto

Volvamos de nuevo sobre la [clase *Rectangulo*](#) estudiada en páginas anteriores. Los miembros dato de la clase *Rectangulo* son un objeto de la clase *Punto*, el *origen*, y las dimensiones: *ancho* y *alto* del rectángulo.

Veamos ahora el código que tenemos que agregar a la clase *Rectangulo* para que se puedan efectuar copias de los objetos de dicha clase.

```
public class Rectangulo implements Cloneable{
    private int ancho ;
    private int alto ;
    private Punto origen;
//los constructores
    public Object clone(){
        Rectangulo obj=null;
        try{
            obj=(Rectangulo)super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        obj.origen=(Punto)obj.origen.clone();
        return obj;
    }
//otras funciones miembro
}
```

Un objeto de la clase *Rectangulo* contiene un subobjeto de la clase *Punto*. En la redefinición de la función miembro *clone* de la clase *Rectangulo* se ha de efectuar una duplicación de dicho subobjeto llamando a la versión *clone* definida en la clase *Punto*.

Recuérdese que la llamada a *clone* siempre devuelve un objeto de la clase base *Object* que ha de ser promocionado (casting) a la clase derivada adecuada.

Veamos ahora, como se crea un objeto de la clase *Rectangulo* y se duplica. Para crear un objeto *rCopia* que es una copia del objeto *rect* de la clase *Rectangulo*, se escribe.

```
Rectangulo rect=new Rectangulo(new Punto(0, 0), 4, 5);
Rectangulo rCopia=(Rectangulo)rect.clone();
```

Si hemos redefinido en la clase *Rectangulo* la función miembro *toString* de la clase base *Object*, podemos comprobar que los objetos *rect* y *rCopia* guardan los mismos valores en sus miembros dato.

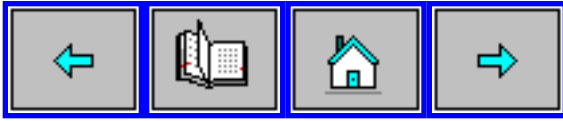
```
System.out.println("rectángulo "+ rect);
System.out.println("copia "+ rCopia);
```

```
public class Rectangulo implements Cloneable{
    private int ancho ;
    private int alto ;
    private Punto origen;

    public Rectangulo() {
        origen = new Punto(0, 0);
        ancho=0;
        alto=0;
    }
    public Rectangulo(Punto p) {
        this(p, 0, 0);
    }
    public Rectangulo(int w, int h) {
        this(new Punto(0, 0), w, h);
    }
    public Rectangulo(Punto p, int w, int h) {
        origen = p;
        ancho = w;
        alto = h;
    }
    public Object clone(){
        Rectangulo obj=null;
        try{
            obj=(Rectangulo)super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        obj.origen=(Punto)obj.origen.clone();
        return obj;
    }
    public void mover(int dx, int dy) {
        origen.trasladar(dx, dy);
    }
}
```

```
}  
public int area() {  
    return ancho * alto;  
}  
public String toString(){  
    String texto=origen+" ancho: "+ancho+" alto: "+alto;  
    return texto;  
}  
}
```

Duplicación de objetos (arrays)



[Pasando datos a una función](#)

[Duplicación de arrays unidimensionales](#)

[Duplicación de arrays bidimensionales](#)

Duplicación de arrays unidimensionales

 clonico1: [Lista.java](#), [ClonicoApp1.java](#)

Veamos de nuevo [la clase *Lista*](#) que hemos cambiado ligeramente para mostrar algunas particularidades del método *clone*. Le hemos añadido un string que guarda el nombre de la lista.

```
public class Lista implements Cloneable{
    private int[] x;
    public int n;
    public String nombre;
    //...
}
```

Para duplicar un objeto de la clase *Lista* hemos de implementar el interface *Cloneable* y redefinir la función miembro *clone* de la clase base *Object*, de forma similar a como se hizo para definir la versión [clone de la clase *Rectangulo*](#). Recuérdese que los arrays vienen descritos por clases en Java. Por tanto, la versión *clone* de la clase *Lista* ha de contener el código correspondiente a la copia del subobjeto array unidimensional de enteros *int[]*.

```
public Object clone(){
    Lista obj=null;
    try{
        obj=(Lista)super.clone();
    }catch(CloneNotSupportedException ex){
        System.out.println(" no se puede duplicar");
    }
}
```

```

        obj.x=(int[])obj.x.clone();
        return obj;
    }

```

Como podemos apreciar en el código, el subobjeto de la clase *String* se copia de forma automática como los tipos básicos de datos, sin necesidad de hacerlo de forma explícita.

El código completo de este ejemplo, es el siguiente

```

public class Lista implements Cloneable{
    private int[] x;
    public int n;
    public String nombre;
    public Lista(int[] x, String nombre) {
        this.x=x;
        n=x.length;
        this.nombre=nombre;
    }
    public Object clone(){
        Lista obj=null;
        try{
            obj=(Lista)super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        obj.x=(int[])obj.x.clone();
        return obj;
    }
    private void ordenar(){
        int aux;
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[i]>x[j]){
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;
                }
            }
        }
    }
    public int menor(){
        Lista aux=(Lista)clone();

```



```

        aux.ordenar();
        return aux.x[0];
    }

    public int mayor(){
        ordenar();
        return x[n-1];
    }
    public String toString(){
        String texto=nombre;
        for(int i=0; i<n; i++){
            texto+="\t"+x[i];
        }
        return texto;
    }
}

```

Modificación del objeto cuando se llama a las funciones miembro

Para hallar el *mayor* de una lista de números, ordenamos los datos en orden creciente y luego, retornamos el último.

```

public int mayor(){
    ordenar();
    return x[n-1];
}

```

La función pública *mayor* llama a *ordenar*, la cual modifica el orden de los elementos del array miembro dato *x*. El objeto resulta modificado al llamar a la función miembro *mayor*.

Si por algún motivo deseamos preservar el orden original de los datos, procedemos del modo que se indica en la definición de la función *menor*.

```

public int menor(){
    Lista aux=(Lista)clone();
    aux.ordenar();
    return aux.x[0];
}

```

Creamos una copia *aux* de **this** (línea marcada en negrita), modificamos la copia *aux* llamando a la función miembro *ordenar*, pero dejando intacto a **this**, y por último, devolvemos el primer elemento del array *x* de *aux*. El objeto *aux* de la clase *Lista* es temporal, ya que solamente existe en el cuerpo de la función *menor*.

Si hemos redefinido la función miembro *toString* en la clase *Lista*, podemos comprobar como la llamada a la función *menor* no modifica al objeto que la llama, y la llamada a la función *mayor* modifica al objeto que la llama.

```
int[] datos={2, 5, -1, 3, 0};
Lista lista=new Lista(datos, "ordenar");
System.out.println("antes ---->" + lista);
lista.menor();
System.out.println("despues ---->" + lista);
System.out.println("*****");
System.out.println("antes -----> " + lista);
lista.mayor();
System.out.println("despues ----> " + lista);
```

Duplicación de arrays bidimensionales



clonico2: [Matriz.java](#), [ClonicoApp2.java](#)

Finalizaremos este estudio, con la duplicación de un objeto, uno de cuyos miembros es un array bidimensional.

Veamos la definición de la clase *Matriz* cuadrada, que tiene como miembros dato un array bidimensional y la dimensión de la matriz.

```
public class Matriz implements Cloneable{
    private int[][] x;
    public int n;
    //...
}
```

De nuevo, para poder hacer copias de los objetos de la clase *Matriz* hemos de implementar el interface *Cloneable*, y redefinir la función miembro *clone* de la clase base *Object*.

Dado que los objetos de la clase *Matriz* son compuestos, contienen subobjetos de la clase *int[][]*, la redefinición de *clone* será similar a la de la clase *Rectangulo* y *Lista*, pero además, hemos de entender que es un [array bidimensional](#).

Un array bidimensional está formado por arrays unidimensionales. Luego, para copiar un array bidimensional hemos de copiar cada uno de los arrays unidimensionales que lo forman.

```
public Object clone(){
    Matriz obj=null;
    try{
        obj=(Matriz)super.clone();
    }catch(CloneNotSupportedException ex){
        System.out.println(" no se puede duplicar");
    }
    obj.x=(int[][])obj.x.clone();
    for(int i=0; i<obj.x.length; i++){
        obj.x[i]=(int[])obj.x[i].clone();
    }
    return obj;
}
```

```
public class Matriz implements Cloneable{
    private int[][] x;
    public int n;
    public Matriz(int[][] x) {
        this.x=x;
        n=x.length;
    }
    public Object clone(){
        Matriz obj=null;
        try{
            obj=(Matriz)super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        obj.x=(int[][])obj.x.clone();
        for(int i=0; i<obj.x.length; i++){
            obj.x[i]=(int[])obj.x[i].clone();
        }
        return obj;
    }
    public void modificar(){
```

```

        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                x[i][j]=x[j][i]+i-j;
            }
        }
    }
    public String toString(){
        String texto="";
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                texto+="\t"+x[i][j];
            }
            texto+="\n";
        }
        texto+="\n";
        return texto;
    }
}

```

Hemos definido en la clase *Matriz* una función miembro denominada *modificar* que modifica los valores que guarda el array bidimensional del objeto que la llama. Esta función puede realizar cualquier tarea como la de transformar una matriz en otra triangular para calcular su determinante.

Supongamos una función *f* y otra *g* a la que se le pasa un objeto de la clase *Matriz* en su único argumento. En el cuerpo de *f* y *g* se realizan diversas tareas con el objeto que se ha pasado por ejemplo, llamar a la función miembro *modificar*.

```

void f(Matriz matriz){
    matriz=(Matriz)matriz.clone();
    matriz.modificar();
}
void g(Matriz matriz){
    matriz.modificar();
}

```

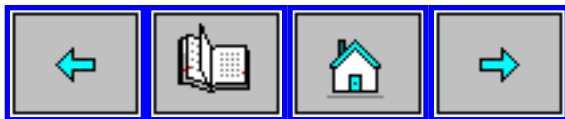
En la primera versión *f*, la llamada a la función miembro *modificar* la realiza una copia de la *matriz* que se le pasa, mientras que en la función *g*, la llamada a la función miembro *modificar* se realiza a partir de la matriz que se le pasa.

Si la clase *Matriz* ha redefinido la función miembro *toString* de la clase base *Object*, podemos comprobar el efecto de las llamadas a las funciones *f* y *g*.

```
int[][] datos={{2, 5, -1}, {0, -2, 4}, {1, -3, 2}};  
Matriz obj=new Matriz(datos);  
System.out.println("antes "+ obj);  
f(obj);  
System.out.println("después "+ obj);  
System.out.println("*****");  
System.out.println("antes "+ obj);  
g(obj);  
System.out.println("después "+ obj);
```

Comprobaremos que la llamada de *f* preserva la matriz *obj* original, mientras que la llamada de *g*, modifica el objeto original *obj*.

El primer applet



[Introducción](#)

[El primer applet](#)

[Insertando el applet en una página web](#)

[Comprensión de los archivos `.class` \(deployment\)](#)

El lenguaje Java se puede usar para crear dos tipos de programas: los applets y las aplicaciones. Un applet es un elemento más de una página web, como una imagen o una porción de texto. Cuando el navegador carga la página web, el applet insertado en dicha página se carga y se ejecuta.

Mientras que un applet puede transmitirse por la red Internet una aplicación reside en el disco duro local. Una aplicación Java es como cualquier otra que está instalada en el ordenador. La otra diferencia es que un applet no está autorizado a acceder a archivos o directorios del ordenador cliente si no es un applet completamente fiable.

El primer applet



applet1: [Applet1.java](#)



Para crear un applet tenemos que definir una clase denominada *Applet1* [derivada](#) de *Applet*. La primera sentencia **import** nos proporciona información acerca de las clases del [paquete *applet*](#). Dicho paquete contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador, entre las cuales está la clase base *Applet*.

```
import java.applet.*;
public class Applet1 extends Applet {
}
```

El siguiente paso es dar funcionalidad a la clase, definir nuestras propias funciones miembro o redefinir funciones de la clase base *Applet*.

Definimos la función *init* para establecer el [color de fondo del applet](#) mediante *setBackground*. La función *init* se llama cuando se carga el applet.

```
public class Applet1 extends Applet {
    public void init(){
        setBackground(Color.white);
    }
    //...
}
```

A continuación, vamos a mostrar un mensaje, para ello definimos el método *paint*. El método *paint* nos suministra [el contexto gráfico](#) *g*, un objeto de la clase *Graphics* con el cual podemos dibujar en el área de trabajo del componente llamando desde dicho objeto *g* a las funciones definidas en la clase *Graphics*.

Para mostrar un mensaje, llamamos desde el objeto *g* a la función miembro *drawString*, el primer argumento es el string que deseamos mostrar, y los dos números indican las coordenadas de la línea base del primer carácter.

```
import java.applet.*;

public class Applet1 extends Applet {
    public void init(){
        setBackground(Color.white);
    }
    public void paint(Graphics g){
        g.drawString("Primer applet", 10, 10);
    }
}
```

Un applet, no es como una aplicación que tiene un método **main**. El applet está insertado en una página web que se muestra en la ventana del navegador. El navegador toma el control del applet llamando a algunos de sus métodos, uno de estos es el método *paint* que se llama cada vez que se necesita mostrar el applet en la ventana del navegador.

Cuando el applet se carga, el navegador llama a su método *init*. En este método el programador realiza tareas de inicialización, por ejemplo, establecer las propiedades de los controles, disponerlos en el applet, cargar imágenes, etc.

El método *init* se llama una sola vez. Después, el navegador llama al método *paint*.

A continuación, se llama al método *start*. Este método se llama cada vez que se accede a la página que contiene el applet. Esto quiere decir, que cuando dejamos la página web que contiene el applet y regresamos de nuevo pulsando en el botón "hacia atrás" el método *start* vuelve a llamarse de nuevo, pero no se llama el método *init*.

Cuando dejamos la página web que contiene el applet, por ejemplo, pulsando en un enlace, se llama al método *stop*.

Finalmente, cuando salimos del navegador se llama al método *destroy*.

Insertando un applet en una página web

Las etiquetas HTML como <H1>, <TABLE>, , etc. señalan el tamaño y la disposición del texto y las figuras en la ventana del navegador. Cuando Sun Microsystems desarrolló el lenguaje Java, se añadió la etiqueta que permite insertar applets en las páginas web. Como otras etiquetas tiene un comienzo <APPLET> y un final señalado por </APPLET>

En el Entorno Integrado de Desarrollo (IDE) de JBuilder creamos un proyecto nuevo, en este caso un applet. JBuilder genera la clase que describe el applet con algunos métodos y la guarda en un archivo cuyo nombre es el mismo que el de la clase y con extensión **.java**. Genera también, un archivo HTML para la documentación del proyecto, y el archivo HTML de la página que contiene el applet tal como se ve en el siguiente cuadro


```

<HTML>
<HEAD>
<TITLE>
HTML Test Page
</TITLE>
</HEAD>
<BODY>
applet1.Applet1 will appear below in a Java enabled browser.<BR>
<APPLET
  CODEBASE = "."
  CODE      = "applet1.Applet1.class"
  NAME      = "TestApplet"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
</APPLET>
</BODY>
</HTML>

```

Cuado se compila el applet se producen archivos cuya extensión es **.class**. Uno de estos archivos es el que resulta de la compilación de la clase que describe el applet, en nuestro caso `Applet1.class` situado en el subdirectorio `applet1`.

Si queremos insertar un applet en una página web, la forma más segura es copiar la etiqueta `<APPLET> ... </APPLET>` desde la página web generada por JBuilder a nuestra página.

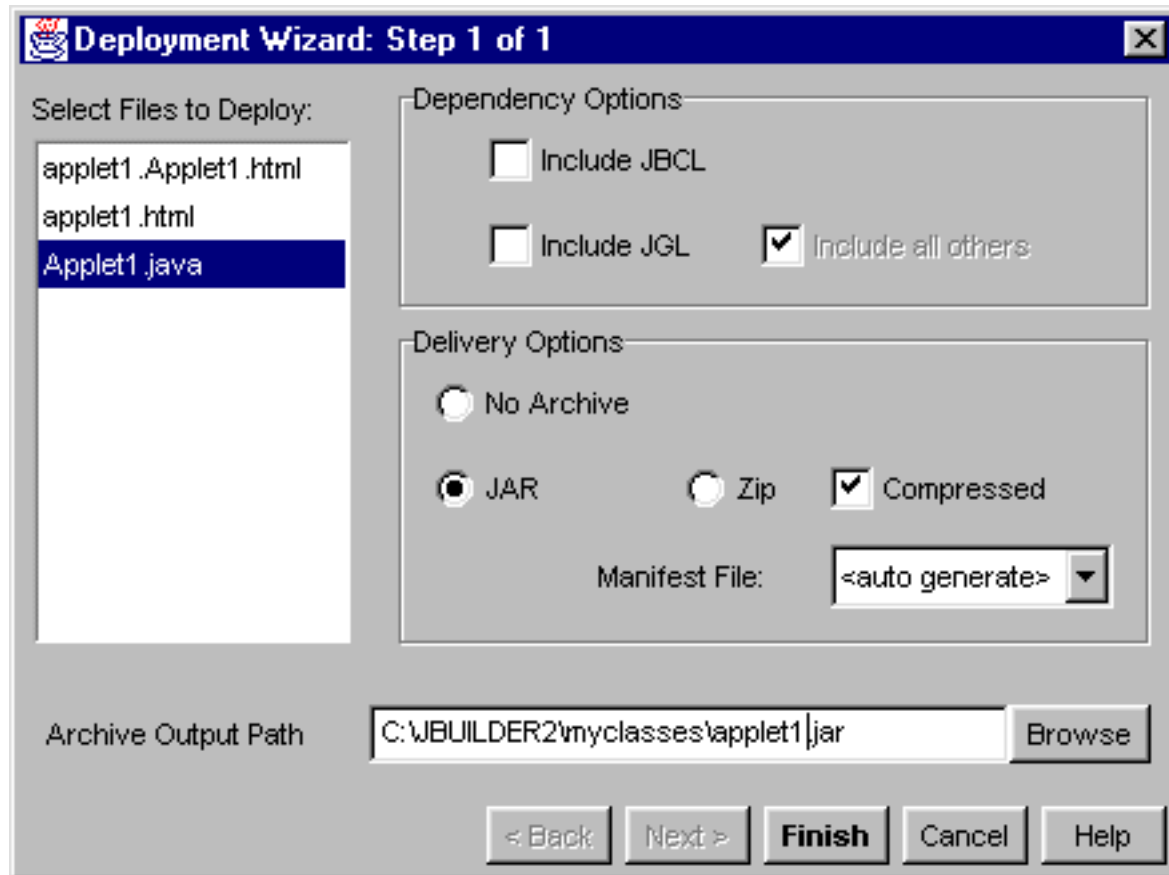
Dentro de la etiqueta `applet` el parámetro más importante es `CODE` que señala el nombre del archivo cuya extensión es **.class**, y cuyo nombre coincide con el de la clase que describe el applet.

Los valores de los parámetros `WIDTH` y `HEIGHT` determinan las dimensiones del applet. En este caso el applet tiene una anchura de 400 y una altura de 300.

El nombre del applet, parámetro `NAME`, es importante cuando se pretende comunicar los applets insertados en una página web.

Comprensión de los archivos .class (deployment)

Cuando el proyecto es complejo, al compilarlo se crean varios archivos **.class** en el mismo subdirectorio. Resulta engorroso trasladarlos desde nuestro ordenador al servidor cuando publicamos las páginas web. Se corre el peligro de mezclar los archivos o perder alguno por el camino. Para facilitar esta tarea, se puede comprimir todos los archivos **.class** resultantes del proceso de compilación de un proyecto en un único archivo cuya extensión es **.jar**, mediante un asistente denominado **Deployment Wizard**

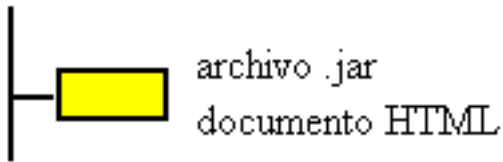


Excluimos del proceso (deployment) los archivos con extensión **.html**. Introducimos en el campo **Archive Output Path** el nombre del archivo comprimido cuya extensión es **.jar**. Normalmente, le daremos el mismo nombre que tiene el proyecto.

En las versiones anteriores a la 1.1, si un applet estaba formado por varias clases o tenía recursos como imágenes GIF o archivos de sonido, cada uno de los archivos se tenía que descargar individualmente del servidor. Esto suponía una carga extra para el servidor, y la necesidad de que el usuario tuviese que esperar hasta que todos los componenets que forman el applet estuviesen disponibles.

Para solventar este problema Sun introdujo los archivos JAR, que son similares a los archivos ZIP, de hecho se pueden descomprimir con la misma herramienta WinZip. De este modo, todos los archivos que forman el applet están situados en un único archivo comprimido, con lo que disminuye el trabajo del servidor y el tiempo de descarga.

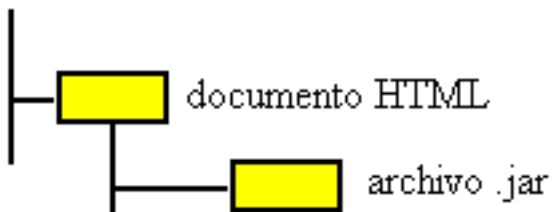
Los archivos JAR incluyen un archivo denominado MANIFEST.MF en el subdirectorio META-INF que contiene la lista de los componentes del archivo JAR y puede incluir firmas digitales.



Supongamos que colocamos el archivo applet1.jar en el mismo subdirectorio que documento HTML que contiene el applet, la etiqueta <APPLET>... </APPLET> se escribe de la forma que sigue. Fijarse que aparece un nuevo parámetro ARCHIVE que indica el nombre y la ubicación del archivo comprimido **.jar**.

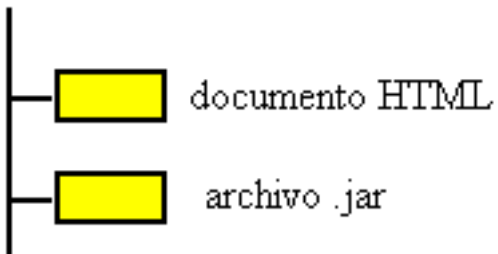
```

<APPLET
  CODE      = "applet1.Applet1.class"
  ARCHIVE = "applet1.jar"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
</APPLET>
  
```



Si el archivo applet1.jar está en un subdirectorio denominado jars por debajo del documento HTML escribimos

```
<APPLET
  CODE      = "applet1.Applet1.class"
  ARCHIVE   = "jars/applet1.jar"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
</APPLET>
```



Si el archivo applet1.jar está en un subdirectorio denominado jars al mismo nivel que el subdirectorio que contiene el documento HTML escribimos

```
<APPLET
  CODE      = "applet1.Applet1.class"
  ARCHIVE   = "../jars/applet1.jar"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
</APPLET>
```

Los parámetros




[Introducción](#)

[La anchura y altura del applet](#)

[Otros parámetros](#)

Los parámetros es la forma en la que se comunica la página web y el applet.

 **applet2:** [Applet2.java](#)



La anchura y altura del applet

Las dimensiones de la applet se establecen mediante los valores de los parámetros `WIDTH` y `HEIGHT` de la etiqueta `applet`.

La anchura y altura del applet se pueden establecer mediante la función *setSize*, de la cual existen dos versiones

```
setSize(400,300);
```

o bien,

```
setSize(new Dimension(400,300));
```

Donde *Dimension*, es una clase con dos miembros datos públicos *width* y *height*. Ambas sentencias establecen la dimensión del applet en 400 pixels de ancho y 300 pixels de alto.

Ahora bien, hemos de tener cuidado con esta sentencia ya que podría plantearse la siguiente situación

contradictoria

```
<APPLET
  CODE      = "applet2.Applet2.class"
  ARCHIVE   = "applet2.jar"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
</APPLET>
```

```
public class Applet2 extends Applet{
    public void init(){
        setSize(150,200);
    }
    //...
}
```

En el appletviewer el tamaño de la ventana y tamaño del applet son distintos. La ventana del appletviewer es mayor que el applet. Sin embargo, en Internet Explorer 5.0 la ventana que aparece en el navegador y el applet tienen las mismas dimensiones, las dadas por los parámetros **WIDTH** y **HEIGHT** dejando sin efecto la sentencia *setSize*.

Para evitar posibles inconvenientes, el applet puede leer los datos asociados a los parámetros **WIDTH** y **HEIGHT** de la etiqueta **APPLET** mediante la función *getParameter* miembro de la clase base *Applet*, y después pasarle dichos datos a la función *setSize* para establecer el tamaño del applet.

En la función miembro *init*, *getParameter* miembro de la clase *Applet* lee el dato asociado al parámetro (**WIDTH** o **HEIGHT**), y devuelve un string, que ha de [convertirse en un dato numérico](#) de tipo **int**. Los valores numéricos se guardan en las variables locales *ancho* y *alto*, y luego, se pasan a la función *setSize* para establecer el tamaño del applet. De este modo el tamaño y disposición del applet está controlado por la página web en la que está insertado.

```
public void init(){
    int ancho = Integer.parseInt(this.getParameter("WIDTH"));
    int alto = Integer.parseInt(this.getParameter("HEIGHT"));
    this.setSize(ancho,alto);
}
```

Como vimos al estudiar el capítulo de las [excepciones](#), la conversión de un string en un número es mejor llevarla a cabo, por razones de seguridad, en un bloque **try...catch**. En el caso de que el proceso de conversión falle, se crea y se lanza un objeto *ex* de la clase *NumberFormatException* que es capturado por el bloque **catch** para notificar este problema al usuario.

```
public void init(){
    int ancho=400;        //valores por defecto
    int alto=300;
    try{
        ancho = Integer.parseInt(this.getParameter("WIDTH"));
        alto = Integer.parseInt(this.getParameter("HEIGHT"));
    }catch(NumberFormatException ex){
        System.out.println("Error en los parámetros WIDTH y HEIGHT");
    }
    this.setSize(ancho,alto);
}
```

Otros parámetros

Normalmente, interrumpimos el proceso de creación del applet con el asistente en el primer paso, pero como ya se ha comentado, JBuilder nos proporciona un diálogo para añadir parámetros a la etiqueta **APPLET**.

En el segundo paso del asistente de creación de applet [Applet Wizard Step 2 of 3](#) podemos definir los parámetros, generándose automáticamente el código que lee los valores asociados a dichos parámetros.

Applet Parameters

* - Required Field

| Name* | Type* | Desc | Variable* | Default |
|----------|--------|------|-----------|-----------------|
| ABSCISA | int | | x | 10 |
| ORDENADA | int | | y | 20 |
| MENSAJE | String | | texto | I primer applet |

Add Parameter Remove Parameter

< Back Next > Finish Cancel Help

Los parámetros que vamos a definir son tres, la posición (ABSCISA y ORDENADA) del mensaje que se va a imprimir en el applet, y el MENSAJE mismo. En la figura y en el cuadro adjunto se muestra la definición de cada uno de los parámetros

| Nombre | Tipo | Variable | Valor por defecto |
|----------|--------|----------|-------------------|
| ABSCISA | int | x | 10 |
| ORDENADA | int | y | 20 |
| MENSAJE | String | texto | El primer applet |

Una vez completado el primer parámetro, se pulsa el botón titulado **Add Parameter** para continuar con el siguiente. El campo titulado **Desc** corresponde a la descripción del parámetro que es opcional.

Una vez pulsado el botón **Finish**, JBuilder genera los parámetros en el archivo **.html**, y genera el código fuente que lee los valores asociados a dichos parámetros, tal como se ve en los dos cuadros que se muestran a continuación. Asimismo, genera una función la redefinición de *getParameterInfo* de la clase base *Applet* que devuelve la información relativa a los parámetros pero que no tiene de momento interés


```

<APPLET
  CODEBASE = "."
  CODE      = "applet2.Applet2.class"
  NAME      = "TestApplet"
  WIDTH     = 400
  HEIGHT    = 300
  HSPACE    = 0
  VSPACE    = 0
  ALIGN     = middle
>
<PARAM NAME = "ABSCISA" VALUE = "10">
<PARAM NAME = "ORDENADA" VALUE = "20">
<PARAM NAME = "MENSAJE" VALUE = "El primer applet">
</APPLET>

```

El código generado por JBuilder incluye la definición de una función auxiliar *getParameter* con dos argumentos, el nombre del parámetro y su valor por defecto, esta función llama a *getProperty* de la clase *System*

```

public class Applet2 extends Applet {
    boolean isStandalone=false;
    int x;
    int y;
    String texto;
    //Get a parameter value

    public String getParameter(String key, String def) {
        return isStandalone ? System.getProperty(key, def) :
            (getParameter(key) != null ? getParameter(key) : def);
    }

    public void init() {
        try { x = Integer.parseInt(this.getParameter("ABSCISA", "10")); }
        catch (Exception e) { e.printStackTrace(); }
        try { y = Integer.parseInt(this.getParameter("ORDENADA", "20")); }
        catch (Exception e) { e.printStackTrace(); }
        try { texto = this.getParameter("MENSAJE", "El primer applet"); }
        catch (Exception e) { e.printStackTrace(); }
        try {
            jbInit();
        }
        catch (Exception e) {

```

```

        e.printStackTrace();
    }
}
//...
}

```

Podemos optar por conservar el código generado por el IDE o escribir nuestro propio código, de forma semejante al que hemos escrito para leer los parámetros `WIDTH` y `HEIGHT`. Ahora bien, los valores devueltos por *getParameter* y después convertidos se deben de guardar en variables de instancia para que luego puedan ser utilizadas por otras funciones miembro. Así, el valor del parámetro `ABSCISA` se guarda en el miembro `x`, el valor del parámetro `ORDENADA` en `y`, y el valor del parámetro `MENSAJE` en *texto*.

En la redefinición de *paint*, se muestra en el contexto gráfico `g`, el contenido del mensaje guardado en *texto* en la posición `x` e `y` mediante la llamada a la función *drawString*.

```

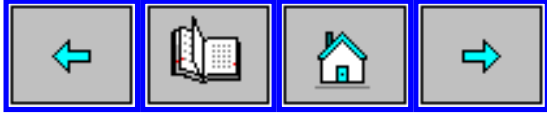
public class Applet2 extends Applet {
    int x;
    int y;
    String texto;

    public void init() {
        int ancho=250;           //valores por defecto
        int alto=100;
        x=10;
        y=20;
        try{
            ancho=Integer.parseInt(this.getParameter("WIDTH"));
            alto=Integer.parseInt(this.getParameter("HEIGHT"));
            x=Integer.parseInt(this.getParameter("ABSCISA"));
            y=Integer.parseInt(this.getParameter("ORDENADA"));
        }catch(NumberFormatException ex){
            System.out.println("Error en los parámetros");
        }
        texto=this.getParameter("MENSAJE");

        this.setSize(ancho,alto);
    }
    public void paint(Graphics g){
        g.drawString(texto, x, y);
    }
}

```


Respuesta a las acciones del usuario sobre un grupo de controles (casillas y botones de radio)



Sucesos (events)

[Respuesta a las acciones sobre controles casilla de verificación \(*Checkbox*\)](#)

[Respuesta a las acciones del usuario sobre un grupo de botones de radio](#)

Respuesta a las acciones sobre casillas de verificación (*Checkbox*)



casilla: [CasillaApplet.java](#)



El control *Checkbox*

Una casilla de verificación es un objeto de la clase *Checkbox*. Para crear tres casillas de verificación se llama a alguno de sus constructores

```
Checkbox chkCorbata = new Checkbox();  
Checkbox chkChaqueta = new Checkbox();  
Checkbox chkSombrero = new Checkbox();
```

Establecemos el título o texto que identifica cada una de las casillas con la función miembro *setLabel*

```
chkCorbata.setLabel( "Corbata" );  
chkChaqueta.setLabel( "Chaqueta" );  
chkSombrero.setLabel( "Sombrero" );
```

Establecemos su estado inicial mediante la función miembro *setState*. Si queremos que la casilla titulada "Sombrero" aparezca inicialmente activada (checked), se escribe

```
chkSombrero.setState( true );
```

La función *getState*, devuelve **true**, si la casilla está activada (checked) y **false** cuando está desactivada (unchecked)

```
boolean estado=chkCorbata.getState()
```

Propósito

Tenemos tres casillas tituladas Corbata, Chaqueta y Sombrero inicialmente desactivadas. Cuando se activan se imprime un mensaje que indica que prenda o prendas lleva puestas.

Diseño

Crear el applet, situar tres casillas (*Checkbox*) en la parte superior.

Cambiar sus propiedades **name** y **label** en sus respectivas hojas de propiedades

Establecer [*FlowLayout*](#), como gestor de diseño del applet. Cambiar la propiedad **hgap** para separar horizontalmente los botones.

Redefinir la función *paint* para mostrar el mensaje

Respuesta a las acciones del usuario

Para manejar los sucesos que se producen en las casillas de verificación se deben seguir los mismos pasos que para [seleccionar un elemento en una lista](#) o en una caja de selección.

- Se crea una clase que implemente el interface *ItemListener*, y que ha de definir *itemStateChanged*
- Mediante *addItemListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso viene encapsulada en un objeto de la clase *ItemEvent*.

Siguiendo el modelo denominado Standard adapter empleado para [manejar los sucesos que provienen de tres](#)

[botones](#), creamos una clase *ItemCasillas* que implemente el interface *ItemListener*, y defina la función *itemStateChanged*.

```
class ItemCasillas implements ItemListener{
    private CasillaApplet applet;
    public ItemCasillas(CasillaApplet applet){
        this.applet=applet;
    }
    public void itemStateChanged(ItemEvent ev){
        applet.casillas_itemStateChanged(ev);
    }
}
```

Creamos un objeto *item* de la clase *ItemCasillas* y le pasamos **this**, el objeto applet, para inicializar el miembro dato *applet* de la clase *ItemCasillas*. En la definición de la función miembro *itemStateChanged*, se llama a la función respuesta *casillas_itemStateChanged*.

```
ItemCasillas item=new ItemCasillas(this);
```

Asociamos mediante *addItemListener* cada una de las casillas de verificación con el objeto *item* que registra las acciones sobre estos controles.

```
chkCorbata.addItemListener(item);
chkChaqueta.addItemListener(item);
chkSombrero.addItemListener(item);
```

En la definición de la función respuesta *casillas_itemStateChanged*, se describe la tarea a realizar. En nuestro caso, describe como viste una persona:

- Si la casilla *chkChaqueta* está activada, entonces añade al texto "Cómo viste" el mensaje "viste chaqueta".
- Si la casilla *chkCorbata* está activada, le añade el mensaje "lleva corbata".
- Si la casilla *chkSombrero* está activada ,le añade el mensaje "se pone el sombrero".

La función *getState*, devuelve **true**, si la casilla está activada (checked) y **false** cuando está desactivada (unchecked)

```
public void casillas_itemStateChanged(ItemEvent ev){
    texto="Cómo viste: ";
    if(chkCorbata.getState()){
        texto+="lleva corbata, ";
    }
    if(chkChaqueta.getState()){
```

```

        texto+="viste chaqueta, ";
    }
    if(chkSombrero.getState()){
        texto+="se pone el sombrero, ";
    }
    repaint();
}

```

Finalmente, se llama la función *paint* para mostrar el texto resultante, mediante la llamada a la función *drawString*.

```

public void paint(Graphics g){
    g.drawString(texto, 10, 50);
}

```

El código completo de este ejemplo, es el siguiente

```

package casilla;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CasillaApplet extends Applet {
    Checkbox chkCorbata = new Checkbox();
    Checkbox chkChaqueta = new Checkbox();
    Checkbox chkSombrero = new Checkbox();
    FlowLayout flowLayout1 = new FlowLayout();
    String texto="Cómo viste: ";

    public void init() {
        chkCorbata.setLabel("Corbata");
        chkChaqueta.setLabel("Chaqueta");
        chkSombrero.setLabel("Sombrero");
        ItemCasillas item=new ItemCasillas(this);
        chkCorbata.addItemListener(item);
        chkChaqueta.addItemListener(item);
        chkSombrero.addItemListener(item);
        flowLayout1.setHgap(10);
        this.setLayout(flowLayout1);
        this.add(chkCorbata, null);
        this.add(chkChaqueta, null);
        this.add(chkSombrero, null);
    }

    public void casillas_itemStateChanged(ItemEvent ev){

```

```

    texto="Cómo viste: ";
    if(chkCorbata.getState()){
        texto+="lleva corbata, ";
    }
    if(chkChaqueta.getState()){
        texto+="viste chaqueta, ";
    }
    if(chkSombrero.getState()){
        texto+="se pone el sombrero, ";
    }
    repaint();
}

public void paint(Graphics g){
    g.drawString(texto, 10, 50);
}
}

class ItemCasillas implements ItemListener{
    private CasillaApplet applet;
    public ItemCasillas(CasillaApplet applet){
        this.applet=applet;
    }
    public void itemStateChanged(ItemEvent ev){
        applet.casillas_itemStateChanged(ev);
    }
}

```

Respuesta a las acciones del usuario sobre un grupo de botones de radio



radio: [RadioApplet.java](#)

En la página anterior hemos visto como se define una respuesta única a la [acción sobre un conjunto de tres botones](#). Un grupo de botones de radio, es una elección más acertada que un conjunto de tres botones, ya que en un grupo uno sólo de los botones de radio está en estado activado. Para crear un grupo de botones de radio es necesario seguir los siguientes pasos.



Los controles *Checkbox* y *CheckboxGroup*

Se crean los controles del tipo *Checkbox* y un control *CheckboxGroup*.

```
Checkbox chkRojo = new Checkbox();
Checkbox chkVerde = new Checkbox();
Checkbox chkAzul = new Checkbox();
CheckboxGroup chkGrupo=new CheckboxGroup();
```

En *init* se pone una etiqueta para identificar cada uno de los botones de radio, mediante *setLabel*,

```
chkRojo.setLabel("Rojo");
chkVerde.setLabel("Verde");
chkAzul.setLabel("Azul");
```

Se asocia cada uno de los botones de radio con su grupo, el objeto *chkGrupo* de la clase *CheckboxGroup*. Por último, se establece el botón de radio en estado activado, que se presenta inicialmente cuando aparece el applet, mediante la función miembro *setSelectedCheckbox* de la clase *CheckboxGroup*.

```
chkRojo.setCheckboxGroup(chkGrupo);
chkVerde.setCheckboxGroup(chkGrupo);
chkAzul.setCheckboxGroup(chkGrupo);
chkGrupo.setSelectedCheckbox(chkRojo);
```

Para obtener el botón de radio que ha sido seleccionado se llama a la función miembro *getSelectedCheckbox* de la clase *CheckboxGroup*.

```
Checkbox radio=chkGrupo.getSelectedCheckbox();
```

Propósito

El applet contiene un conjunto de tres botones de radio situados en la parte superior del applet. El nombre de los botones es el de los colores primarios: azul, verde y rojo. Al activar un botón de radio se pinta un rectángulo de dicho color.

Diseño

Crear el applet, situar tres botones de radio (*Checkbox*) en la parte superior.

Cambiar sus propiedades **name** y **label** en las correspondiente hoja de propiedades

Establecer [*FlowLayout*](#), como gestor de diseño del applet. Cambiar la propiedad **hgap** para separar horizontalmente los botones.

En el código fuente crear un objeto de la clase *CheckboxGroup*, asociar mediante *setCheckboxGroup* cada botón de radio al grupo. Establecer el botón de radio inicialmente activado.

Crear un array *colores* de objetos de la clase [*Color*](#) con los tres colores primeros: rojo, verde y azul.

```
final Color[] colores={Color.red, Color.green, Color.blue};
```

Redefinir la función *paint* para pintar un rectángulo

Respuesta a las acciones del usuario

Para responder a las acciones sobre un grupo de botones de radio empleamos el [modelo denominado Standard adapter](#), se ha de crear una clase que denominamos *ItemRadio* que implemente el interface *ItemListener*. La clase define la función *itemStateChanged*. La información acerca del suceso viene encapsulada en el objeto *ev* de la clase *ItemEvent*.

```
class ItemRadio implements ItemListener{
    private RadioApplet applet;
    public ItemRadio(RadioApplet applet){
        this.applet=applet;
    }
    public void itemStateChanged(ItemEvent ev){
        applet.radio_itemStateChanged(ev);
    }
}
```

Se crea un objeto *item* de la clase *ItemRadio* y se le pasa **this** al constructor, para inicializar el miembro dato *applet* de la clase *ItemRadio*. En la definición de la función miembro *itemStateChanged*, se llama a la función respuesta *radio_itemStateChanged*.

```
ItemRadio item=new ItemRadio(this);
```

Asociamos, mediante *addItemListener*, cada una de los botones de radio con el objeto *item* que está

interesado en las acciones sobre dichos controles.

```
chkRojo.addItemListener(item);
chkVerde.addItemListener(item);
chkAzul.addItemListener(item);
```

En la definición de la función respuesta *radio_itemStateChanged*, se describe la tarea a realizar. En nuestro caso dibujar un rectángulo del color que se corresponde con el título del botón de radio.

Primero tenemos que saber sobre cual de los botones de radio hemos actuado, para ello se llama a la función miembro *setSelectedCheckbox*, de la clase *CheckboxGroup*. Posteriormente, obtenemos el índice del color del botón de radio que ha sido activado

```
public void radio_itemStateChanged(ItemEvent ev){
    Checkbox radio=chkGrupo.getSelectedCheckbox();
    if(radio.equals(chkRojo)){
        indice=0;
    }
    if(radio.equals(chkVerde)){
        indice=1;
    }
    if(radio.equals(chkAzul)){
        indice=2;
    }
    repaint();
}
```

Finalmente, se llama la función *paint* para dibujar el rectángulo pintado del color seleccionado.

```
public void paint(Graphics g){
    g.setColor(colores[indice]);
    g.fillRect(20, 50, 100, 50);
}
```

Además de las funciones miembro de la clase *Checkbox*, el objeto *ev* de la clase *ItemEvent*, nos suministra [información acerca del suceso](#) originado por la acción del usuario. Así, *getItemSelectable* nos devuelve el control sobre el que se ha actuado. *getStateChanged* nos devuelve un entero que nos indica entre otras cosas, si la casilla sobre la que se ha actuado ha sido activada o desactivada. El código equivalente de la función respuesta *radio_itemStateChanged*, sería el siguiente

```
public void radio_itemStateChanged(ItemEvent ev){
    Object fuente = ev.getItemSelectable();
    if((fuente==chkRojo)&& (ev.getStateChange() == ItemEvent.SELECTED)){
        indice=0;
    }
}
```

```

    }
    if((fuente==chkVerde)&& (ev.getStateChange() == ItemEvent.SELECTED)){
        indice=1;
    }
    if((fuente==chkAzul)&& (ev.getStateChange() == ItemEvent.SELECTED)){
        indice=2;
    }
    repaint();
}

```

El código completo de este ejemplo, es el siguiente

```

package radio;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RadioApplet extends Applet {
    Checkbox chkRojo = new Checkbox();
    Checkbox chkVerde = new Checkbox();
    Checkbox chkAzul = new Checkbox();
    CheckboxGroup chkGrupo=new CheckboxGroup();
    FlowLayout flowLayout1 = new FlowLayout();
    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice;

    public void init() {
        chkRojo.setLabel("Rojo");
        chkVerde.setLabel("Verde");
        chkAzul.setLabel("Azul");
        chkRojo.setCheckboxGroup(chkGrupo);
        chkVerde.setCheckboxGroup(chkGrupo);
        chkAzul.setCheckboxGroup(chkGrupo);
        chkGrupo.setSelectedCheckbox(chkRojo);

        ItemRadio item=new ItemRadio(this);
        chkRojo.addItemListener(item);
        chkVerde.addItemListener(item);
        chkAzul.addItemListener(item);
        flowLayout1.setHgap(10);
        this.setLayout(flowLayout1);
        this.add(chkRojo, null);
        this.add(chkVerde, null);
        this.add(chkAzul, null);
    }
}

```

```

    }

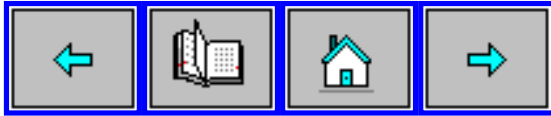
    public void radio_itemStateChanged(ItemEvent ev){
        Checkbox radio=chkGrupo.getSelectedCheckbox();
        indice=0;
        if(radio.equals(chkRojo)){
            indice=0;
        }
        if(radio.equals(chkVerde)){
            indice=1;
        }
        if(radio.equals(chkAzul)){
            indice=2;
        }
        repaint();
    }

    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(20, 50, 100, 50);
    }
}

class ItemRadio implements ItemListener{
    private RadioApplet applet;
    public ItemRadio(RadioApplet applet){
        this.applet=applet;
    }
    public void itemStateChanged(ItemEvent ev){
        applet.radio_itemStateChanged(ev);
    }
}

```

La programación del ratón (I)



Sucesos (events)

Introducción

Listeners y adapters

Pulsando el botón izquierdo del ratón

Dibujar los puntos y guardarlos en memoria

Introducción

El ratón es uno de los dispositivos estándares en un interfaz gráfico, que facilita la interacción del usuario con el programa. Las acciones de pulsar sobre un botón, seleccionar un elemento del menú, o una herramienta en una caja de herramientas, activar o desactivar una casilla de verificación, dibujar una figura en el canvas del programa Paint de Windows, seleccionar una palabra en el procesador de textos, etc, se realizan frecuentemente cuando se trabaja con algún programa.

No hace demasiado tiempo, la programación del ratón era complicada y se realizaba a bajo nivel, con instrucciones cercanas al lenguaje de la máquina. La creación de un interfaz gráfico que hiciese uso del ratón era una tarea muy complicada hasta la aparición de las librerías específicas, primero de funciones y luego de clases, como TurboVision, ObjectVision, etc. Dichas librerías definen elementos estándares, como ventanas, marcos, controles y menús, dejando al usuario la tarea de crear el interfaz con dichos elementos, y de definir su comportamiento, liberándole de los detalles de la programación a bajo nivel.

Listeners y adapters

Un [interface](#) declara un conjunto de funciones, las clases implementan, (**implements**), los interfaces definiendo cada una de dichas funciones. Por ejemplo, si una clase *BotonRaton* implementa el interface *MouseListener*, ha de redefinir todas las funciones aunque solamente estemos interesados en algunas de ellas.

```

class BotonRaton implements MouseListener{
    public void mouseEntered(MouseEvent ev){
        System.out.println("El puntero está sobre el botón");
    }
    public void mouseExited(MouseEvent ev){
        System.out.println("El puntero ha salido del botón");
    }
    public void mousePressed(MouseEvent ev){}
    public void mouseReleased(MouseEvent ev){}
    public void mouseClicked(MouseEvent ev){}
    //otras funciones miembro...
}

```

Para evitar el código inútil, se han creado clases que terminan con la palabra *Adapter*, que implementan el correspondiente interface *Listener* redefiniendo todas las funciones declaradas en el interface sin que hagan nada en particular.

```

public abstract class MouseAdapter implements MouseListener {
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}

```

La clase *BotonRaton* ya no implementa (*implements*) el interface *MouseListener*, sino que deriva (*extends*) de la clase *MouseAdapter*. El código que define la clase *BotonRaton* se escribe en términos de la clase *MouseAdapter* de la siguiente forma

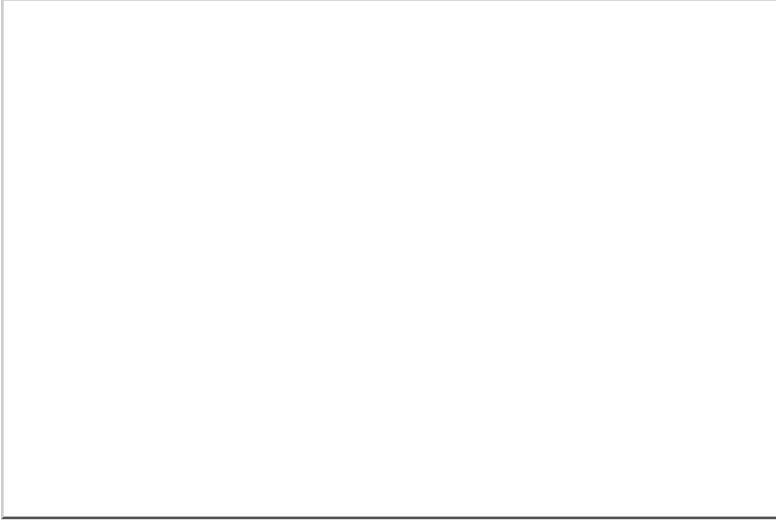
```

class BotonRaton extends MouseAdapter{
    public void mouseEntered(MouseEvent ev){
        System.out.println("El puntero está sobre el botón");
    }
    public void mouseExited(MouseEvent ev){
        System.out.println("El puntero ha salido del botón");
    }
    //otras funciones miembro...
}

```

El programador puede elegir entre implementar (**implements**) un interface o derivar (**extends**) de un adaptador según sea el caso, tal como veremos en esta página y la siguiente.

Pulsando el botón izquierdo del ratón



Propósito

Vamos a dibujar pequeños círculos de color rojo sobre el área de trabajo del applet cada vez que se pulsa el botón izquierdo del ratón. Los puntos son círculos centrados en la posición que señala el puntero del ratón cuando se pulsa el botón izquierdo.

Diseño

Crear un applet mínimo, solamente con el método *init* o *jbInit* si se emplea JBuilder.

En la función respuesta a la acción de pulsar el botón izquierdo del ratón, dibujar un círculo centrado en la posición en el que se ha pulsado el botón izquierdo del ratón. Las coordenadas x e y de dicho punto se obtienen a partir del objeto *ev* de la clase *MouseEvent* mediante *getX* y *getY*. A continuación, dibujar el círculo en el [contexto gráfico](#) del applet.

```
void funcionRespuesta(MouseEvent ev) {  
    int x=ev.getX();  
    int y=ev.getY();  
}
```

Respuesta a las acciones del usuario

Vamos a estudiar de nuevo, tres aproximaciones para resolver el problema.

1.-La clase derivada de *Applet* implementa el interface *MouseListener*.



raton1: [RatonApplet1.java](#)

Ya vimos como podíamos manejar los sucesos provenientes de un botón haciendo que la clase que describe el [applet implemente el interface *ActionListener*](#). En este ejemplo vamos a emplear el mismo modelo.

La clase *RatonApplet1* solamente puede derivar de *Applet*, no puede derivar de más de una clase, pero puede implementar más de un interface. La clase *RatonApplet1* al implementar el interface *MouseListener* debe definir todas las funciones declaradas en dicho interface, aunque solamente estemos interesados en una de ellas.

```
public class RatonApplet1 extends Applet implements MouseListener {
    //....
    public void mousePressed(MouseEvent event) {
        //código ...
    }
    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
}
```

Asociamos el componente que genera los sucesos, el applet (**this**), y el objeto de la clase que implementa el interfaz *MouseListener*, es decir, que maneja dichos sucesos, que es también el applet (**this**).

```
this.addMouseListener(this);
```

o abreviadamente,

```
addMouseListener(this);
```

Cuando se pulsa el botón izquierdo del ratón se llama a *mousePressed*, aquí tenemos que situar el código correspondiente a la tarea a realizar, que no es otra que dibujar pequeños círculos de color rojo.

En primer lugar, se obtiene el [contexto gráfico](#) *g*, (un objeto de la clase *Graphics*) mediante *getGraphics*. Una vez obtenido el contexto gráfico, se pueden pintar en ella llamando a distintas funciones definidas en la clase *Graphics*. Finalmente, no debemos de olvidarnos de liberar los recursos asociados al contexto gráfico *g*, mediante la llamada a *dispose*.

La función respuesta *mousePressed* nos suministra un objeto *ev* de la clase *MouseEvent* que nos proporciona información acerca del suceso (event). Las funciones miembro *getX*, *getY* de la clase *MouseEvent*, nos suministra las coordenadas del punto donde estaba situado el puntero del ratón en el momento de pulsar su botón izquierdo. Alternativamente, *getPoint* nos suministra estas dos coordenadas

encapsuladas en un objeto de la clase *Point*. Una vez que tenemos las coordenadas del punto podemos dibujar mediante *fillOval* un círculo centrado en dicho punto de un determinado radio.

```
public void mousePressed(MouseEvent ev) {
    Graphics g = getGraphics();
    g.setColor(Color.red);
    g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
    g.dispose();
}
```

El código completo de este ejemplo, es el siguiente

```
package raton1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet1 extends Applet implements MouseListener {
    private final int radio = 8;
    public void init() {
        addMouseListener(this);
    }
    public void mousePressed(MouseEvent ev) {
        Graphics g = getGraphics();
        g.setColor(Color.red);
        g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
        g.dispose();
    }
    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
}
```

2.-Creamos una clase que deriva de *MouseAdapter*



raton2: [RatonApplet2.java](#)

En la segunda aproximación similar a la denominada [Standard adapter](#), creamos una clase denominada *RatonRegistra* que en vez de implementar el interface *MouseListener* deriva de *MouseAdapter* y redefine *mousePressed*, véase el apartado [listeners y adapters](#) para ahorrarnos la definición de varias funciones que no realizan ninguna tarea.

Como hemos visto, la función respuesta *mousePressed* nos suministra el objeto *ev* de la clase *MouseEvent*. De este objeto podemos extraer información acerca de la [fuente de los sucesos](#), el applet, mediante la llamada a la función *getSource*. Después, obtenemos el contexto gráfico *g*, mediante *getGraphics*, y finalmente pintamos en dicho contexto.

```
class RatonRegistra extends MouseAdapter {
    private final int radio = 8;
    public void mousePressed(MouseEvent ev) {
        Applet app = (Applet)ev.getSource();
        Graphics g = app.getGraphics();
        g.setColor(Color.blue);
        g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
        g.dispose();
    }
}
```

Asociamos, mediante *addMouseListener*, la fuente de los sucesos asociados al ratón, el applet (**this**), con un objeto de la clase *RatonRegista* que los maneja.

```
RegistraRaton raton=new RatonRegistra();
this.addMouseListener(raton);
```

o abreviadamente,

```
addMouseListener(new RatonRegistra());
```

El código completo de este ejemplo, es el siguiente.

```

package raton2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet2 extends Applet{
    public void init() {
        addMouseListener(new RatonRegistra());
    }
}

class RatonRegistra extends MouseAdapter {
    private final int radio = 8;
    public void mousePressed(MouseEvent ev) {
        Applet app = (Applet)ev.getSource();
        Graphics g = app.getGraphics();
        g.setColor(Color.blue);
        g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
        g.dispose();
    }
}

```

3.-Creamos una clase anónima que deriva de *MouseAdapter*



raton3: [RatonApplet3.java](#)

En la tercera aproximación ([Anonymous adapter](#)), creamos una clase anónima que deriva de *MouseAdapter*. Esta es la forma que JBuilder genera el código cuando en el modo diseño, seleccionamos el applet (**this**) y en su correspondiente panel Events hacemos doble-clic sobre el editor asociado a *mousePressed*.

```

new java.awt.event.MouseAdapter() {
    //define las funciones respuesta
}

```

El siguiente paso, consiste en asociar mediante *addMouseListener* la fuente de los sucesos asociados al ratón, el applet (**this**), con un objeto de la clase anónima que los registra.

```

this.addMouseListener(new java.awt.event.MouseAdapter() { //... }

```

Ahora solamente nos queda definir el código de la función respuesta *mousePressed*. Como se ha dicho en otro apartado, JBuilder [separa el código de inicialización del código de la función respuesta](#) para que el código quede lo más legible posible. Como podemos apreciar, la función *mousePressed* llama a la función *this_mousePressed*. Se define esta última de forma similar a los dos ejemplos previos

- Se obtiene el contexto gráfico *g* del componente, en este caso el applet mediante *getGraphics*
- Se dibuja sobre dicho contexto *g*, llamando a las funciones miembro de la clase *Graphics*.

```
void this_mousePressed(MouseEvent ev) {
    Graphics g = getGraphics();
    g.setColor(Color.green);
    g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
    g.dispose();
}
```

El código completo de este ejemplo, es el siguiente

```
package raton3;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet3 extends Applet {
    private final int radio = 8;

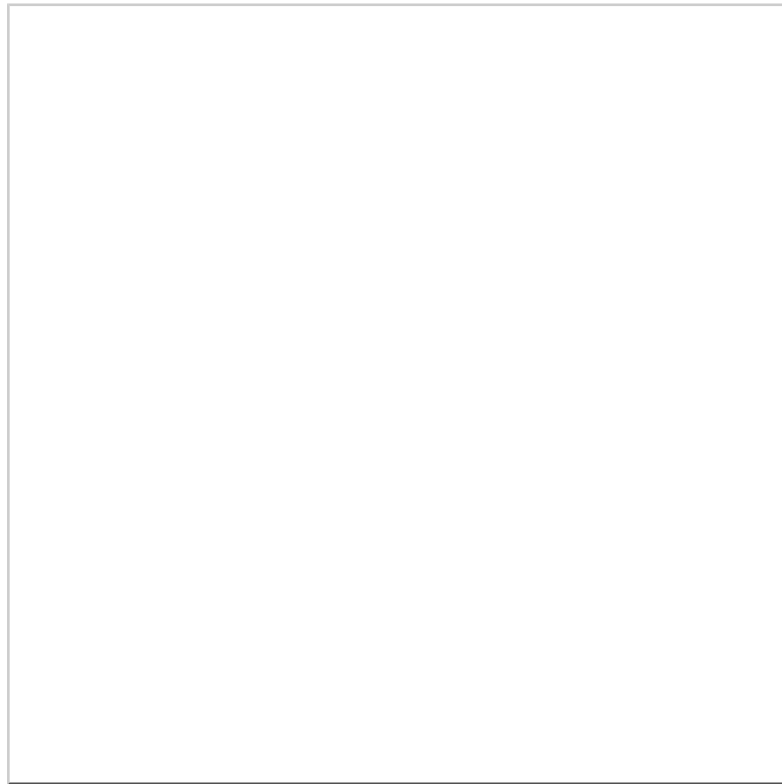
    public void init(){
        this.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                this_mousePressed(e);
            }
        });
    }

    void this_mousePressed(MouseEvent ev) {
        Graphics g = getGraphics();
        g.setColor(Color.green);
        g.fillOval(ev.getX()-radio, ev.getY()-radio, 2*radio, 2*radio);
        g.dispose();
    }
}
```

Dibujar los puntos y guardarlos en memoria



raton4: [RatonApplet4.java](#)



Propósito

En los programas anteriores se dibujan los puntos en la ventana del applet, y se oculta por otra ventana, al descubrir el applet, los puntos han desaparecido. Vamos a crear un programa de manera que las coordenadas de los puntos sean guardados en memoria y que vuelvan a pintarse cuando la ventana del applet se descubre después de haber estado oculta parcial o totalmente por otra ventana. Para probarlo, se dibujan algunos puntos rojos en la superficie del applet, luego se oculta parcialmente el applet subiendo o bajando esta página web, actuando sobre la barra de desplazamiento vertical de la ventana del navegador.

Debemos de recordar, que la función miembro *paint* se redefine en la clase derivada de *Applet* para pintar algo en su superficie, en las siguientes circunstancias:

- Cuando aparece por primera vez el applet
- Cuando se llama a la función *repaint* desde otro método, véase el ejemplo [hacer doble-clic sobre un elemento de la lista](#).
- Cuando la ventana del applet ha sido ocultada por otra y se descubre.

Diseño

Crear un applet mínimo, solamente con el método *init* o *jbInit* si empleamos JBuilder

Crear un array de objetos de la clase *Point*, cuya dimensión venga dada por la constante *MAXPUNTOS*

Guardar los puntos que se dibujan con el ratón en el array. Los objetos de la clase *Point* se obtienen a partir del objeto *ev* de la clase *MouseEvent*, mediante la función *getPoint*.

```
void funcionRespuesta(MouseEvent ev) {
    Point p=ev.getPoint();
}
```

Redefinir la función *paint*, para dibujar todos los puntos guardados en el array.

Respuesta a las acciones del usuario

Partimos del programa anterior guardando los puntos en una array de objetos del tipo *Point*, hasta un máximo de *MAXPUNTOS*. A la definición de la función respuesta *this_mousePressed* le añadimos el código que guarda los puntos en el array *puntos*, y previene que no se sobrepase la dimensión de dicho array

```
void this_mousePressed(MouseEvent ev) {
    Graphics g=getGraphics();
    g.setColor(Color.red);
    if(nPuntos<MAXPUNTOS){
        puntos[nPuntos]=ev.getPoint();
        nPuntos++;
    }
    g.fillOval(ev.getX()-4, ev.getY()-4, 8, 8);
    g.dispose();
}
```

Redefinimos la función *paint* para que dibuje en forma de circunferencias de color azul todos los puntos guardados en memoria. De este modo, diferenciamos los puntos que se dibujan al pulsar el botón izquierdo del ratón (círculos de color rojo), de los puntos cuyas coordenadas se guardan en memoria y se dibujan (circunferencias de color azul) cuando se llama a la función *paint*.

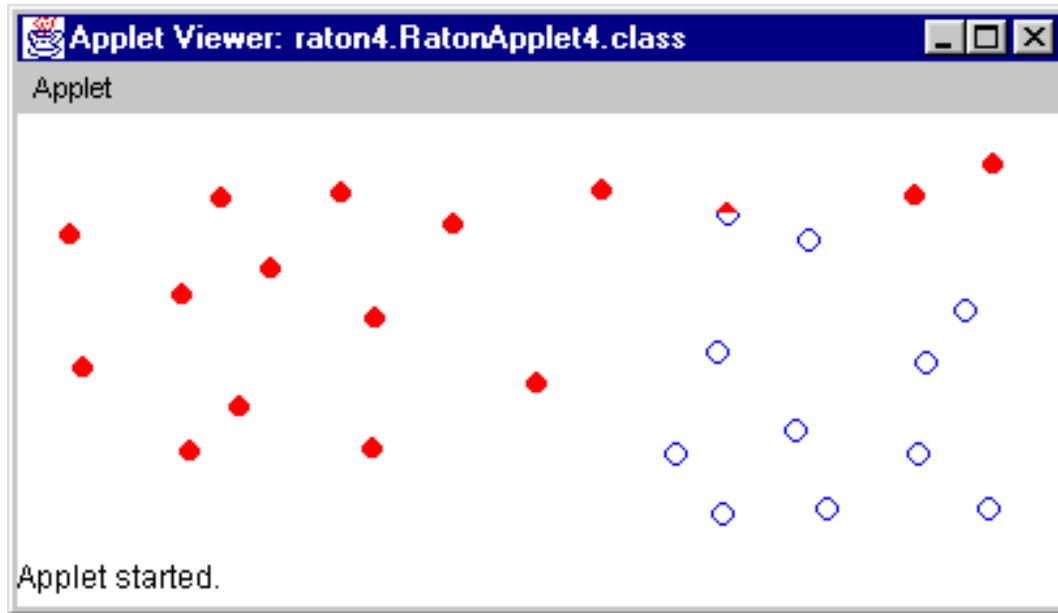
```
public void paint(Graphics g){
    g.setColor(Color.blue);
    for(int i=0; i<nPuntos; i++){
        g.drawOval(puntos[i].x-4, puntos[i].y-4, 8, 8);
    }
}
```

```

    }
}

```

¿Qué ocurre cuando se oculta parcialmente la ventana del applet y se vuelve a descubrir. Nos daremos cuenta como se muestra en la figura adjunta, que solamente se vuelve a dibujar aquella zona que ha estado ocultada. Como podemos ver en la figura se vuelven a dibujar (en color azul) los puntos e incluso (aquí está la sorpresa) la parte de un punto que ha estado parcialmente oculto, los demás permanecen inalterados



El código completo de este ejemplo, es el siguiente

```

package raton4;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet4 extends Applet {
    final int MAXPUNTOS=20;
    Point puntos[]=new Point[MAXPUNTOS];
    int nPuntos=0;
    public void init() {
        this.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                this_mousePressed(e);
            }
        });
    }
    void this_mousePressed(MouseEvent ev) {
        Graphics g=getGraphics();

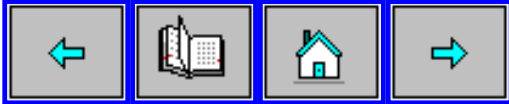
```



```
        g.setColor(Color.red);
        if(nPuntos<MAXPUNTOS){
            puntos[nPuntos]=ev.getPoint();
            nPuntos++;
        }
        g.fillOval(ev.getX()-4, ev.getY()-4, 8, 8);
        g.dispose();
    }

    public void paint(Graphics g){
        g.setColor(Color.blue);
        for(int i=0; i<nPuntos; i++){
            g.drawOval(puntos[i].x-4, puntos[i].y-4, 8, 8);
        }
    }
}
```

La programación del ratón (II)



[Sucesos \(events\)](#)

Un programa simple de dibujo "a mano alzada"



Propósito

Este ejercicio final va a consistir en un programa de dibujo "a mano alzada". Para trazar una línea irregular con el ratón necesitamos de un punto inicial, es decir, aquél que se obtiene al pulsar el botón izquierdo del ratón. Luego, mantenemos pulsado el botón izquierdo del ratón y lo arrastramos dibujándose de este modo una curva irregular que une las posiciones por las que ha pasado el puntero del ratón.

Diseño

Crear un applet mínimo, solamente con el método *init* o *jbInit* si se usa JBuilder

En la función respuesta a la acción de pulsar el botón izquierdo del ratón y de arrastrar el ratón se obtienen las coordenadas *x* e *y* de la posición que señala el puntero, a partir del objeto *ev* de la clase *MouseEvent* mediante *getX* y *getY*.


```
void funcionRespuesta(MouseEvent ev) {  
    int x=ev.getX();  
    int y=ev.getY();  
}
```

Para dibujar una polilínea se precisa de un punto inicial que se obtiene al pulsar el botón izquierdo del ratón. El punto final, se obtiene al arrastrar el ratón luego, se une mediante una línea el punto inicial y final. El punto final se convierte en el inicial para el siguiente movimiento del ratón luego, se traza la segunda línea entre dichos puntos, y así sucesivamente.

Respuesta a las acciones del usuario

Para hacer este programa necesitamos implementar dos interfaces *MouseListener* y *MouseMotionListener* que definan las funciones *mousePressed* y *mouseDragged*, respectivamente. Para ello, vamos a seguir dos aproximaciones.

1.-La clase que describe el applet implementa los interfaces *MouseListener* y *MouseMotionListener*.

 **raton6:** [RatonApplet6.java](#)

La clase derivada que describe [el applet implementa el interface *MouseListener* y *MouseMotionListener*](#). Recuérdese que una clase solamente puede derivar de otra clase pero puede implementar varios interfaces, y tiene que definir todas las funciones declaradas en dichos interfaces.

```
public class RatonApplet6 extends Applet
    implements MouseListener, MouseMotionListener {
    //funciones del interface MouseListener
    public void mousePressed(MouseEvent ev) {
        //código de la función respuesta
    }

    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}

    //funciones del interface MouseMotionListener
    public void mouseDragged(MouseEvent ev) {
        //código de la función respuesta
    }
    public void mouseMoved(MouseEvent event) {}
}
```

El siguiente paso, es el de asociar la fuente de los sucesos asociados al ratón el applet (**this**) con el objeto de la clase que implementa los interfaces *MouseListener* y *MouseMotionListener* y que registra dichos sucesos, que también es el applet (**this**).

```
this.addMouseListener(this);
```

```
this.addMouseListener(this);
```

Finalmente, queda definir las funciones respuesta *mousePressed* y *mouseDragged*. En la primera, se obtiene el punto inicial, cuando se pulsa el botón izquierdo del ratón.

```
public void mousePressed(MouseEvent ev) {
    uX=ev.getX();
    uY=ev.getY();
}
```

En la llamada a la función *mouseDragged*, se traza una línea entre dos puntos muy próximos. El punto previo de coordenadas *uX* y *uY*, y el punto actual que los suministra el parámetro *ev* de la clase *MouseEvent* de dicha función. Se dibuja una línea en un contexto gráfico *g*, que une los puntos actual y previo. Finalmente, el punto actual se convierte en el punto previo. El [contexto gráfico](#) *g* se obtiene mediante la función *getGraphics*.

La función *mouseDragged* se llama muchísimas veces, por lo que se crean muchos objetos *g*, que deberían ser liberados por el [recolector de basura](#) (garbage collector) al finalizar su ámbito de existencia, desde el momento en que fué creado hasta el final de la definición de dicha función. En vez de confiar en que dicho recolector libere la memoria ocupada por cada objeto *g* creado cuando se sale de la función *mouseDragged*, podemos hacerlo manualmente mediante la llamada a *dispose*.

Cuando el objeto gráfico *g* lo suministran las funciones *paint* o *update* en su único parámetro no es necesario esta llamada, la liberación de memoria la realiza automáticamente el sistema cuando dichas funciones retornan.

```
public void mouseDragged(MouseEvent ev) {
    int x = ev.getX();
    int y = ev.getY();
    Graphics g=getGraphics();
    g.drawLine(uX, uY, x, y);
    uX=x;    uY=y;
    g.dispose();
}
```

El código completo de este ejemplo, es el siguiente

```

package raton6;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet6 extends Applet
    implements MouseListener, MouseMotionListener {
    int uX=0, uY=0;

    public void init() {
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }
    //interface MouseListener
    public void mousePressed(MouseEvent ev) {
        uX=ev.getX();
        uY=ev.getY();
    }
    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}

    //interface MouseMotionListener
    public void mouseDragged(MouseEvent ev) {
        int x = ev.getX();
        int y = ev.getY();
        Graphics g=getGraphics();
        g.drawLine(uX, uY, x, y);
        uX=x;    uY=y;
        g.dispose();
    }
    public void mouseMoved(MouseEvent event) {}
}

```

2.-Creamos dos clases anónimas que derivan respectivamente de *MouseAdapter* y de *MouseMotionAdapter*



raton5: [RatonApplet5.java](#)

El Entorno Integrado JBuilder crea [una clase anónima que deriva de *MouseAdapter*](#) (Anonymous adapter) cuando seleccionamos el applet (**this**) en el modo diseño y en su correspondiente panel Events hacemos doble-clic sobre el

editor asociado a *mousePressed*. Posteriormente, asocia mediante *addMouseListener* un objeto de dicha clase anónima que maneja los sucesos asociados al ratón, con el applet (**this**) que es el componente que los genera.

```
this.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        this_mousePressed(e);
    }
});
```

Del mismo modo, se crea una clase anónima que deriva de *MouseMotionAdapter* cuando hacemos doble-clic sobre el editor asociado a *mouseDragged* en el panel Events correspondiente al applet. Posteriormente, se asocia mediante *addMouseMotionListener* un objeto de dicha clase anónima que maneja los sucesos asociados al ratón, con el applet (**this**) que es el componente que los genera.

```
this.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        this_mouseDragged(e);
    }
});
```

El resto del código consiste en definir las funciones respuesta *this_mousePressed* y *this_mouseDragged* cuyo nombre ha generado JBuilder con el fin de separar el código de inicialización del código de las funciones respuesta.

El código completo de este ejemplo, es el siguiente

```
package raton5;

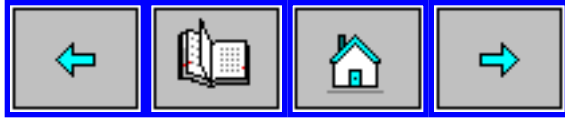
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RatonApplet5 extends Applet {
    int uX=0, uY=0;

    public void init() {
        this.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                this_mousePressed(e);
            }
        });
        this.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                this_mouseDragged(e);
            }
        });
    }
}
```

```
void this_mousePressed(MouseEvent ev) {  
    uX=ev.getX();  
    uY=ev.getY();  
}  
  
void this_mouseDragged(MouseEvent ev) {  
    int x = ev.getX();  
    int y = ev.getY();  
    Graphics g=getGraphics();  
    g.drawLine(uX, uY, x, y);  
    uX=x;    uY=y;  
    g.dispose();  
}  
}
```

Verificación de la información que se introduce en un control de edición



[Sucesos \(events\)](#)

[Introducción](#)

[El control de edición](#)

[Verificación de los datos mientras se introducen](#)

[Verificación después de introducir los datos](#)

Introducción

La mayoría de los applets disponen de controles de edición en los que se introducen datos para su procesamiento por el programa. La verificación de los datos que se introducen en los controles de edición tiene gran importancia para el correcto funcionamiento del programa.

La verificación del dato, se puede realizar mientras se teclean los caracteres o bien después de haber introducido dicho dato en el control de edición. El primer caso, se emplea para eliminar los caracteres no deseados que tecleamos sin darnos cuenta, por ejemplo letras cuando se requieren caracteres numéricos o viceversa. El segundo caso, se emplea por ejemplo, para verificar que la cantidad introducida está dentro de un intervalo apropiado.

El control de edición

Se crea el control de edición llamando a uno de sus constructores

```
TextField textEntero = new TextField();
```


Se establece el tamaño del control mediante *setColumns*

```
textEntero.setColumns(4);
```

Se puede habilitar o inhabilitar el control de edición, es decir, puede hacerse de lectura/escritura o de sólo lectura.

```
textEntero.setEditable(false);
```

Para poner texto en el control de edición, se llama a *setText*.

```
textEntero.setText("3");
```

Para obtener el texto del control de edición, se emplea *getText*.

```
String texto=textEntero.getText();
```

Verificación de los datos mientras se introducen



edicion1: [EdicionApplet1.java](#)

Propósito

Filtrar los caracteres que se introducen en el control de modo que solamente se muestren caracteres no numéricos.

Diseño

Crear un applet y situar sobre el applet en el modo diseño (pestaña **Design**) un control etiqueta (Label) y un control de edición (TextField).

Cambiar sus propiedades en sus respectivas hojas de propiedades

Establecer [FlowLayout](#) como gestor de diseño del applet

Respuesta a las acciones del usuario

Para gestionar los sucesos procedentes del teclado que se producen en un componente, se han de seguir los siguientes pasos.

- Se crea una clase que implemente el interface *KeyListener*, y que ha de definir las tres funciones declaradas en dicho interface

```
public interface KeyListener extends EventListener {
    public void keyTyped(KeyEvent e);
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
}
```

- Mediante *addKeyListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso viene encapsulada en un objeto de la clase *KeyEvent*.

En vez de implementar el interface *KeyListener*, tenemos la opción de crear una clase que derive de la clase *KeyAdapter* (véase [Listeners y adapters](#)) y redefinir algunas de sus funciones miembro por ejemplo, *keyPressed*.

```
class ValidaCaracter extends KeyAdapter{
    public void keyPressed(KeyEvent ev){
        int codigo=ev.getKeyCode();
        if(codigo>=KeyEvent.VK_0 && codigo<=KeyEvent.VK_9){
            ev.consume();
        }
    }
}
```

El objeto *ev* de la clase *KeyEvent* nos suministra código de la tecla que se ha pulsado que se extrae mediante su función miembro *getKeyCode* o el carácter tecleado mediante *getKeyChar*. Si dicho código corresponde a una tecla numérica, el suceso se consume, concluye su procesamiento, y por tanto, no se muestra en el control de edición.

```
int codigo=ev.getKeyCode();
if(codigo>=KeyEvent.VK_0 && codigo<=KeyEvent.VK_9){
    ev.consume();
}
```

```
}
```

Otra forma equivalente sería la siguiente

```
char tecla=ev.getKeyChar();
if(tecla>='0' && tecla<='9'){
    ev.consume();
}
```

Asociamos mediante *addKeyListener*, el control de edición *textIntro* en el que ocurren los sucesos relacionados con el teclado, con un objeto de la clase *ValidaCaracter* que maneja dichos sucesos.

```
ValidaCaracter valChar=new ValidaCaracter();
textIntro.addKeyListener(valChar);
```

o bien, en una sólo línea

```
textIntro.addKeyListener(new ValidaCaracter());
```

De un modo análogo, se podría definir una clase que filtrase los caracteres numéricos.

El código completo de este ejemplo, es el siguiente

```
package edicion1;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class EdicionApplet1 extends Applet {
    TextField textIntro = new TextField();
    Label label1 = new Label();
    FlowLayout flowLayout1 = new FlowLayout();

    public void init() {
        textIntro.setColumns(10);
        label1.setText("Introducir caracteres NO numéricos");
        this.setLayout(flowLayout1);
        this.add(label1, null);
        this.add(textIntro, null);
        ValidaCaracter valChar=new ValidaCaracter();
        textIntro.addKeyListener(valChar);
    }
}
```

```
}  
  
class ValidaCaracter extends KeyAdapter{  
    public void keyPressed(KeyEvent ev){  
        int codigo=ev.getKeyCode();  
        if(codigo>=KeyEvent.VK_0 && codigo<=KeyEvent.VK_9){  
            ev.consume();  
        }  
    }  
}
```

Verificación después de introducir los datos



edicion2: [EdicionApplet2.java](#)

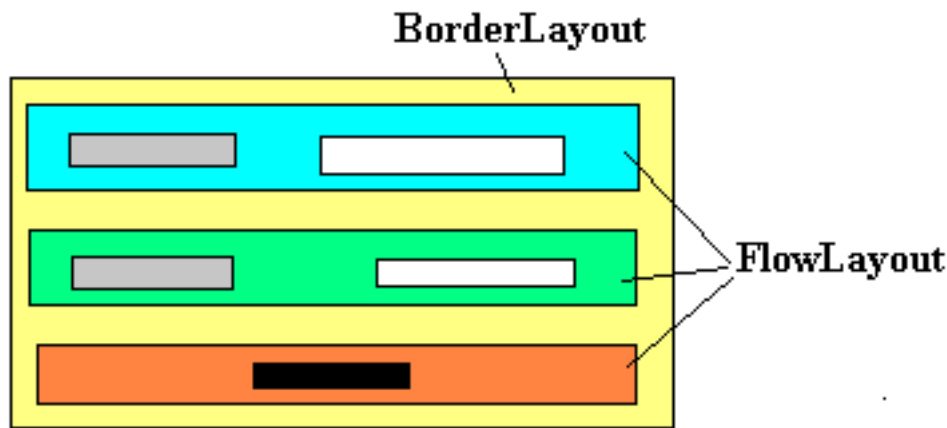
Propósito

El propósito del programa es verificar la información que ha introducido el usuario en los controles de edición. En el primer control de edición se espera que introduzca un número entero, y en el segundo un número decimal. Si se introducen caracteres no numéricos, o una coma como separador de la parte entera y decimal el control de edición recupera el foco, para que el usuario corrija los errores.

Nota: Es posible que el applet no funcione en el navegador Internet Explorer 4.0 de Microsoft como se describe, pero funciona correctamente en el AppletViewer del IDE JBuilder 2.0.

Diseño

Crear un applet, en el modo diseño (pestaña **Design**), situar tres paneles, uno en la parte superior, otro en el centro y otro en la parte inferior, tal como se indica en la figura.



- Sobre el panel superior situar una etiqueta (Label) y un control de edición (TextField)
- Sobre el panel central situar una etiqueta (Label) y un control de edición (TextField)
- Sobre el panel inferior situar un botón (Button).

Cambiar las propiedades de cada uno de los controles en sus respectivas hojas de propiedades.

Establecer [*FlowLayout*](#) como gestor de diseño de cada uno de los paneles

Establecer [*BorderLayout*](#) como gestor de diseño del applet, de modo que el panel superior quede al norte (NORTH), el central en el centro (CENTER) y el inferior en el sur (SOUTH).

Nota acerca de los paneles:

Cuando se coloca un panel AWT sobre el applet en el modo diseño, se hace invisible para el programador. Este inconveniente se puede resolver de dos formas distintas:

1. Cambiar el color del fondo del panel de modo que se haga visible. Cuando el diseño este completo, se restaura el color por defecto.
2. Emplear paneles **BevelPanel** de la paleta **JBCL Containers**, estos paneles son visbles ya que tienen un borde (bevel). Cuando el diseño esté terminado reemplazamos *BevelPanel* por *Panel* en el código fuente, mediante **Search/Replace** o el botón correspondiente de la barra de herramientas, y eliminamos las sentencias **import** que hacen referencia a los controles propios de Borland.

```
import borland.jbcl.control.*;
import borland.jbcl.layout.*;
```

Respuesta a las acciones del usuario

Cuando un control de edición tiene el foco podemos introducir caracteres en dicho control, cuando pulsamos la tecla del tabulador, el foco pasa a otro control, y así sucesivamente.

Para gestionar los sucesos asociados al cambio de foco, se han de seguir los siguientes pasos.

- Se crea una clase que implemente el interface *FocusListener*, y que ha de definir las dos funciones declaradas en dicho interface

```
public interface FocusListener extends EventListener {
    public void focusGained(FocusEvent e);
    public void focusLost(FocusEvent e);
}
```

- Mediante *addFocusListener* se conecta el componente con un objeto de la clase que maneja los sucesos originados en dicho componente.
- La información acerca del suceso, viene encapsulada en un objeto de la clase *FocusEvent*.

En vez de implementar el interface *FocusListener*, tenemos la opción de crear una clase que derive de la clase *FocusAdapter* (véase [Listeners y adapters](#)) y redefinir algunas de sus funciones miembro por ejemplo *focusLost*.

Podemos verificar el dato introducido en un control de edición cuando dicho control pierde el foco. Vamos a ver dos ejemplos: verificar que se introduce un número entero y verificar que se introduce un número decimal.

Número entero

En primer lugar, creamos una clase que deriva de *FocusAdapter*, (véase [Listeners y adapters](#)) y redefinimos la función miembro *focusLost* en la que estamos interesados.

```
class ValidaInt extends FocusAdapter{
    public void focusLost(FocusEvent ev){
        TextField tEntrada=(TextField)(ev.getSource());
        try{
            Integer.parseInt(tEntrada.getText().trim());
        }catch(NumberFormatException e){
            tEntrada.requestFocus();
            tEntrada.selectAll();
        }
    }
}
```

```

    }
}

```

El objeto *ev* de la clase *FocusEvent* guarda toda la información relativa al suceso, y en particular, la fuente o [el control que lo ha generado](#), dicha información se extrae mediante la función *getSource*.

Una vez que se ha obtenido el control de edición *tEntrada* que ha generado el suceso, se obtiene el texto, un objeto de la clase *String*, que se ha introducido en dicho control mediante la función *getText*.

Posteriormente, se eliminan los espacios en blanco al principio y al final mediante la función *trim*, y [se convierte el string en un número entero](#) mediante la función *parseInt*.

Si el proceso de conversión falla, por que se han introducido caracteres que no son números, [se produce un excepción](#) del tipo *NumberFormatException*. En este caso, el control de edición vuelve a tomar el foco *requestFocus*, y se selecciona todos los caracteres introducidos para que puedan ser borrados de una sola vez pulsando la tecla RETROCESO.

```

tEntrada.requestFocus( );
tEntrada.selectAll( );

```

Asociamos, mediante *addFocusListener*, el control de edición *textEntero* en el que ocurren los sucesos relacionados con el teclado con un objeto de la clase *ValidaInt* que registra dichos sucesos.

```

ValidaInt valInt=new ValidaInt( );
textEntero.addFocusListener(valInt);

```

o bien, en una sólo línea de código

```

textEntero.addFocusListener(new ValidaInt( ));

```

Número real

Lo mismo que hemos hecho para un número entero se puede repetir para un número real. Esto es importante, ya que tenemos la costumbre de poner una coma como carácter separador de la parte entera y de la parte decimal, lo que no es admitido por el programa. Por tanto, para su buen comportamiento debemos de notificar esta circunstancia al usuario.

```

class ValidaDouble extends FocusAdapter{
    public void focusLost(FocusEvent ev){

```

```

        TextField tEntrada=(TextField)(ev.getSource());
        try{
            Double.valueOf(tEntrada.getText()).doubleValue();
        }catch(NumberFormatException e){
            tEntrada.requestFocus();
            tEntrada.selectAll();
        }
    }
}

```

Asociamos, mediante *addFocusListener*, el control de edición *textDecimal* en el que ocurren los sucesos relacionados con el teclado con un objeto de la clase *ValidaDecimal* que registra dichos sucesos.

```

ValidaDouble valDouble=new ValidaDouble();
textDecimal.addFocusListener(valDouble);

```

o bien, en una sólo línea de código

```

textDecimal.addFocusListener(new ValidaDouble());

```

El código fuente completo de este ejemplo, es el siguiente

```

package edicion2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class EdicionApplet2 extends Applet {
    TextField textEntero = new TextField();
    TextField textDecimal = new TextField();
    Button btnAceptar = new Button();
    Panel bevelPanel1 = new Panel();
    Panel bevelPanel2 = new Panel();
    Panel bevelPanel3 = new Panel();
    Label label1 = new Label();
    Label label2 = new Label();
    FlowLayout flowLayout1 = new FlowLayout();
    FlowLayout flowLayout2 = new FlowLayout();
    FlowLayout flowLayout3 = new FlowLayout();
    BorderLayout borderLayout1 = new BorderLayout();

```



```

    public void init() {
        this.setLayout(borderLayout1);
        btnAceptar.setLabel("Aceptar");
        bevelPanel3.setLayout(flowLayout3);
        bevelPanel2.setLayout(flowLayout2);
        bevelPanel1.setLayout(flowLayout1);
//número entero
        textEntero.setText("3");
        textEntero.setColumns(4);
        label1.setText("Introduce un número entero");
        ValidaInt valInt=new ValidaInt();
        textEntero.addFocusListener(valInt);

//número decimal
        textDecimal.setText("1.3");
        textDecimal.setColumns(4);
        label2.setText("Introduce un número decimal");
        ValidaDouble valDouble=new ValidaDouble();
        textDecimal.addFocusListener(valDouble);

        this.add(bevelPanel1, BorderLayout.NORTH);
        bevelPanel1.add(label1, null);
        bevelPanel1.add(textEntero, null);
        this.add(bevelPanel2, BorderLayout.CENTER);
        bevelPanel2.add(label2, null);
        bevelPanel2.add(textDecimal, null);
        this.add(bevelPanel3, BorderLayout.SOUTH);
        bevelPanel3.add(btnAceptar, null);
    }
}

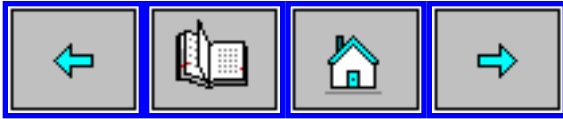
class ValidaDouble extends FocusAdapter{
    public void focusLost(FocusEvent ev){
        TextField tEntrada=(TextField)(ev.getSource());
        try{
            Double.valueOf(tEntrada.getText()).doubleValue();
        }catch(NumberFormatException e){
            tEntrada.requestFocus();
            tEntrada.selectAll();
        }
    }
}

class ValidaInt extends FocusAdapter{

```

```
public void focusLost(FocusEvent ev){
    TextField tEntrada=(TextField)(ev.getSource());
    try{
        Integer.parseInt(tEntrada.getText().trim());
    }catch(NumberFormatException e){
        tEntrada.requestFocus();
        tEntrada.selectAll();
    }
}
```

El control *Canvas* (I)



Sucesos (events)

El control *Canvas*

Seleccionar un color en una caja de selección (*Choice*)

Dibujar una circunferencia de radio variable en un canvas

El control *Canvas*

Un control *Canvas* es un control apropiado para dibujar en su superficie. El control *Canvas* define la función *paint* que pinta su superficie con el color de fondo

```
public void paint(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0, 0, width, height);
}
```

Para usar el control *Canvas* se define una clase derivada que redefina la función *paint*. En primer lugar, hemos de traer la información referente a la clase *Canvas* mediante la sentencia **import**.

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

A continuación, definimos la clase derivada de *Canvas* que denominamos *MiCanvas*, redefinimos la función miembro *paint* y definimos otras funciones miembro que consideremos necesarias.

```
public class MiCanvas extends Canvas {
    //miembros dato ...
    public MiCanvas() {
        //...
    }
    public void paint(Graphics g){
        //definir esta función
```

```

    }
//otras funciones miembro
}

```

En la clase que describe el applet, creamos el nuevo control y lo situamos en el applet como cualquier otro control.

```

public class CanvasApplet1 extends Applet {
    MiCanvas canvas;
//...
    public void init(){
        canvas=new MiCanvas();
        this.add(canvas, BorderLayout.CENTER);
        //...
    }
}

```

Por defecto, un *Canvas* no tiene tamaño, lo puede representar un problema cuando disponemos varios componentes en el applet. Por ejemplo, pueden ocurrir situaciones en las que el canvas no se vea. En este caso, es necesario redefinir las funciones miembro *getPreferredSize* y *getMinumunSize*, ambas funciones devuelven un objeto de la clase *Dimension*. En la siguiente porción de código se redefine *getPreferredSize*.

```

    public Dimension getPreferredSize(){
        return new Dimension(80, 300);
    }

```

En los ejemplos que hemos estudiado en páginas anteriores, situamos los controles en el applet y dibujamos en la superficie del mismo. Ahora bien, es una práctica habitual, situar los controles sobre un panel en el applet y dibujar sobre un canvas tal como vamos a ver en los ejemplos que se estudian en esta página.

Seleccionar un color y pintar el canvas de dicho color.



canvas1: [CanvasApplet1.java](#), [MiCanvas.java](#)



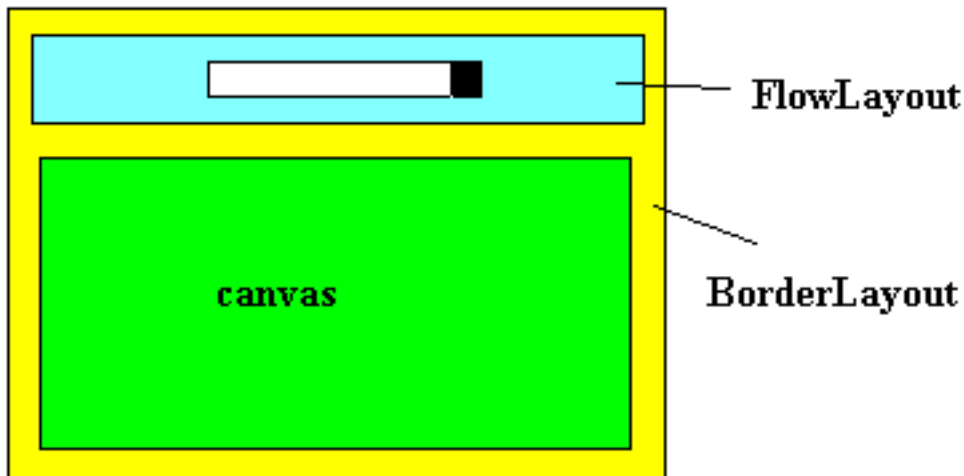
Propósito

Volvemos sobre el ejemplo ya estudiado en el que se [selecciona un color en un control selección \(Choice\)](#). Ahora vamos pintar el canvas del color seleccionado.

La clave del programa estriba en la comunicación entre dos objetos el applet y el canvas. La selección de un elemento del control selección en el applet se refleja en un cambio en el color del canvas.

Diseño

Crear el applet, en modo de diseño (pestaña **Design**) situar un panel en la parte superior del applet y un control selección (*Choice*) sobre el panel.

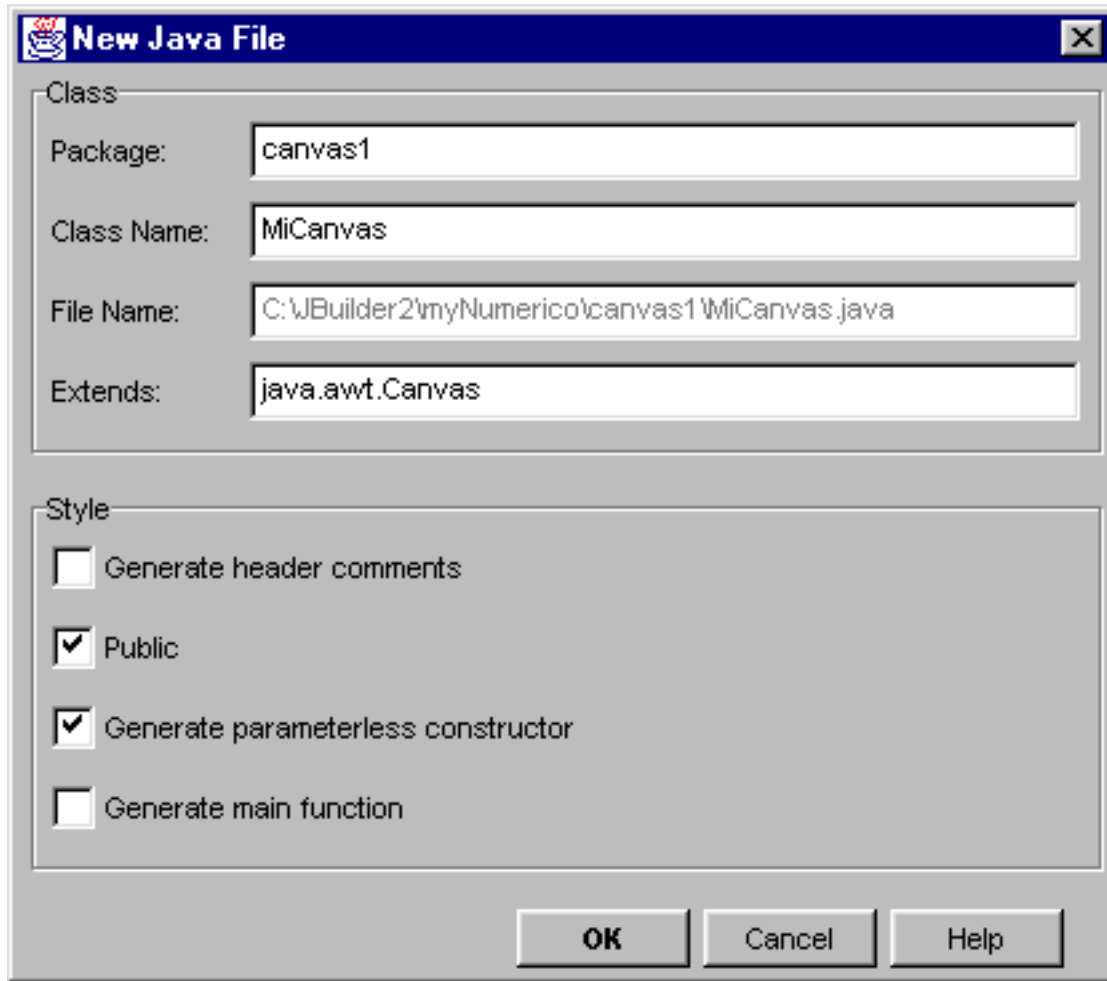


Establecer [FlowLayout](#) como gestor de diseño del panel, de modo que el control queda centrado en el panel.

Establecer [BorderLayout](#) como gestor de diseño del applet, de modo que el panel quede al norte (NORTH)

Añadir al proyecto una clase derivada de *Canvas* denominada *MiCanvas*

Para crear una clase derivada de *Canvas* en el Entorno Integrado de Desarrollo (IDE) de JBuilder, una vez creado el proyecto, se selecciona **File/New** y a continuación se selecciona el icono **Class** en el diálogo **New**. Cuando se pulsa OK aparece el diálogo **New Java File** para la creación de una nueva clase. En el campo **Class name** se introduce el nombre de la clase, en este caso *MiCanvas*, y en el campo **Extends** la clase de la cual deriva, por defecto es *java.lang.Object*, que sustituimos por *java.awt.Canvas*.



En el modo código fuente (pestaña **Source**), crear un objeto de la clase *MiCanvas* y añadir dicho objeto al applet en la posición central (CENTER), véase el apartado anterior.

Llenar el control selección (*Choice*) con los elementos siguientes : rojo, verde, azul

Crear un array *colores* de objetos de la clase [Color](#), con los colores primarios.

```
Color[] colores={Color.red, Color.green, Color.blue};
```

Redefinir en la clase *MiCanvas* la función *paint* para que pinte todo el canvas del color seleccionado.

La clave del programa está como se ha mencionado, en la comunicación entre los dos objetos: el applet y el canvas.

Respuesta a las acciones del usuario

Seleccionamos el control selección. En el panel Events hacemos doble-clic sobre el editor asociado a *itemStateChanged*. JBuilder genera el nombre de la función respuesta para que solamente tengamos que introducir el código que define la tarea a realizar

En el código de la función respuesta obtenemos el índice del color seleccionado en el control selección, y se lo pasamos a la función *setColor* miembro de la clase *MiCanvas*. Alternativamente, se podría pasar un objeto de la clase *Color*.

```
public class CanvasApplet1 extends Applet {
//...
    void lista_itemStateChanged(ItemEvent e) {
        int indice=lista.getSelectedIndex();
        canvas.setColor(indice);
    }
}
```

En la clase *MiCanvas* se define la función miembro *setColor* y se redefine la función *paint*, para que pinte todo el canvas del color seleccionado. Desde *setColor* se llama a la función *paint*. Dentro de esta última, la función *getSize* obtiene el tamaño del canvas, el miembro *width* proporciona la anchura, y el miembro *height* la altura.

```
package canvas1;

import java.awt.*;

public class MiCanvas extends Canvas {
    final Color[] colores={Color.red, Color.green, Color.blue};
    int indice=0;
    public MiCanvas() {
        setBackground(colores[0]);
    }
    void setColor(int indice){
        this.indice=indice;
        repaint();
    }
    public void paint(Graphics g){
        g.setColor(colores[indice]);
        g.fillRect(0, 0, getSize().width, getSize().height);
    }
}
```

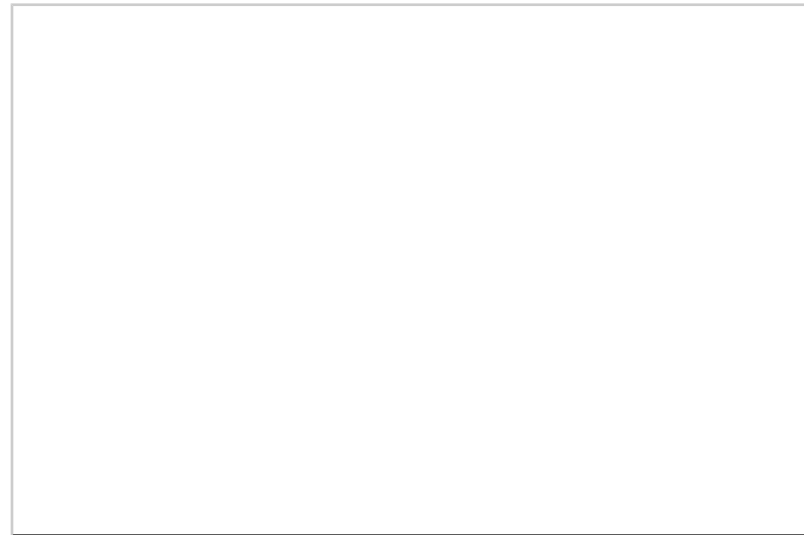
```
}  
}  
}
```

Cada vez que se selecciona un color en la caja de selección se llama a la función respuesta *lista_itemStateChanged* miembro de la clase que describe el applet. Esta llama a *setColor* miembro de la clase que describe el canvas, y finalmente, ésta llama a *paint* para pintar el canvas.

Dibujar una circunferencia de radio variable en un canvas.



canvas2: [CanvasApplet2.java](#), [MiCanvas.java](#)



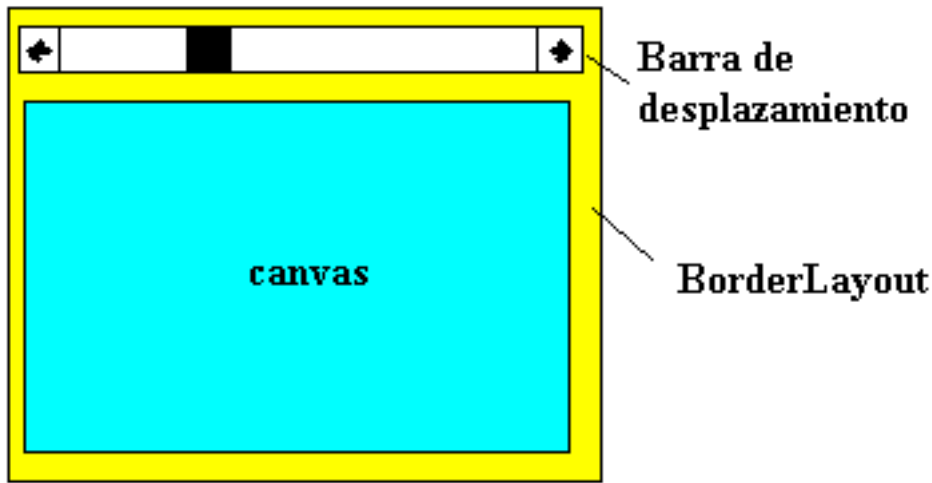
Propósito

Se ha estudiado ya el [control barra de desplazamiento](#), y cómo responder a las acciones del usuario sobre el dedo, las flechas o las regiones comprendidas entre el dedo y las flechas. En vez de dibujar la circunferencia en el applet dibujaremos la circunferencia en un canvas.

Se sugiere al lector que elabore por sí mismo el diseño y la respuesta a las acciones del usuario sobre la barra de desplazamiento, y establezca la comunicación entre el applet y el canvas, de modo que los cambios en la posición del dedo en la barra de desplazamiento se reflejen en el radio de la circunferencia que se dibuja en el canvas.

Diseño

Crear el applet, en modo de diseño (pestaña **Design**) situar una barra de desplazamiento en la parte superior del applet.



Establecer [*BorderLayout*](#) como gestor de diseño del applet, de modo que la barra de desplazamiento quede al norte (NORTH)

Añadir al proyecto una clase derivada de *Canvas* denominada *MiCanvas*

En el modo código fuente (pestaña **Source**), crear un objeto de la clase *MiCanvas* y añadir dicho objeto al applet en la posición central (CENTER).

Redefinir en la clase *MiCanvas* la función *paint* para que dibuje una circunferencia centrada en el canvas..

Respuesta a las acciones del usuario

En modo diseño, se selecciona el control barra de desplazamiento y en su correspondiente pane Event se hace doble-clic sobre el editor asociado a *adjustmentValueChanged*. En la función respuesta le pasamos la posición del dedo en la barra de desplazamiento, es decir, el radio de la circunferencia al control *canvas* mediante la llamada a la función miembro *setRadio* de la clase *MiCanvas*.

```
public class CanvasApplet2 extends Applet {
//...
void sbRadio_adjustmentValueChanged(AdjustmentEvent ev) {
    canvas.setRadio(ev.getValue());
}
}
```

En la clase *MiCanvas* se define la función miembro *setRadio* y se redefine la función *paint*, para que pinte la circunferencia con el radio especificado. Desde *setRadio* se llama a la función *paint*.

El centro de la circunferencia está situado en el centro del canvas. Obtenemos el tamaño del canvas de la misma forma que lo obtuvimos para el applet, mediante *getSize*. El miembro *width* proporciona la anchura del canvas, y el miembro *height* su altura.

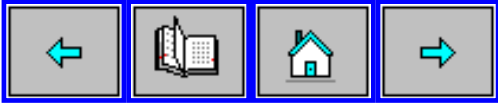
```
package canvas2;

import java.awt.*;

public class MiCanvas extends Canvas {
    int radio=10;
    public MiCanvas() {
        setBackground(Color.white);
    }
    void setRadio(int radio){
        this.radio=radio;
        repaint();
    }
    public void paint(Graphics g){
        int x1=getSize().width/2;
        int y1=getSize().height/2;
        g.setColor(Color.black);
        g.drawOval(x1-radio, y1-radio, 2*radio, 2*radio);
    }
}
```

Cada vez que se actúa sobre el dedo en la barra de desplazamiento, sobre las flechas situadas en los extremos de la barra, o en las dos regiones comprendidas entre las flechas y el dedo, se se llama a la función respuesta *sbRadio_adjustmentValueChanged* que establece el valor del radio, leyendo la posición del dedo en la barra de desplazamiento, a continuación se llama *setRadio* miembro de la clase que describe el canvas, y finalmente, ésta llama a *paint* para pintar la circunferencia centrada en el canvas con el radio especificado.

El control *Canvas* (II)



Sucesos (events)

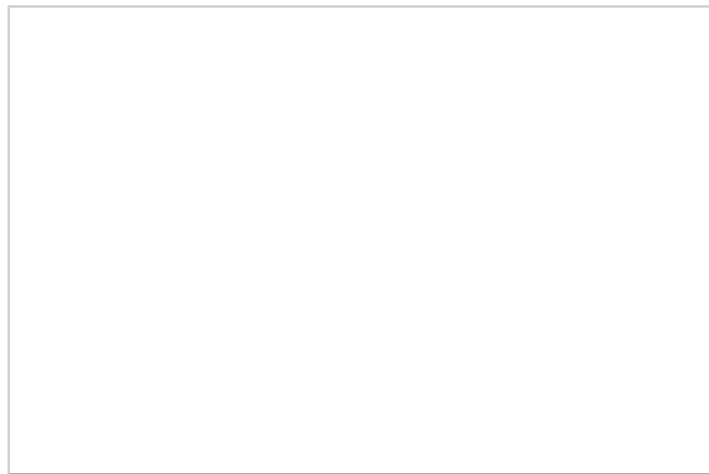
Dibujar un punto pulsando el botón izquierdo del ratón

Dibujar "a mano alzada"

Dibujar un punto pulsando el botón izquierdo del ratón



canvas3: [CanvasApplet3.java](#), [MiCanvas.java](#)



Propósito

Volvemos a estudiar el ejemplo en el que se [dibujaban pequeños círculos al pulsar el botón izquierdo del ratón](#), y se guardaban en memoria en un array. Ahora se trata de dibujar los puntos no en el applet sino en un control canvas.

Diseño

Crear el applet y establecer [BorderLayout](#) como gestor de diseño

Crear una clase *MiCanvas* derivada de *Canvas*

Crear un objeto *canvas* de la clase *MiCanvas* y situarlo en el centro (CENTER) de applet.

Respuesta a las acciones del usuario

En modo diseño (pestaña **Design**), seleccionamos el objeto canvas en el panel de componentes y en el panel Events situado a la derecha hacemos doble-clic sobre el editor asociado a *mousePressed*. JBuilder genera el código tal como se ha explicado en las páginas anteriores, y de la forma que se indica en el listado

Como vemos, se asocia el control *canvas*, donde se van a producir los sucesos relacionados con el ratón, con un objeto de una clase anónima que deriva de la clase *MouseAdapter* y que redefine la función *mousePressed*.

La función respuesta *canvas_mousePressed*, llama a la función *dibujaPunto* miembro de la clase *MiCanvas* y le pasa las coordendas del punto de la superficie del *canvas* donde se ha pulsado el botón izquierdo del ratón. La abscisa *x*, y la ordenada y del punto están encapsuladas en un objeto de la clase *Point*.

```
public class CanvasApplet3 extends Applet {
    MiCanvas canvas;
    BorderLayout borderLayout1 = new BorderLayout();

    public void init() {
        canvas=new MiCanvas();
        this.setLayout(borderLayout1);
        this.add(canvas, BorderLayout.CENTER);
        canvas.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                canvas_mousePressed(e);
            }
        });
    }

    void canvas_mousePressed(MouseEvent ev) {
        canvas.dibujaPunto(ev.getPoint());
    }
}
```

La definición de la clase que describe el canvas, *MiCanvas* es similar a la estudiada en ejemplos precedentes. El código de las funciones *dibujaPunto* y de la redefinición de *paint*, es similar al ejemplo estudiado [dibujar los puntos y guardarlos en memoria](#).

Recordaremos que cuando el [contexto gráfico](#) *g* se obtiene mediante la función *getGraphics* es necesario liberar los recursos asociados a dicho contexto mediante llamada a la función *dispose*.

```
package canvas3;

import java.awt.*;

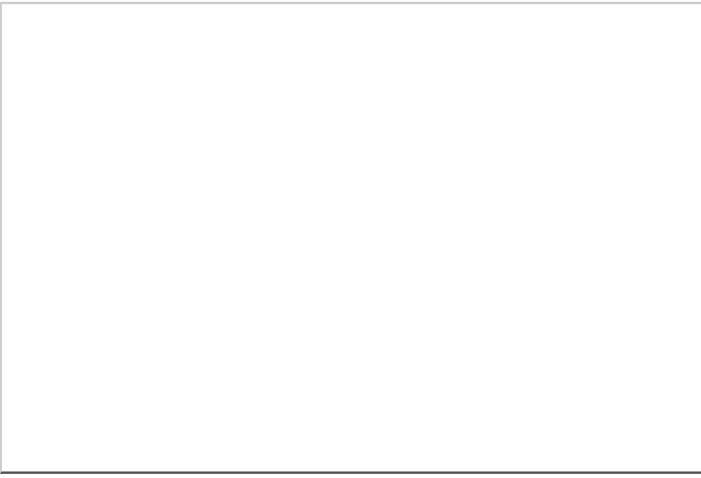
public class MiCanvas extends Canvas {
    final int MAXPUNTOS=20;
    Point puntos[]=new Point[MAXPUNTOS];
    int nPuntos=0;

    public MiCanvas() {
        setBackground(Color.white);
    }

    void dibujaPunto(Point p){
        Graphics g=getGraphics();
        g.setColor(Color.red);
        if(nPuntos<MAXPUNTOS){
            puntos[nPuntos]=p;
            nPuntos++;
        }
        g.fillOval(p.x-4, p.y-4, 8, 8);
        g.dispose();
    }

    public void paint(Graphics g){
        g.setColor(Color.blue);
        for(int i=0; i<nPuntos; i++){
            g.drawOval(puntos[i].x-4, puntos[i].y-4, 8, 8);
        }
    }
}
```

Dibujar "a mano alzada"



Propósito

Como vimos ya en el ejemplo [un simple programa de dibujo "a mano alzada"](#), tenemos dos posibles aproximaciones para resolver este problema:

1. Que la clase que describe el canvas, *MiCanvas*, [implemente los interfaces *MouseListener* y *MouseMotionListener*](#), y defina todas las funciones declaradas en dichos interfaces aunque solamente estemos interesados en alguna de ellas.
2. Crear dos clases anónimas y relacionar el objeto canvas (la fuente de los sucesos asociados al ratón) con [objetos de dos clases anónimas](#) que deriven de *MouseAdapter* y de *MouseMotionAdapter*.

Diseño

El mismo del ejemplo anterior

Respuesta a las acciones del usuario

Se sugiere al lector que trate de resolver por sí mismo este ejercicio, tomando como base el [estudio de los sucesos asociados al ratón](#)

1.-La clase que describe el canvas implementa los interfaces *MouseListener* y *MouseMotionListener*



canvas4: [CanvasApplet4.java](#), [MiCanvas.java](#)

En el constructor de la clase *MiCanvas* asociamos mediante *addMouseListener* y *addMouseMotionListener*, el productor de los sucesos asociados al ratón, el canvas (**this**) con el objeto (**this**) de la clase que implementa los interfaces *MouseListener* y *MouseMotionListener*, y que maneja dichos sucesos definiendo las funciones declaradas en dichos interfaces.

El código de la clase *MiCanvas*, sería el siguiente

```
package canvas4;

import java.awt.*;
import java.awt.event.*;

public class MiCanvas extends Canvas implements
    MouseListener, MouseMotionListener{

    int uX, uY;
    public MiCanvas() {
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }
    //interface MouseListener
    public void mousePressed(MouseEvent ev) {
        uX=ev.getX();
        uY=ev.getY();
    }

    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    //interface MouseMotionListener
    public void mouseDragged(MouseEvent ev) {
        int x = ev.getX();
        int y = ev.getY();
        Graphics g=getGraphics();
        g.drawLine(uX, uY, x, y);
        uX=x;    uY=y;
        g.dispose();
    }
    public void mouseMoved(MouseEvent event) {}
}
```

2.-Creamos dos clases anónimas que derivan respectivamente de *MouseAdapter* y de *MouseMotionAdapter*



canvas5: [CanvasApplet5.java](#), [MiCanvas.java](#)

Definimos la clase *MiCanvas* derivada de *Canvas*, creamos un objeto canvas de dicha clase y lo situamos en el centro del applet. En modo diseño, seleccionamos el objeto canvas en el panel de componentes y en el panel Events situado a la derecha hacemos doble-clic sobre el editor asociado a *mousePressed* y *mouseDragged*. JBuilder [genera el código](#) tal

como se ha explicado en las páginas anteriores.

Como vemos en el listado, se asocia el control *canvas*, donde se van a producir los sucesos relacionados con el ratón, con dos objetos de dos clases anónimas que derivan, respectivamente de las clases *MouseAdapter* y *MouseMotionAdapter*, y que redefine las funciones *mousePressed* y *mouseDragged*.

```
public class CanvasApplet5 extends Applet {
    MiCanvas canvas;
    BorderLayout borderLayout1 = new BorderLayout();

    public void init() {
        canvas=new MiCanvas();
        this.setLayout(borderLayout1);
        this.add(canvas, BorderLayout.CENTER);
        canvas.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                canvas_mousePressed(e);
            }
        });
        canvas.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                canvas_mouseDragged(e);
            }
        });
    }

    void canvas_mouseDragged(MouseEvent e) {
        canvas.dibujaCurva(e.getX(), e.getY());
    }

    void canvas_mousePressed(MouseEvent e) {
        canvas.puntoInicial(e.getX(), e.getY());
    }
}
```

Desde la función respuesta *canvas_mousePressed* se llama a la función miembro *puntoInicial* de la clase que describe el canvas, *MiCanvas*, y le pasa las coordendas del punto del canvas donde se ha pulsado el botón izquierdo del ratón.

Desde la función respuesta *canvas_mouseDragged* se llama a la función miembro *dibujaCurva* de la clase que describe el canvas, *MiCanvas*, y le pasa las coordendas del punto del canvas donde está situado el puntero del ratón a medida que se va arrastrando.

La definición de la clase *MiCanvas*, es la siguiente

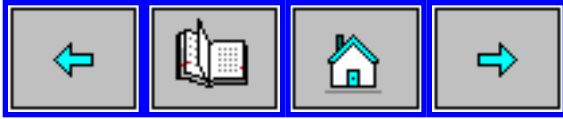

```
package canvas5;

import java.awt.*;

public class MiCanvas extends Canvas{
    int uX, uY;
    public MiCanvas() {
    }
    public void puntoInicial(int x, int y) {
        uX=x;
        uY=y;
    }

    public void dibujaCurva(int x, int y) {
        Graphics g=getGraphics();
        g.drawLine(uX, uY, x, y);
        uX=x;    uY=y;
        g.dispose();
    }
}
```

Sucesos, componentes, interfaces y funciones respuesta



Sucesos (events)

En las siguientes tablas, se resume todo lo que se ha estudiado en páginas previas. Estas tablas nos servirán de referencia para programar las respuestas a las acciones del usuario o el sistema sobre los diversos componentes. Recuerdese que es necesario seguir los pasos siguientes:

- Crear una clase que implemente el interface (o los interfaces), y definir en ella todas las funciones declaradas en el interface(s)
- Asociar el (los) componente(s) con un objeto de dicha clase que maneja los sucesos provenientes de los componentes.
- La información asociada a un suceso particular está encapsulada en un objeto de una clase derivada de *Event* cuyo nombre depende del tipo de suceso.

Para cada tipo de suceso hay una clase separada que lo describe. Las clases son de dos tipos:

- Las que corresponden a sucesos de bajo nivel que puede recibir cualquier componente

ComponentEvent, FocusEvent, KeyEvent, MouseEvent.

- Las que corresponden a sucesos de más alto nivel (semantic events) que solamente tienen significado para ciertos componentes .

ActionEvent, AdjustmentEvent, ItemEvent

Un suceso de alto nivel (semantic) está compuesto por varios sucesos de bajo nivel. Por ejemplo, pulsar el botón izquierdo del ratón y liberarlo, son dos sucesos de bajo nivel. Ambos generan un clic del ratón que es a su vez un suceso de bajo nivel. Cuando dicho suceso ocurre sobre un botón se genera un suceso de alto nivel descrito por la clase *ActionEvent*. Cuando se pulsa con el ratón sobre un elemento de un control lista se genera un suceso descrito por la clase *ItemEvent*.

Los sucesos de alto nivel que tienen significado para los componentes que se especifican se muestran en la tabla 1

Tabla 1.

| Suceso (Event) | Componente | Acción |
|------------------|-----------------------------|---|
| ActionEvent | Button List TextField | Pulsar sobre el botón Hacer doble-clic sobre un elemento de la lista Pulsar la tecla retorno (Enter) |
| AdjustementEvent | Scrollbar | Cualquier acción sobre la barra de desplazamiento |
| ItemEvent | Choice List Checkbox | Seleccionar un elemento Seleccionar o deseleccionar un elemento de la lista Activar o desactivar la casilla de verificación |

Para cada tipo de suceso la aplicación puede añadir objetos interesados (listeners) en dichos sucesos. Las clases que describen dichos objetos implementan interfaces. Cuando el suceso ocurre se llama a los métodos del interface que implementa la clase. En la Tabla 2 se recogen los interfaces para cada tipo de suceso.

Por ejemplo, un objeto (listener) interesado en los sucesos provenientes de un botón se describe mediante una clase que implementa el interface *ActionListener* y define la función *actionPerformed*. Cuando se pulsa el botón, dicho objeto llama a la función *actionPerformed* y le pasa un objeto de la clase *ActionEvent* que contiene la información relativa al suceso generado.

Como vemos hay una correspondencia entre el nombre del suceso y el nombre del interface, excepto para *MouseEvent*, que hay dos interfaces para el mismo suceso por razones de eficiencia.

Tabla 2.

| Suceso (Event) | Interface (Listener) | Método |
|------------------|----------------------|---------------------------------------|
| ActionEvent | ActionListener | actionPerformed |
| AdjustementEvent | AdjustementListener | adjustementValueChanged |
| FocusEvent | FocusListener | focusGained focusLost |
| ItemEvent | ItemListener | itemStateChanged |
| KeyEvent | KeyListener | keyTyped keyPressed keyReleased |

| | | |
|------------|---------------------|--|
| MouseEvent | MouseListener | mouseClicked mouseEntered mouseExited mousePressed mouseReleased |
| | MouseMotionListener | mouseDragged mouseMoved |

La fuente de los sucesos mantienen una lista de objetos interesados (listeners) en los mismos. Se añaden a la lista mediante una función denominada *addXXXListener*, donde XXX es el tipo de suceso.

```
boton.addActionListener(accion);
```

La tabla 3, muestra los objetos fuente de sucesos y los tipos de objetos interesados en dichos sucesos (listeners) que se pueden añadir (add)

Tabla 3.

| Control | Interface (Listener) |
|-----------|--|
| Button | ActionListener |
| Choice | ItemListener |
| Checkbox | ItemListener |
| Component | FocusListener KeyListener MouseListener MouseMotionListener |
| List | ActionListener ItemListener |

Ejercicio 1



Sucesos (events)

Propósito

Diseño

Respuesta a las acciones del usuario

El código fuente

Propósito

En este ejercicio se trata de crear un fuente de texto, a partir de las selecciones efectuadas en tres listas cuyos elementos son:

- los nombre de las fuentes de texto
- los estilos
- los tamaños

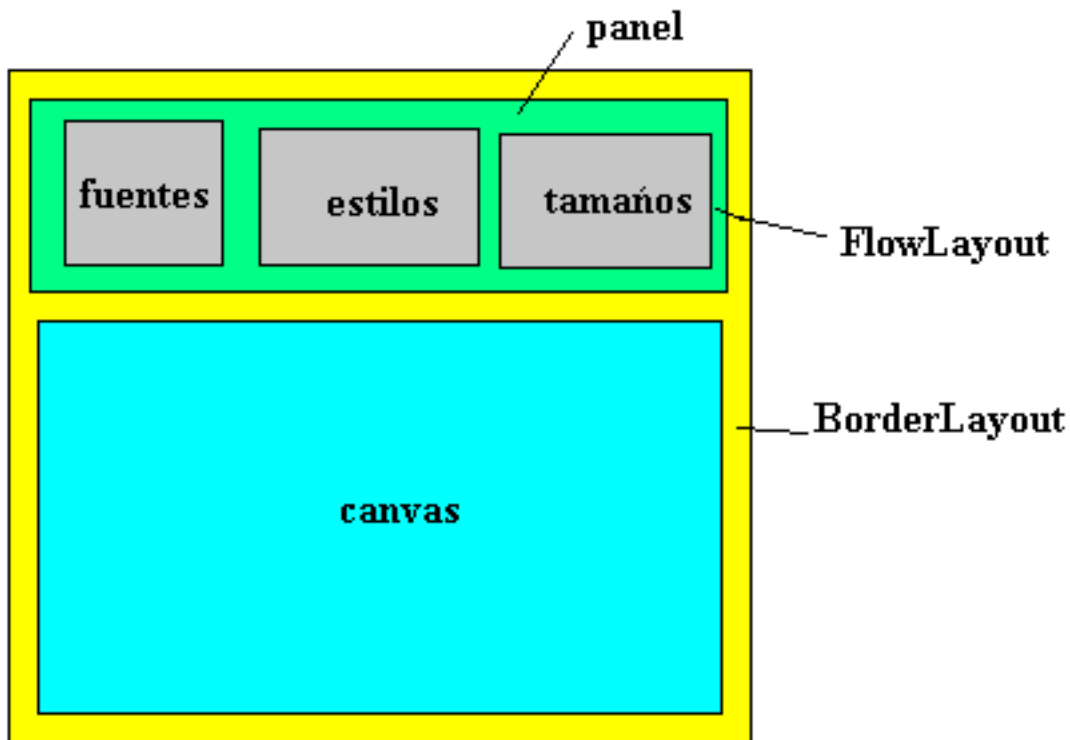
Una vez creada la fuente de texto, se establece y se dibuja un texto en el canvas.



Diseño

Crear el applet. En modo diseño (pestaña **Design**) situar en su parte superior un panel. Sobre este panel colocar tres controles lista (*List*)

Cambiar el nombre de la propiedad **name** de cada uno de los controles en sus correspondientes hojas de propiedades.



Establecer [FlowLayout](#) como gestor de diseño del panel

Establecer [BorderLayout](#) como gestor de diseño del applet, situando al panel en la posición norte (NORTH)

Crear una clase derivada de [Canvas](#) y definir la función *paint* para que dibuje un texto.

Seleccionar el modo código fuente (pestaña **Source**). Llenar las tres [listas](#) (*List*) con los elementos de los arrays: *nombresFuentes*, *nombreEstilos*, y *tamaños*.

- El [nombre de las fuentes](#) es un array de strings que se obtiene de la siguiente forma

```
String[] nombresFuentes=getToolkit().getFontList();
```

- Se crea el array de los [nombre de los estilos](#)

```
String[] nombreEstilos={"Plain", "Bold", "Italic", "Bold+Italic"};
```

Los estilos son números enteros que se guardan en los miembros estáticos de la clase *Font*:
Font.PLAIN=0, *Font.BOLD=1*, *Font.ITALIC=2*.

- Se crea el array de strings que representa los tamaños disponibles

```
String[] tamaños={"12", "14", "16", "18", "24", "36"};
```

Una vez llenado los controles lista (*List*) no debemos de olvidarnos de establecer el elemento inicialmente seleccionado, mediante *select*, en cada una de los controles lista.

Respuesta a las acciones del usuario

Definir [una respuesta única para tres controles](#) que se comportan de forma semejante

En la función respuesta a las acciones del usuario sobre los tres controles, obtener los elementos seleccionados de cada una de las tres [listas](#) (nombre o índice según se requiera).

A partir de estos datos, crear un objeto de la clase [Font](#).

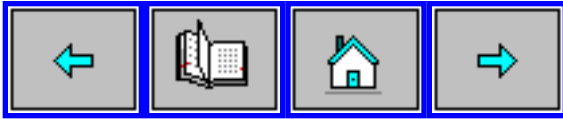
Finalmente, [comunicar el applet y el canvas](#), de modo que cada vez que cambie la selección en las listas, se dibuje un texto en el canvas con la fuente creada.

El código fuente



ejercicio1: [EjercicioApplet1.java](#), [MiCanvas.java](#)

Ejercicio 2



Sucesos (events)

[Propósito](#)

[Diseño](#)

[Respuesta a las acciones del usuario](#)

[Mejora del programa](#)

[El código fuente](#)

Propósito

En este ejercicio vamos a combinar un control de edición y una barra de desplazamiento.

- Pretendemos que cuando cambie la posición del dedo en la barra de desplazamiento, el valor al que equivale dicha posición se muestre en el control de edición.
- Por otra parte, cuando introducimos un valor en el control de edición y pulsamos la tecla Retorno, queremos que se refleje este hecho en la posición del dedo en la barra de desplazamiento.

La posición del dedo en la barra de desplazamiento, nos proporciona un conjunto de valores entre un mínimo y un máximo, que por defecto son los números enteros comprendidos entre 0 y 100 (estos valores se pueden cambiar en la hoja de propiedades de la barra de desplazamiento, en los editores asociados a las propiedades **minimum** y **maximum**).

Mejora del programa

El programa se puede mejorar si en vez de números enteros podemos introducir números reales (**double**) entre un mínimo y un máximo, por ejemplo entre 0.0 y 1.0. Para ello, tenemos que definir dos funciones que nos realicen la tarea de cambio de escala.

Por ejemplo, el valor 0.4 que introducimos en el control de edición equivale a la posición 40 del dedo en la barra de desplazamiento. La posición 90 del dedo en la barra de desplazamiento equivale al valor 0.9 en el control de edición.

Diseño

Crear el applet. En modo diseño (pestaña **Design**) situar en la parte izquierda un control de edición y en la parte derecha una barra de desplazamiento en posición horizontal.

Cambiar las propiedades de los dos controles en sus correspondientes hojas de propiedades.

Establecer [*BorderLayout*](#) como gestor de diseño del applet, de modo que el control de edición quede al oeste (WEST) y la barra de desplazamiento al centro (CENTER).

Respuesta a las acciones del usuario

1.-Definir la función respuesta a las acciones del usuario sobre un [control barra de desplazamiento](#).

La tarea de la función respuesta es que la posición del dedo en la barra de desplazamiento se muestra en el [control de edición](#).

2.-[Filtrar los caracteres](#) que se pueden introducir en el control de edición.

Solamente se permiten los caracteres numéricos, el carácter separador punto de la parte entera y decimal, las teclas de edición Retroceso y Suprimir. Los caracteres correspondientes a estas dos teclas son los siguientes:

- Tecla Retroceso: *KeyEvent.VK_DELETE*
- Tecla Suprimir: *KeyEvent.VK_BACK_SPACE*

Los caracteres se filtran en respuesta a la acción de pulsar una tecla, se define *KeyPressed* del interface *KeyListener*.

3.-Definir la función respuesta a la acción de pulsar la tecla Retorno o Enter en el control de edición.

Como hemos podido leer en la tabla 1 del [resumen](#) de este importante capítulo, cuando se pulsa Retorno en un control de edición se genera un suceso del tipo *ActionEvent*, igual que al [pulsar en un botón](#).

Las tareas de la función respuesta son las siguientes:

- Leer el dato numérico introducido en el [control de edición](#).
- Verificar que está dentro de los valores mínimo y máximo
- Mover el dedo de la [barra de desplazamiento](#) a una nueva posición.

Mejora del programa

Definir dos miembros dato denominados *minimo* y *maximo* del tipo **double**. Guardar en ellos los valores mínimo y máximo, por ejemplo 0.0 y 1.0.

Definir una función que transforme la posición del dedo (un número entero comprendido entre 0 y 100) en un número real comprendido entre *minimo* y *maximo*.

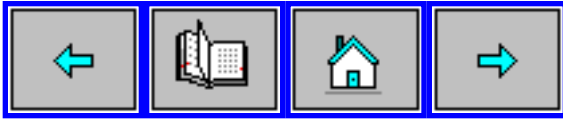
Definir una función que transforme el dato introducido en el control de edición (un número real comprendido entre *minimo* y *maximo*) en la posición del dedo (un número entero comprendido entre 0 y 100).

El código fuente



ejercicio2: [EjercicioApplet2.java](#)

La clase derivada de *Thread*



Subprocesos (threads)

El método *run*

Derivando de la clase *Thread*

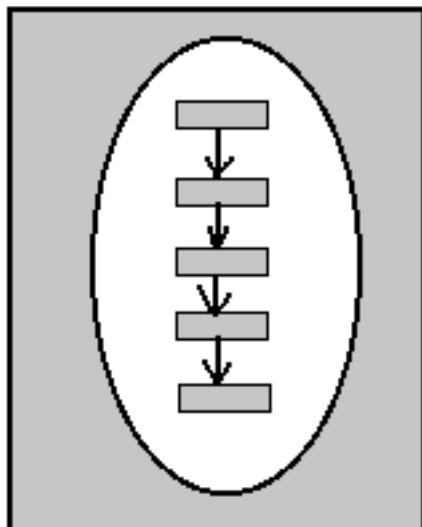
Prioridades

Control de la ejecución de los subprocesos

Todos los programadores están familiarizados con los programas secuenciales. Todos ellos tienen un principio, una secuencia de ejecución y un final. Por ejemplo, una aplicación comienza en la función *main*, ejecuta las sentencias del cuerpo de dicha función en orden consecutivo y el programa acaba cuando se llega al final de dicha función.

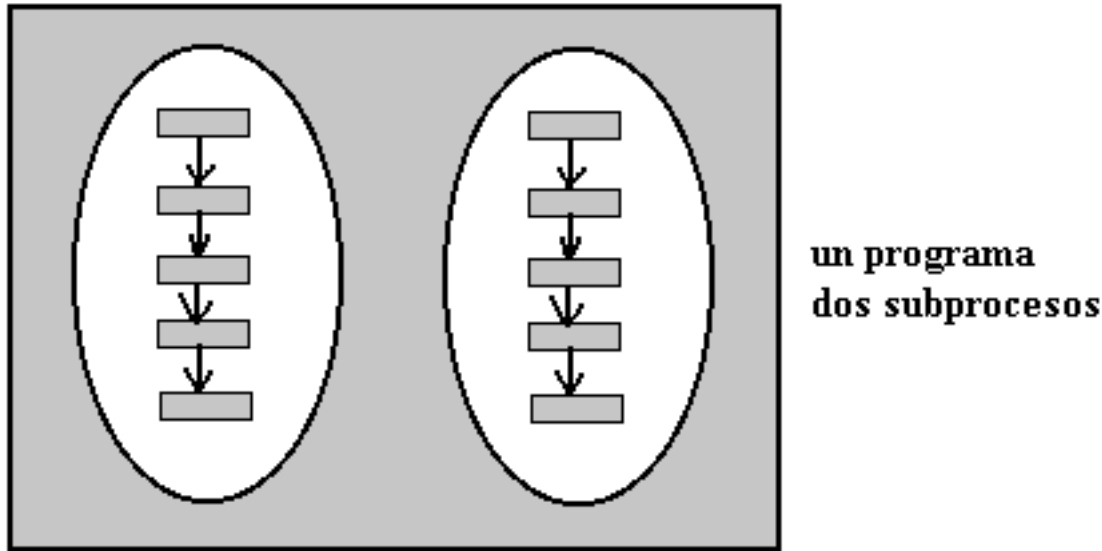
Un subproceso es similar a un programa secuencial, tiene un principio, una secuencia y un final. La diferencia fundamental estriba en que un subproceso no es un programa que pueda correr aislado, sino que se ejecuta dentro de un programa.

En la figura se muestra un subproceso que corre dentro de un programa.



un programa
un subproceso

En esta otra figura se muestran dos subprocesos corriendo dentro de un mismo programa



Todos estamos acostumbrados a manejar programas que contienen varios subprocesos en ejecución. Por ejemplo, mientras escribimos en el procesador de textos Word un subproceso se encarga de verificar la ortografía, y de señalar la palabra que no está correctamente escrita, e incluso de corregirnos según vamos escribiendo.

La ilusión de la ejecución paralela de los subprocesos en un sistema que tiene una única CPU proviene del hecho de que cada subproceso tiene la oportunidad de ejecutar una porción de código cada vez a intervalos regulares. Esta aproximación se denomina *timeslicing*. Como la CPU ejecuta millones de instrucciones por segundo, la percepción para el usuario es el de una ejecución en paralelo.

El método *run*

El método *run* es el corazón del subproceso, es donde tiene lugar la acción del subproceso. Hay dos modos de proporcionar el el método *run* a un subproceso:

- Derivando una clase de *Thread* y redefiniendo el método *run*
- [Implementando el interface *Runnable*](#) y definiendo la función *run* de dicho interface.

La razón de que existan estas dos posibilidades es que en Java no existe la herencia múltiple. Si una clase deriva de otra no podemos hacer que derive también de *Thread*. Por ejemplo, un applet deriva de la clase base *Applet* por tanto, ha de implementar el interface *Runnable* para que pueda definir el método *run*. En el estudio de la [animación](#) veremos esta aproximación.

Derivando de la clase *Thread*



thread0: [Hilo.java](#), [ThreadApp.java](#)

Creamos una [clase derivada](#) de *Thread* que redefina el método *run*.

```
public class Hilo extends Thread {
    public Hilo(String nombre) {
        super(nombre);
    }
    public void run(){
        //definir run...
    }
}
```

La clase *Thread* tiene varios constructores, además del constructor por defecto (sin argumentos). Al constructor de la clase derivada *Hilo* le pasamos el nombre del subproceso y éste se lo pasa al constructor de la clase base *Thread* mediante la palabra reservada **super**.

Creando el subproceso

Creamos dos objetos (o dos subprocesos) de la clase *Hilo*, en el cuerpo de la función *main* de la aplicación.

```
Hilo hilo1=new Hilo("Subproceso 1");
Hilo hilo2=new Hilo("Supproceso 2");
```

El nombre de cada subproceso se pasa en el único parámetro de su constructor.

El subproceso está en el estado New Thread, el subproceso está inicializado, y listo para ponerlo en marcha llamando al método *start* de la clase base *Thread*.

Poniendo en marcha el subproceso

El método *start* crea los recursos del sistema necesarios para que el subproceso se ejecute y a continuación, llama al método *run*, el subproceso se dice que está en el estado Runnable.

```
hilo1.start();
hilo2.start();
```

Corriendo el subproceso

La redefinición de la función miembro *run* en la clase derivada *Hilo*, es muy simple. Se ejecuta un bucle **for**, y dentro del bucle se imprime, el nombre del subproceso, que se obtiene mediante la función miembro *getName* de la clase *Thread* y el valor que va tomando la variable contador *i* del bucle.

```
public void run(){
    for(int i=1; i<10; i++){
        System.out.println(getName()+" : "+i);
    }
}
```

Ejecutando la aplicación

```
public class ThreadApp {
    public static void main(String[] args) {
        Hilo hilo1=new Hilo("Subproceso 1");
        Hilo hilo2=new Hilo("Supproceso 2");
        hilo1.start();
        hilo2.start();
    }
}
```

Al ejecutar la aplicación, se crea primero el objeto *hilo1*, luego el objeto *hilo2*.

El objeto *hilo1*, llama a *start*, y a continuación se llama a su función miembro *run*, que imprime el nombre del subproceso Subproceso1, y a continuación el valor de la variable contador *i*. La salida debería ser la siguiente.

```
Subproceso1: 1
Subproceso1: 2
Subproceso1: 3
Subproceso1: 4
Subproceso1: 5
Subproceso1: 6
Subproceso1: 7
Subproceso1: 8
Subproceso1: 9
```

El objeto *hilo2*, llama a *start* y a continuación a su función miembro *run*, que imprime el nombre del subproceso Subproceso2, y a continuación el valor de la variable contador *i*. La salida a continuación de la del cuadro anterior, debería ser la siguiente.

```
Subproceso2: 1
Subproceso2: 2
Subproceso2: 3
Subproceso2: 4
Subproceso2: 5
Subproceso2: 6
Subproceso2: 7
Subproceso2: 8
Subproceso2: 9
```

La salida que vemos en la consola no es la misma que se ha descrito, debido a que el sistema da la oportunidad al segundo subproceso de ejecutarse tal como se ve en la figura de la izquierda un poco más abajo.

Pausa en la ejecución del subproceso

Podemos mejorar el método *run*, introduciendo una pausa durante la ejecución del subproceso

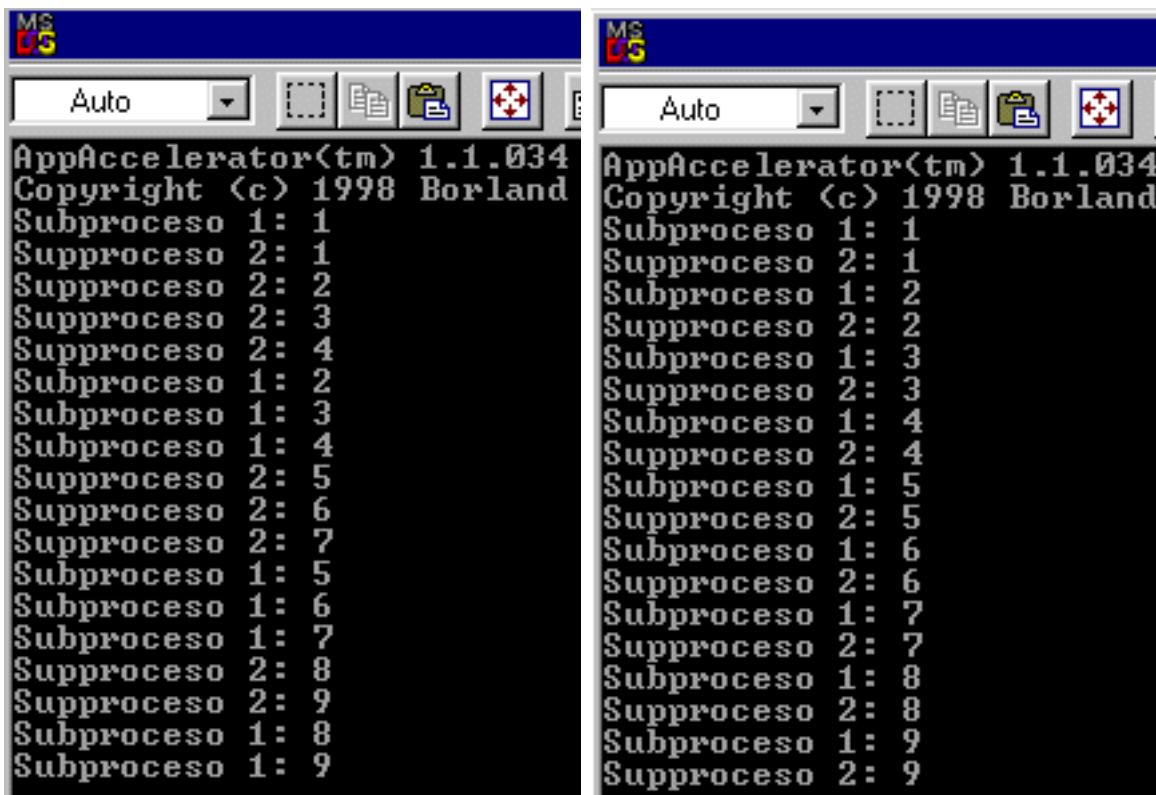
```
public void run(){
    for(int i=1; i<10; i++){
        System.out.println(getName()+" : "+i);
        try{
            sleep(100);
        }catch(InterruptedException ex){}
    }
}
```


Cuando se llama a la función *sleep*, el subproceso pasa del estado Runnable al estado Not Runnable, dando la oportunidad a otros subprocesos de ejecutarse. La función *sleep* se hereda de *Thread* y su argumento es el tiempo de pausa en milisegundos. El bloque **try..catch** que comprende al método es necesario pero no tiene por qué hacer nada específico.

Una vez que se ha llamado a la sentencia *sleep* el subproceso vuelve al estado Runnable. En general, un subproceso está en el estado Not Runnable cuando:

- Se llama al método *sleep*
- Se llama al método *suspend*
- Se llama al método *wait*, para esperar hasta que se satisfaga alguna condición
- Cuando el subproceso está bloqueado en una operación Entrada/Salida.

Cuando volvemos a correr la aplicación la salida cambia (figura de la derecha). Los subprocesos 1 y 2 se ejecutan una vez cada uno, primero el Subproceso1 y luego el Subproceso2. Para obtener este resultado, quitar los delimitadores de comentarios */*...*/* que anulan el bloque **try...catch** en la definición del método *run* de la clase *Hilo*.



```

AppAccelerator(tm) 1.1.034
Copyright (c) 1998 Borland
Subproceso 1: 1
Subproceso 2: 1
Subproceso 2: 2
Subproceso 2: 3
Subproceso 2: 4
Subproceso 1: 2
Subproceso 1: 3
Subproceso 1: 4
Subproceso 2: 5
Subproceso 2: 6
Subproceso 2: 7
Subproceso 1: 5
Subproceso 1: 6
Subproceso 1: 7
Subproceso 2: 8
Subproceso 2: 9
Subproceso 1: 8
Subproceso 1: 9

AppAccelerator(tm) 1.1.034
Copyright (c) 1998 Borland
Subproceso 1: 1
Subproceso 2: 1
Subproceso 1: 2
Subproceso 2: 2
Subproceso 1: 3
Subproceso 2: 3
Subproceso 1: 4
Subproceso 2: 4
Subproceso 1: 5
Subproceso 2: 5
Subproceso 1: 6
Subproceso 2: 6
Subproceso 1: 7
Subproceso 2: 7
Subproceso 1: 8
Subproceso 2: 8
Subproceso 1: 9
Subproceso 2: 9

```

Muerte del subproceso

Un proceso pasa al estado Death (muerto) cuando se completa su método *run*. Por ejemplo, cuando se completa el bucle **for**, se sale del bucle y se llega al final de la función *run*, el subproceso muere de muerte natural.

Un subproceso se muere cuando ya no es necesario, es decir cuando

- El método *run* finaliza su ejecución
- Se llama al método *stop* de la clase *Thread*

Un subproceso en el estado Death no puede ser revivido y ejecutado de nuevo.

```
public class Hilo extends Thread {
    public Hilo(String nombre) {
        super(nombre);
    }
    public void run(){
        for(int i=1; i<10; i++){
            System.out.println(getName()+" : "+i);
            try{
                sleep(100);
            }catch(InterruptedException ex){}
        }
    }
}

public class ThreadApp {
    public static void main(String[] args) {
        Hilo hilo1=new Hilo("Subproceso 1");
        Hilo hilo2=new Hilo("Supproceso 2");
        hilo1.start();
        hilo2.start();
    }
}
```

Prioridades



thread1: [Hilo.java](#), [ThreadApplet1.java](#)

Como hemos visto, en los ordenadores que tienen una única CPU, los subprocesos corren uno cada vez proporcionando la ilusión de procesos que se ejecutan al mismo tiempo. Se puede modificar la prioridad de los subprocesos después de su creación mediante *setPriority*. A esta función se le pasa un entero comprendido entre dos valores mínimo y máximo. Cuando mayor sea el entero, mayor será la prioridad con la que se ejecuta el correspondiente subproceso.

La clase *Thread* define tres constantes que representan los niveles de prioridad relativos para los subprocesos. Tomando 1 el valor de la mínima prioridad y 10 el valor de la máxima prioridad.

- MIN_PRIORITY
- MAX_PRIORITY
- NORM_PRIORITY

Un ejemplo de subproceso de baja prioridad es el que libera la memoria no usada que se ejecuta en la [Máquina Virtual Java](#). Aún cuando la liberación de memoria es una tarea muy importante, su baja prioridad evita que el procesador esté ocupado demasiado tiempo en ella, dejando a los procesos críticos el uso prioritario de la CPU. Esto no significa que en un momento dado se pueda agotar toda la memoria y se bloquee el sistema, ya que cuando la memoria se agota, los subprocesos críticos entran en estado de espera, dejando a la CPU que ejecute el subproceso que libera la memoria no usada.

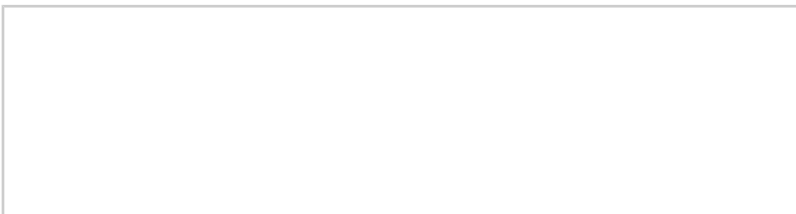
Para obtener el nivel de prioridad de un subproceso se usa la función *getPriority*

```
int prioridad=hilo1.getPriority();
```

Teniendo en cuenta que la prioridad es una propiedad relativa, se puede cambiar la prioridad de un subproceso respecto de otro aumentando o disminuyendo su valor de prioridad, por ejemplo,

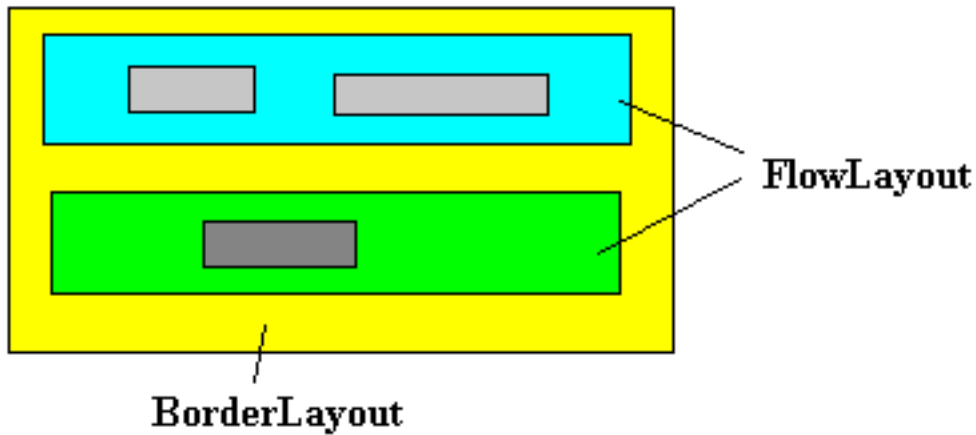
```
hilo2.setPriority(hilo1.getPriority()+1);
```

En el ejemplo anterior, podemos observar la prioridad, siempre que el tiempo de pausa, el argumento de la función *sleep*, sea pequeño (menor que 5). No obstante, vamos a crear un applet en el que podamos observar visualmente el efecto de la prioridad de dos subprocesos.



Diseño

Crear el applet, en modo de diseño (pestaña **Design**) situar dos paneles, uno en la parte superior y otro en la parte inferior. Sobre el panel superior poner dos controles de edición (*TextField*), sobre el panel inferior poner un botón (*Button*).



Cambiar las propiedades de los controles en sus respectivas hojas de propiedades

Establecer [*FlowLayout*](#) como gestor de diseño de los paneles, de modo que queden centrados en el panel y suficientemente separados horizontalmente.

Establecer [*BorderLayout*](#) como gestor de diseño del applet, de modo que el panel superior quede al norte (NORTH) y el inferior al centro (CENTER) o al sur (SOUTH).

Crear una clase denominada *Hilo*, semejante a la del apartado anterior, sustituyendo la sentencia que imprime un texto en la consola por la sentencia que escribe en un [control de edición](#). Naturalmente, dentro de la clase *Hilo* se tiene que tener acceso al control de edición pasándoselo en su constructor. Para observar las prioridades, se emplea un bucle **for** dentro de la función miembro *run* de 4000 o 5000 pasos, y se pone un tiempo de pausa en la función *sleep* pequeño de 1 ó 2.

Respuesta a las acciones del usuario

En la función respuesta a la acción de pulsar sobre el botón:

- Crear dos subprocesos mediante **new**
- Establecer la prioridad de cada subproceso llamando a *setPriority*
- Ponerlos en marcha llamando a la función *start*.

Comentarios

Puede entenderse fácilmente la clase *Hilo*, la única diferencia es que tiene un miembro dato que es el

control de edición en el cual se va a mostrar el valor de la variable contador i del bucle **for**.

```
import java.awt.*;

public class Hilo extends Thread {
    private TextField contador;
    public Hilo(TextField contador, String nombre) {
        super(nombre);
        this.contador=contador;
    }
    public void run(){
        for(int i=1; i<5000; i++){
            contador.setText(String.valueOf(i));
            try{
                sleep(1);
            }catch(InterruptedException ex){}
        }
    }
}
```

La definición de la función respuesta a la acción de pulsar el botón titulado Empieza es la siguiente.

```
void btnEmpieza_actionPerformed(ActionEvent e) {
    Hilo hilo1=new Hilo(tTexto1, "Subproceso 1");
    Hilo hilo2=new Hilo(tTexto2, "Subproceso 2");
    hilo1.setPriority(Thread.MAX_PRIORITY);
    hilo2.setPriority(Thread.MIN_PRIORITY);
    hilo1.start();
    hilo2.start();
}
```

El Subproceso 1 tiene la máxima prioridad (está a la izquierda en el applet), y el Subproceso 2 tiene mínima prioridad (está a la derecha en el applet). Como vemos, el subproceso de más alta prioridad no anula al subproceso de más baja prioridad. El sistema escoge primero al de más alta prioridad, pero da alguna oportunidad al de más baja prioridad de ejecutarse.

Se sugiere al lector cambiar las prioridades relativas de los subprocesos, y en cada uno de los casos observar el efecto del cambio en el tiempo de pausa, (argumento de la función *sleep*).

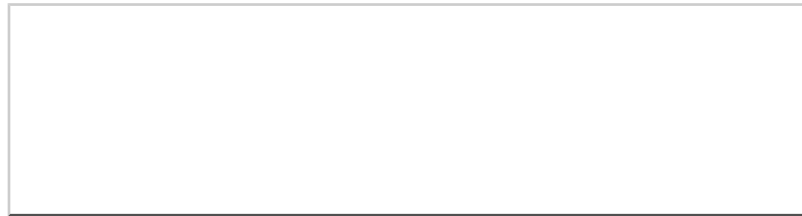
Control de la ejecución de los subprocesos



thread3: [Hilo.java](#), [ThreadApplet3.java](#)

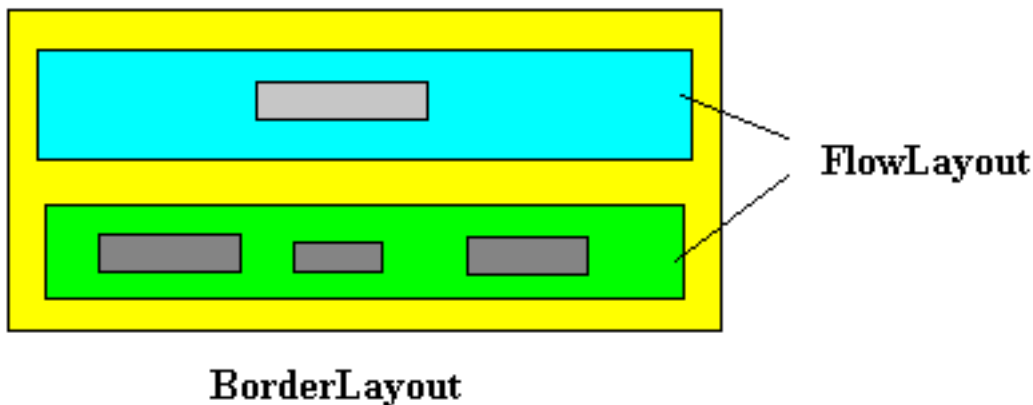
Vamos a crear un applet en el que corra un subproceso:

- El subproceso se pone en marcha pulsando en el botón titulado Empieza
- Se pone en estado de espera pulsando en el botón Pausa, se reanuda su ejecución pulsando en el mismo botón que ahora se titula Continua.
- Se para o mata el subproceso pulsando en el botón titulado Parar.



Diseño

Crear el applet, en modo de diseño (pestaña **Design**) situar dos paneles, uno en la parte superior y otro en la parte inferior. Sobre el panel superior poner un control de edición (*TextField*), sobre el panel inferior poner tres botones (*Button*).



Cambiar las propiedades de los controles en sus respectivas hojas de propiedades

Establecer [FlowLayout](#) como gestor de diseño de los paneles, de modo que queden centrados en el panel y suficientemente separados horizontalmente.

Establecer [BorderLayout](#) como gestor de diseño del applet, de modo que el panel superior quede al norte

(NORTH) y el inferior al centro (CENTER) o al sur (SOUTH).

Crear un clase *Hilo* semejante a la estudiada en el apartado anterior, sustituyendo el bucle **for** por un bucle sin fin, **while(true)**. La clase *Hilo* tendra como miembro dato la variable contador *i*, cuyo valor inicial cero se establece en el constructor

```
import java.awt.*;

public class Hilo extends Thread {
    private TextField contador;
    private int i=0;
    public Hilo(TextField contador, String nombre) {
        super(nombre);
        this.contador=contador;
        i=0;
    }
    public void run(){
        while(true){
            i++;
            contador.setText(String.valueOf(i));
            try{
                sleep(200);
            }catch(InterruptedException ex){}
        }
    }
}
```

Funciones respuesta

En modo diseño (pestaña **Design**), haciendo doble-clic sobre cada uno de los botones se genera el nombre de la función respuesta.

En la función respuesta a la acción de pulsar el botón titulado Empieza, se crea el subprocesso y se pone en marcha, tal como se ha visto en el apartado anterior.

```
void btnEmpezar_actionPerformed(ActionEvent e) {
    btnPausa.setEnabled(true);
    btnParar.setEnabled(true);
    btnPausa.setLabel("  Pausa  ");
    bPausa=true;
}
```

```

        if(hilo1==null){
            hilo1=new Hilo(tTexto1, "Subproceso 1");
            hilo1.start();
        }
    }

```

En la función respuesta a la acción de pulsar en el botón Pausa, se llama a dos funciones de la clase *Thread*: *suspend* y *resume*. La primera pone el subproceso en estado Not Runnable hasta que una llamada a *resume* lo pone de nuevo en estado Runnable. Según sea el valor **true** o **false** de un miembro dato *bPausa* de tipo **boolean** se cambia el título del botón y se llama a *suspend* o a *resume*, tal como se ve en la definición de la función respuesta.

```

void btnPausa_actionPerformed(ActionEvent e) {
    if(bPausa==true){
        hilo1.suspend();
        btnPausa.setLabel("Continua");
        bPausa=false;
    }else{
        btnPausa.setLabel("  Pausa  ");
        hilo1.resume();
        bPausa=true;
    }
}

```

En la definición de la función miembro *run*, de la clase *Hilo*, se ejecuta un bucle sin fin. Tiene que haber algún modo de terminar la ejecución del subproceso. Ya hemos visto que un subproceso alcanza el estado Death (muerte), cuando se llega al final de la función *run*. El otro modo, es llamar desde el subproceso a la función *stop* miembro de la clase *Thread*.

La definición de la función respuesta a la pulsación sobre el botón Parar es la siguiente

```

void btnParar_actionPerformed(ActionEvent e) {
    hilo1.stop();
    hilo1=null;
}

```

Una vez que el subproceso *hilo1* está en estado Death no se puede volver a llamar desde *hilo1* a *start* para ponerlo en marcha, hay que volver a crear un nuevo subproceso. Esta es la razón por la que se asigna **null** a *hilo1* una vez parado el subproceso mediante la función *stop*. En la función respuesta a la acción de pulsar en el botón titulado Empieza escribimos

```

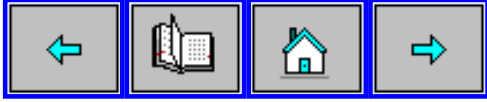
if(hilo1==null){

```



```
        hilo1=new Hilo(tTexto1, "Subproceso 1");  
        hilo1.start();  
    }
```

Implementando el interface *Runnable*



[Subprocesos \(threads\)](#)

[El interface *Runnable*](#)

[La vida del subproceso](#)

[Fechas y horas](#)

[Dibujando el reloj](#)

[Eliminando el parpadeo](#)

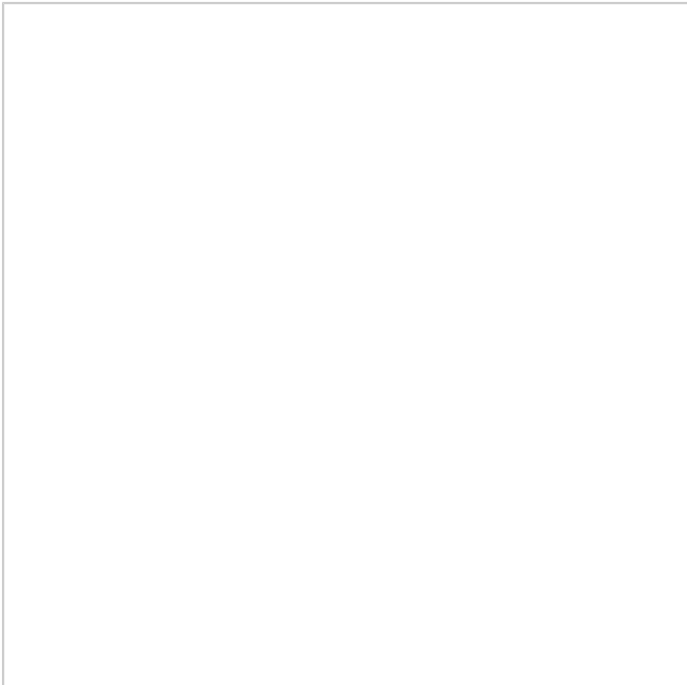
[El double-buffer](#)

[El código fuente](#)

En esta página vamos a estudiar como se crea un reloj analógico y se muestra en la ventana del applet.



reloj: [RelojApplet.java](#)



El interface *Runnable*

El [interface](#) *Runnable* solamente declara una función miembro denominada *run*, que han de definir las clases que implementen este interface.

```
public interface Runnable {
    public abstract void run();
}
```

En nuestro caso, la clase que describe el applet *RelojApplet* deriva de *Applet* por lo que no puede derivar también de *Thread* (no existe la herencia múltiple en Java), sino que tiene que implementar el interface *Runnable* y definir la función *run*.

La vida del subproceso

Además, es necesario crear un subproceso, es decir, un objeto de la clase *Thread*. Creamos este objeto en la redefinición de la función *start* miembro de la clase que describe el applet. La función miembro *start* se llama después de *init* cuando la página que contiene el applet aparece en el navegador. Se crea el objeto *hilo* de la clase *Thread*, desde dicho objeto se llama a su función miembro *start* para poner en marcha el subproceso.

```
public class RelojApplet extends Applet implements Runnable {
    Thread hilo;
    //...
    public void start(){
        if(hilo==null){
            hilo=new Thread(this);
            hilo.start();
        }
    }
}
```

Al constructor de *Thread* solamente le podemos pasar **this**, si la clase que describe el applet implementa el interface *Runnable*.

A continuación de la llamada a la función *start* miembro de *Thread*, se ejecuta la función miembro *run*, que consta de un bucle indefinido **while**. En este bucle se pueden realizar distintas tareas: mover una figura, cambiar la imagen que se muestra en una animación, actualizar los gráficos del contexto del applet, realizar un cálculo intensivo, etc. Además, no nos debemos de olvidar de establecer una pausa llamando a la función *sleep*. Esta función siempre tiene que llamarse dentro de un bloque **try..catch**

```
public void run() {
    while (true) {
        try{
            Thread.sleep(100);
        }catch (InterruptedException e) { }
```

```

        //tarear a realizar...
    }
}

```

Paramos el subproceso cuando abandonamos la página que contiene el applet, se llama entonces a la función *stop* miembro de la clase que describe el applet. En dicha función miembro, se para el subproceso *hilo*, desde este objeto se llama a la función *stop* miembro de la clase *Thread*. Por último, se asigna a *hilo* el valor **null**.

```

public void stop(){
    if(hilo!=null){
        hilo.stop();
        hilo=null;
    }
}

```

Existe una alternativa para salir del bucle sin fin **while** y llegar al final de la función *run* haciendo que el subproceso alcance el estado Death (muerto) sin llamar explícitamente a la función *stop* miembro de la clase *Thread*. Para ello, se obtiene el subproceso actual mediante la función estática *currentThread* de la clase *Thread* y se guarda en el objeto *miHilo*. Naturalmente, subproceso *hilo* y *miHilo* serán los mismos. Cuando se llama a la función *stop* (se abandona la página que contiene el applet) se fuerza a que *hilo* tome el valor nulo (**null**). La condición en **while** se deja de cumplir, llegándose al final del método *run*, que hace que el subproceso muera de muerte natural.

```

public void run() {
    Thread miHilo=Thread.currentThread();
    while(miHilo==hilo) {
        try{
            Thread.sleep(100);
        }catch (InterruptedException e) { }
        //tarear a realizar...
    }
}

public void stop(){
    hilo=null;
}

```

Fechas y horas

La versión Java 1.1 ha cambiado completamente el modo en el que se obtienen la fecha y la hora, a fin de cumplir uno de sus objetivos, que el lenguaje se adapte a través de la Internacionalización a los usos y costumbres de los distintos países. La descripción de las nuevas clases *Calendar*, *Date*, y *DateFormat* no se harán en esta sección y se remite al lector al sistema de ayuda de JBuilder para una descripción más completa.

El código que nos permite obtener la fecha (día, mes y año) es el siguiente

```

Calendar cal=Calendar.getInstance();
Date date=cal.getTime();

```

```

        DateFormat dateFormatter=DateFormat.getDateInstance(DateFormat. FULL,
Locale.getDefault());
        String fecha=dateFormatter.format(date);

```

Usamos la clase [FontMetrics](#) nos proporciona las características de la fuente de texto, para mostrar la fecha en la parte superior del applet, en su contexto gráfico *g*.

```

FontMetrics fm=g.getFontMetrics();
g.setColor(Color.yellow);
g.fillRect(0, 0, anchoApplet, 3*fm.getHeight()/2);
g.setColor(Color.black);
g.drawString(fecha, (anchoApplet-fm.stringWidth(fecha))/2, fm.getHeight());

```

El código que nos permite obtener la hora, los minutos y los segundos es el siguiente

```

SimpleDateFormat formatter = new SimpleDateFormat("s" ,Locale.getDefault());
int segundos=Integer.parseInt(formatter.format(date));
formatter.applyPattern("m");
int minutos=Integer.parseInt(formatter.format(date));
formatter.applyPattern("h");
int hora=Integer.parseInt(formatter.format(date));

```

La clase *SimpleDateFormat* se usa para trasladar la hora en milisegundos a una versión adecuada para el usuario dependiendo de los usos en el país elegido (locale). La clase *Locale* contiene un método denominado *getDefault*, que devuelve el locale por defecto del entorno en el que trabajamos.

Dibujando el reloj

Para dibujar el reloj se siguen los siguientes pasos

Se dibuja la circunferencia, los números 3, 6, 9, y 12 y las marcas situadas en las horas. Para centrar los números se hace uso de las características de la fuente de texto descritas por el objeto *fm* de la clase [FontMetrics](#).

```

g.setFont(new Font("TimesRoman", Font.BOLD, 14));
g.setColor(Color.lightGray);
g.fillOval (xCentro-radio, yCentro-radio, 2*radio, 2*radio);
g.setColor(Color.black);
g.drawString("9",xCentro-radio+3, yCentro-fm.getHeight()/2+fm.getAscent());
g.drawString("3",xCentro+radio-fm.stringWidth("3")-3,yCentro-
fm.getHeight()/2+fm.getAscent());
g.drawString("12",xCentro-fm.stringWidth("12")/2, yCentro-
radio+fm.getHeight()+3);
g.drawString("6",xCentro-fm.stringWidth("6")/2,yCentro+radio-fm.getAscent());
g.setColor(Color.red);
for(int i=0; i<12; i++){
    int xHora=xCentro+(int)(Math.cos(i*Math.PI/6)*(7*radio/9));

```

```

        int yHora=yCentro+(int)(Math.sin(i*Math.PI/6)*(7*radio/9));
        g.fillOval(xHora-2, yHora-2, 4, 4);
    }

```

Se calculan empleando algo de trigonometría, las coordenadas de las posiciones de los extremos de las agujas del reloj. Se ha tenido en cuenta el avance de la aguja de la hora a medida que transcurren los minutos desde la posición 0.

La longitud de las agujas se han tomado en la siguiente proporción: la de los segundos tiene una longitud igual al radio de la circunferencia, la de los minutos los 8/9 de este radio, y la de las horas los 6/9 del radio de la circunferencia.

```

        int xSeg=xCentro+(int)(Math.cos(segundos*Math.PI/30-Math.PI/2)*radio);
        int ySeg=yCentro+(int)(Math.sin(segundos*Math.PI/30-Math.PI/2)*radio);
        int xMin=xCentro+(int)(Math.cos(minutos*Math.PI/30-Math.PI/2)*(8*radio/9));
        int yMin=yCentro+(int)(Math.sin(minutos*Math.PI/30-Math.PI/2)*(8*radio/9));
        int xHora=xCentro+(int)(Math.cos((hora+(double)minutos/60)*Math.PI/6-
Math.PI/2)*(6*radio/9));
        int yHora=yCentro+(int)(Math.sin((hora+(double)minutos/60)*Math.PI/6-
Math.PI/2)*(6*radio/9));

```

Se dibujan de distintos colores las agujas como líneas que van desde el centro de la circunferencia hasta las posiciones de sus respectivos extremos. Se han hecho algo más gruesas las agujas de las horas y de los minutos, trazando dos líneas cercanas en vez de una.

```

        g.setColor(Color.darkGray);
        g.drawLine(xCentro, yCentro, xSeg, ySeg);
        g.setColor(Color.blue);
        g.drawLine(xCentro, yCentro-1, xMin, yMin);
        g.drawLine(xCentro-1, yCentro, xMin, yMin);
        g.drawLine(xCentro, yCentro-1, xHora, yHora);
        g.drawLine(xCentro-1, yCentro, xHora, yHora);

```

Eliminando el parpadeo

Nuestra primera intención es la de escribir un código como el siguiente

```

public void run() {
    while (true) {
        try{
            Thread.sleep(100);
        }catch (InterruptedException e) { }
        repaint();
    }
}
public void paint(Garphics g){
    dibujaReloj(g);
}

```

La función *repaint* llama a *update* de la clase base si no está definida en la clase derivada, y *update* llama a *paint* de la clase derivada para dibujar el reloj a intervalos regulares de tiempo. El efecto que se consigue es un molesto parpadeo.

La solución más simple para eliminar el parpadeo es la de redefinir la función *update* en la clase que describe el applet. La definición de *update* en la clase base *Applet* la encontramos en su clase *Component* (*Applet* es una clase derivada de *Panel*, ésta lo es de *Container*, y a su vez, ésta lo es de *Component*)

```
public void update(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0, 0, width, height);
    g.setColor(getForeground());
    paint(g);
}
```

Lo primero que hace *update* es borrar la ventana y luego, llama al método *paint*. La parte del código que borra la ventana es la que causa el parpadeo. Por lo tanto, hemos de redefinir *update* en la clase que describe el applet. Si dibujamos en la ventana sin borrarla previamente eliminamos el parpadeo.

```
public void update(Graphics g) {
    paint(g);
}
```

Esta tampoco es la solución a este problema, ya que no se borra el fondo y las imágenes aparecen superpuestas

El double-buffer

El double-buffer es la solución a muchos de los problemas asociados con la animación. En vez de dibujar directamente en la ventana del applet dibujamos en un buffer intermedio (contexto gráfico en memoria). Cuando es el momento de actualizar la animación lo que hacemos es volcar lo dibujado desde el contexto en memoria a la ventana del applet en una simple y muy rápida operación de transferencia. Luego, volvemos a dibujar en el contexto gráfico en memoria, lo volcamos a la ventana, y así sucesivamente.

```
public void update(Graphics g){
    if(gBuffer==null){
        imag=createImage(anchoApplet, altoApplet);
        gBuffer=imag.getGraphics();
    }
    gBuffer.setColor(getBackground());
    gBuffer.fillRect(0,0, anchoApplet, altoApplet);
    //dibuja el reloj
    dibujaReloj(gBuffer);
    //transfiere la imagen al contexto gráfico del applet
    g.drawImage(imag, 0, 0, null);
}
```

Como vemos en el código, es muy importante que el contexto gráfico en memoria tenga las dimensiones de la ventana del

applet. Como estamos trabajando en un contexto en memoria no hay que preocuparse por los efectos de borrarlo antes de dibujar sobre dicho contexto. De hecho, es el primer paso que hay que hacer cuando empleamos esta técnica conocida por el nombre de double-buffer.

Una vez borrado *gBuffer*, se dibuja sobre dicho contexto el reloj. Finalmente, se transfiere la imagen creada *imag* desde la memoria al contexto gráfico *g* de la ventana del applet, mediante la función *drawImage*. Fijarse que el método *paint* no se llama ahora desde *update*.

Como ya se ha advertido repetidamente en otros ejemplos. Cuando se obtiene el [contexto gráfico](#) mediante *getGraphics*, es responsabilidad del programador, liberar los recursos asociados a dicho objeto mediante la llamada *dispose*. Dado que *update* se llama muchas veces a lo largo de un proceso de animación corremos el riesgo de agotar la memoria del ordenador.

```
public void update(Graphics g){
    Image imag=createImage(anchoApplet, altoApplet);
    Graphics gBuffer=imag.getGraphics();
//...
}
```

Este problema se puede afrontar obteniendo el contexto gráfico la primera vez que se llama a *update* y guardándolo en el miembro dato *gBuffer*

```
public void update(Graphics g){
    if(gBuffer==null){
        imag=createImage(anchoApplet, altoApplet);
        gBuffer=imag.getGraphics();
    }
//...
}
```

El código fuente

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

public class RelojApplet extends Applet implements Runnable {
    Thread hilo = null;
    int anchoApplet, altoApplet;
//Doble buffer
    Image imag;
    Graphics gBuffer;

    public void init() {
        try {
            jbInit();
        }
        catch (Exception e) {
```



```

        e.printStackTrace();
    }
}

private void jbInit() throws Exception {
    this.setBackground(Color.white);
    anchoApplet=getSize().width;
    altoApplet=getSize().height;
}

public void start() {
    if(hilo == null) {
        hilo = new Thread(this);
        hilo.start();
    }
}

public void stop() {
    hilo.stop();
    hilo = null;
}

public void run() {
    while (true) {
        try{
            Thread.sleep(100);
        }catch (InterruptedException e) { }
        repaint();
    }
}

public void dibujaReloj (Graphics g) {
//fecha
    Calendar cal=Calendar.getInstance();
    Date date=cal.getTime();
    DateFormat dateFormatter=DateFormat.getDateInstance(DateFormat.FULL,
Locale.getDefault());
    String fecha=dateFormatter.format(date);
    FontMetrics fm=g.getFontMetrics();
    g.setColor(Color.yellow);
    g.fillRect(0, 0, anchoApplet, 3*fm.getHeight()/2);
    g.setColor(Color.black);
    g.drawString(fecha, (anchoApplet-fm.stringWidth(fecha))/2, fm.getHeight());

//hora, minutos y segundos
    SimpleDateFormat formatter = new SimpleDateFormat("s" ,Locale.getDefault());
    int segundos=Integer.parseInt(formatter.format(date));
    formatter.applyPattern("m");
    int minutos=Integer.parseInt(formatter.format(date));
    formatter.applyPattern("h");
    int hora=Integer.parseInt(formatter.format(date));

```

```

//el centro del reloj y el radio
int xCentro=getSize().width/2;
int yCentro=2*fm.getHeight()+(altoApplet-2*fm.getHeight())/2;
int radio=(xCentro>(altoApplet-2*fm.getHeight())/2)?(altoApplet-
2*fm.getHeight())/2:xCentro;

//Dibujar la circunferencia, los números y las marcas
g.setFont(new Font("TimesRoman", Font.BOLD, 14));
g.setColor(Color.lightGray);
g.fillOval (xCentro-radio, yCentro-radio, 2*radio, 2*radio);
g.setColor(Color.black);
g.drawString("9",xCentro-radio+3, yCentro-fm.getHeight()/2+fm.getAscent());
g.drawString("3",xCentro+radio-fm.stringWidth("3")-3,yCentro-
fm.getHeight()/2+fm.getAscent());
g.drawString("12",xCentro-fm.stringWidth("12")/2, yCentro-
radio+fm.getHeight()+3);
g.drawString("6",xCentro-fm.stringWidth("6")/2,yCentro+radio-fm.getAscent());
g.setColor(Color.red);
for(int i=0; i<12; i++){
    int xHora=xCentro+(int)(Math.cos(i*Math.PI/6)*(7*radio/9));
    int yHora=yCentro+(int)(Math.sin(i*Math.PI/6)*(7*radio/9));
    g.fillOval(xHora-2, yHora-2, 4, 4);
}

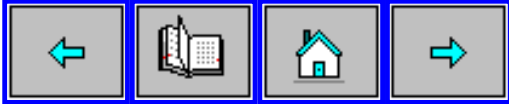
//posición de las agujas del reloj
int xSeg=xCentro+(int)(Math.cos(segundos*Math.PI/30-Math.PI/2)*radio);
int ySeg=yCentro+(int)(Math.sin(segundos*Math.PI/30-Math.PI/2)*radio);
int xMin=xCentro+(int)(Math.cos(minutos*Math.PI/30-Math.PI/2)*(8*radio/9));
int yMin=yCentro+(int)(Math.sin(minutos*Math.PI/30-Math.PI/2)*(8*radio/9));
int xHora=xCentro+(int)(Math.cos((hora+(double)minutos/60)*Math.PI/6-
Math.PI/2)*(6*radio/9));
int yHora=yCentro+(int)(Math.sin((hora+(double)minutos/60)*Math.PI/6-
Math.PI/2)*(6*radio/9));
//las agujas del reloj
g.setColor(Color.darkGray);
g.drawLine(xCentro, yCentro, xSeg, ySeg);
g.setColor(Color.blue);
g.drawLine(xCentro, yCentro-1, xMin, yMin);
g.drawLine(xCentro-1, yCentro, xMin, yMin);
g.drawLine(xCentro, yCentro-1, xHora, yHora);
g.drawLine(xCentro-1, yCentro, xHora, yHora);
}

public void update(Graphics g){
    if(gBuffer==null){
        imag=createImage(anchoApplet, altoApplet);
        gBuffer=imag.getGraphics();
    }
    gBuffer.setColor(getBackground());
    gBuffer.fillRect(0,0, anchoApplet, altoApplet);
//dibuja el reloj

```

```
        dibujaReloj(gBuffer);  
//transfiere la imagen al contexto gráfico del canvas  
        g.drawImage(imag, 0, 0, null);  
    }  
}
```

Sincronización



Subprocesos (threads)

[La palabra reservada **synchronized**](#)

[El modelo Productor/Consumidor](#)

[Cargando imágenes](#)

La palabra reservada **synchronized**

La palabra reservada **synchronized** se usa para indicar que ciertas partes del código, (habitualmente, una función miembro) están sincronizadas, es decir, que solamente un subproceso puede acceder a dicho método a la vez.

Cada método sincronizado posee una especie de llave que puede cerrar o abrir la puerta de acceso. Cuando un subproceso intenta acceder al método sincronizado mirará a ver si la llave está echada, en cuyo caso no podrá accederlo. Si método no tiene puesta la llave entonces el subproceso puede acceder a dicho código sincronizado.

Las siguientes porciones de código son ejemplos de uso del modificador **synchronized**

```
synchronized public void funcion1(){
    //...
}
```

```
public void funcion2(){
    Rectangle rect;
    synchronized(rect){
        rect.width+=2;
    }
    rect.height-=3;
}
```

Un ejemplo que [veremos más adelante](#) es la sincronización de una porción de código usando el objeto **this**, en el interior de una función miembro denominada *mover*

```
public void mover(){
    synchronized (this) {
        indice++;
        if (indice>= numeros.length) {
```

```


        indice=0;
    }
}
//...
}

```

En la primera porción de código, hemos asegurado un método de modo que un sólo subproceso a la vez puede acceder a la *funcion1*. En la segunda y tercera porción de código, tenemos un bloque de código asegurado. La anchura *width* del rectángulo *rect* no puede ser modificada por varios subprocesos a la vez. La altura *height* del rectángulo no está dentro del bloque sincronizado y puede ser modificada por varios subprocesos a la vez. El objeto *rect* se usa en este caso como llave de dicho bloque de código, en el tercer ejemplo este papel lo representa **this**.

Se debe evitar la sincronización de bloques de código y sustituirlas siempre que sea posible por la sincronización de métodos, lo que está más de acuerdo con el espíritu de la programación orientada a objetos. Se debe tener en cuenta que la sincronización disminuye el rendimiento de una aplicación, por tanto, debe emplearse solamente donde sea estrictamente necesario.

El modelo Productor/Consumidor

 thread2: [Productor.java](#), [Consumidor.java](#), [Buffer.java](#), [ThreadApp2.java](#)

En la página previa, los subprocesos eran independientes, cada subproceso contiene los recursos que le son necesarios para su ejecución, corren a su propio paso sin interesarse por el estado o las tareas de los otros subprocesos que se ejecutan a la vez. Sin embargo, hay muchas situaciones interesantes en las que los subprocesos comparten datos y han de considerar el estado y las actividades de los otros subprocesos. Una de estas situaciones se denomina modelo Productor/Consumidor: el productor genera un flujo de datos que son recogidos por el consumidor. Cuando dos subprocesos comparten un recurso común han de estar sincronizados de algún modo.

Un problema se origina cuando el productor va más rápido que el consumidor y genera un segundo dato antes de que el consumidor tenga la oportunidad de recoger el primero. El consumidor se saltará un dato. Del mismo modo, si el consumidor es más rápido que el productor, puede que no tenga datos que recoger, o que recoja varias veces el mismo dato.

Cuando estamos diseñando subprocesos que compiten por recursos limitados hemos de tener cuidado en no caer en dos situaciones extremas, denominadas *starvation* (morir de hambre, el subproceso no progresa por falta de recursos) y *deadlock* (por ejemplo, cuando dos personas están en conflicto y uno está esperando a que el otro tome la iniciativa para resolverlo y viceversa)

El productor

La clase que describe el productor denominada *Productor* [deriva de la clase *Thread*](#) y redefine la función *run*. Tiene como miembro dato un objeto *buffer* de la clase *Buffer* que describiremos más adelante.

La función miembro *run* ejecuta un bucle **for**, cuando se completa el bucle se alcanza el final de *run* y el subproceso entra en el estado Death (muerto), y detiene su ejecución.

En el bucle **for**, se genera una letra al azar y se pone en el objeto *buffer*, llamando a la función *poner* de la clase *Buffer*. A continuación, se imprime y se hace un pausa por un número determinado de milisegundos llamando a la función *sleep* y pasándole el tiempo de pausa, durante este tiempo el subproceso esta en el estado Not Runnable.

El bucle **for** no se suele utilizar sino [un bucle while](#) de la forma que se explicó en la página anterior

```
public class Productor extends Thread {
    private Buffer buffer;
    private final String letras="abcdefghijklmnopqrstuvwxyz";
    public Productor(Buffer buffer) {
        this.buffer=buffer;
    }
    public void run() {
        for(int i=0; i<10; i++){
            char c=letras.charAt((int)(Math.random()*letras.length()));
            buffer.poner(c);
            System.out.println(i+" Productor: " +c);
            try {
                sleep(400);
            } catch (InterruptedException e) { }
        }
    }
}
```

El consumidor

La clase que describe al consumidor denominado *Consumidor*, [deriva también de la clase *Thread*](#) y redefine el método *run*. La definición de *run* es similar a la de la clase *Productor*, salvo que en vez de poner un carácter en el *buffer*, recoge el carácter guardado en el *buffer* intermedio llamando a la función *recoger*.

El tiempo de pausa (argumento de la función *sleep*) puede ser distinto en la clase *Productor* que en la clase *Consumidor*. Por ejemplo, para hacer que el productor sea más rápido que el consumidor, se pone un tiempo menor en la primera que en la segunda.

```

public class Consumidor extends Thread {
    private Buffer buffer;
    public Consumidor(Buffer buffer) {
        this.buffer=buffer;
    }
    public void run(){
        char valor;
        for(int i=0; i<10; i++){
            valor=buffer.recoger();
            System.out.println(i+ " Consumidor: "+valor);
            try{
                sleep(100);
            }catch (InterruptedException e) { }
        }
    }
}

```

El buffer (primera aproximación)

El objeto compartido entre el productor y el consumidor está descrito por la clase denominada *Buffer*. Vamos a llegar a la definición de dicha clase en dos pasos sucesivos.

La clase *Buffer* tiene dos miembros dato: el primero *contenido* guarda un carácter (es el buffer), el segundo, *disponible* indica si el buffer está lleno o está vacío, según que ésta variable valga **true** o **false**.

La definición de las funciones *poner* y *recoger* es muy simple:

- La función *poner* guarda el carácter que se le pasa en su parámetro *c* en el miembro dato *contenido*, y pone el miembro *disponible* en **true** (el buffer está lleno).
- La función *recoger* devuelve el carácter guardado en el miembro *contenido*, siempre que este disponible (el buffer lleno, o *disponible* valga **true**). En el caso de que no esté disponible (*disponible* valga **false**) devuelve un carácter no alfabético: un espacio, un tabulador, lo que desee el usuario.

```

public class Buffer {
    private char contenido;
    private boolean disponible=false;
    public Buffer() {
    }
    public char recoger(){
        if(disponible){
            disponible=false;
            return contenido;
        }
    }
}

```

```

    }
    return ('\t');
}
public void poner(char c){
    contenido=c;
    disponible=true;
}
}

```

La aplicación

Definimos una clase que describe una aplicación. En la función *main*, creamos tres objetos un objeto *b* de la clase *Buffer*, un objeto *p* de la clase *Productor* y otro objeto *c* de la clase *Consumidor*. Al constructor de las clases *Productor* y *Consumidor* le pasamos el objeto *b* compartido de la clase *Buffer*.

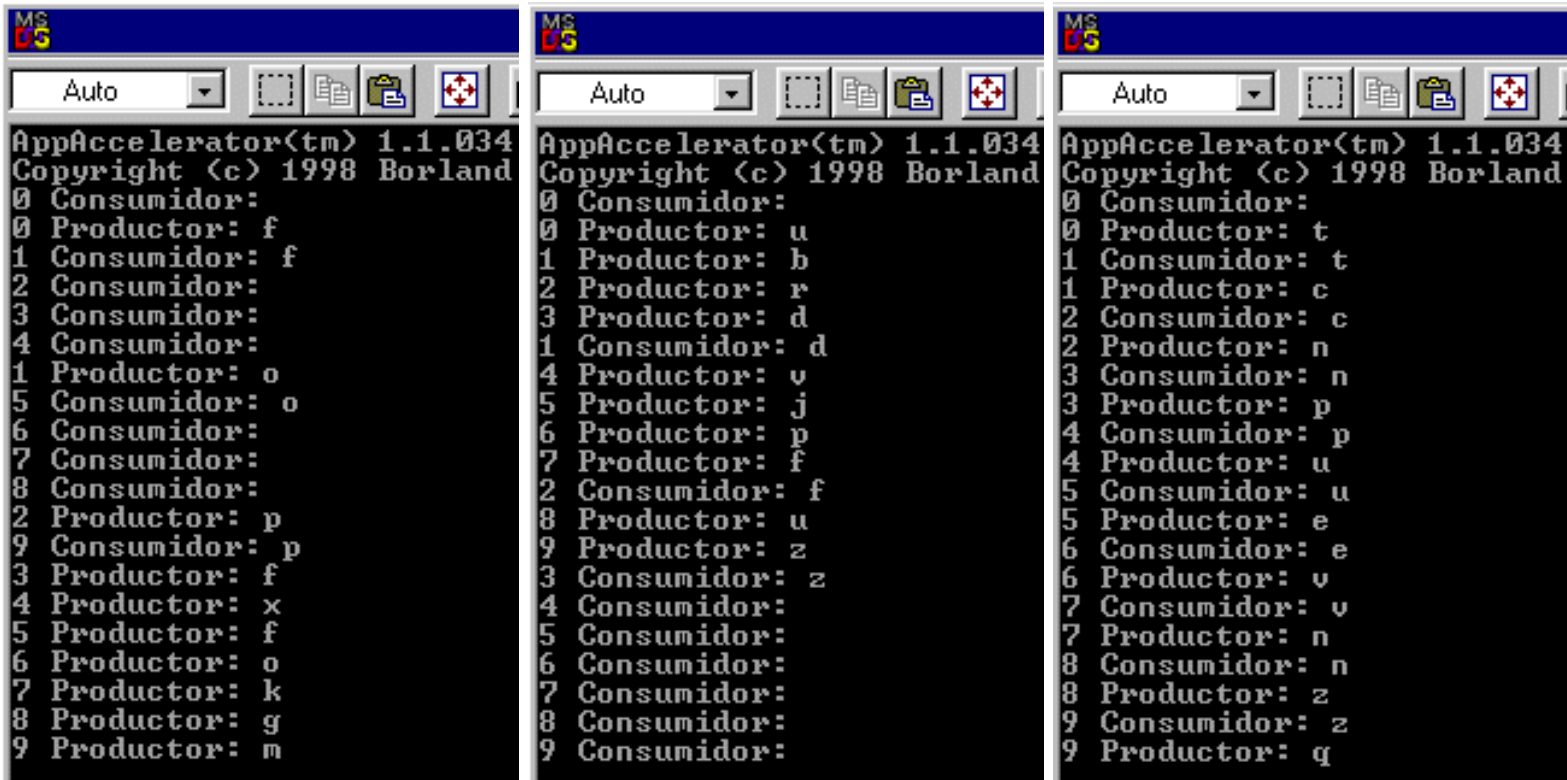
Ponemos en marcha los subprocesos descritos por las clases *Productor* y *Consumidor*, mediante la llamada a la función *start* de modo que el estado de los subprocesos pasa de New Thread a Runnable. Los subprocesos una vez puestos en marcha mueren de muerte natural, pasando al estado Death después de ejecutar un bucle **for** de 10 iteraciones en la función *run*.

```

public class ThreadApp2 {
    public static void main(String[] args) {
        Buffer b=new Buffer();
        Productor p=new Productor(b);
        Consumidor c=new Consumidor(b);
        p.start();
        c.start();
    }
}

```

Cuando corremos la aplicación observamos la siguiente salida



La imagen de la izquierda corresponde a la situación en la que el consumidor va más rápido (100 como argumento de *sleep*) que el productor (400 como argumento de *sleep*). En la primera iteración (líneas marcadas con 0), el consumidor va al buffer y no encuentra ninguna letra (inicialmente esta vacío), el productor pone en el buffer la letra **f**. En la segunda iteración (líneas marcadas con 1) el consumidor consume la letra **f**, y el buffer se queda vacío, el consumidor vuelve a acceder de nuevo al buffer y no encuentra nada, etc.

La imagen del centro corresponde a la situación en la que el productor va más rápido (100 como argumento de *sleep*) que el consumidor (400 como argumento de *sleep*). En la primera iteración (líneas marcadas con 0) el consumidor accede al buffer y lo encuentra vacío, el productor pone en el buffer el letra **u**. En sucesivas iteraciones el productor pone en el buffer las letras **b**, a continuación la sustituye por **r** y luego, por **d**. En la segunda iteración (línea marcada por 1) del consumidor consume esta última letra **d**, etc.

En la tercera imagen, el productor y el consumidor tienen la misma rapidez (100 como argumento de sus funciones *sleep*). El productor y el consumidor están más coordinados, ya que la letra que pone el productor en el buffer es consumida por el consumidor. Esta situación es ideal y limitada a unas pocas iteraciones, ya que en general, los dos subprocesos ejecutan tareas distintas durante miles de iteraciones, por lo que no tienen la misma velocidad aún cuando el argumento de la función *sleep* sea el mismo.

El buffer (segunda aproximación)

Para resolver este problema los subprocesos han de estar sincronizados de dos modos. En primer lugar, se ha de poner el modificador **synchronized** delante de los métodos *poner* y *recoger*. Los dos subprocesos no pueden de este modo acceder simultáneamente al objeto *buffer* compartido.

Como ya hemos mencionado en el primer apartado, una sección crítica es un bloque de código o un método que es identificado por la palabra reservada **synchronized**. De este modo, el *consumidor* no puede acceder al objeto *buffer* cuando el *productor* lo está cambiando (llama a *poner*). El *productor* no podrá cambiarlo cuando el consumidor está obteniendo (llama a *recoger*) el valor que guarda dicho objeto.

En segundo lugar, se ha de mantener una coordinación entre el productor y el consumidor, de modo que cuando el productor ponga una letra en el buffer avise al consumidor de que el buffer está disponible para recoger dicha letra y viceversa, es decir, cuando el consumidor recoja la letra avise al productor de que el buffer está vacío. A su vez, el consumidor esperará hasta que el buffer esté lleno con una letra y el productor esperará hasta que el buffer esté nuevamente vacío para poner otra letra.

La clase *Thread* nos proporcionan los métodos *wait*, *notify* y *notifyAll*, para hacer que los subprocesos esperen hasta que se cumpla una determinada condición, y cuando se cumpla se notifica a otros subprocesos que la condición ha cambiado.

El código de las funciones miembro *recoger* y *poner* será ahora el siguiente:

- **Función *recoger***

Espera (*wait*) mientras (**while**) *disponible* sea **false** (buffer vacío). En caso contrario (*disponible* sea **true** o el buffer esté lleno), devuelve la letra contenida en el buffer (**return contenido**), poner de nuevo *disponible* en **false** (se ha vaciado el buffer al recoger la letra), y avisa (*notify* o *notifyAll*) a los otros subprocesos de este hecho.

- **Función *poner***

Espera (*wait*) mientras (**while**) *disponible* sea **true** (buffer lleno). En caso contrario (*disponible* sea **false** o el buffer esté vacío), guarda en el buffer (miembro dato *contenido*) la letra que se le pasa en el parámetro de la función *poner*. Cambia el valor de *disponible* a **true**, y avisa (*notify* o *notifyAll*) a los otros subprocesos de este hecho.

El método *wait* de la clase *Thread* hace que el subproceso espere en un estado Not Runnable hasta que sea avisado (*notify*) de que continúe. El método *notify* informa al subproceso en espera que continúe su ejecución. *notifyAll* es similar a *notify* excepto que se aplica a todos los subprocesos en estado de espera. Estos métodos solamente se pueden llamar desde funciones sincronizadas (con modificador **synchronized**). Las llamadas a la función *wait* como *sleep* deben de estar dentro de un bloque **try...catch**.

El código de la clase *Buffer* será ahora el siguiente

```
public class Buffer {
    private char contenido;
    private boolean disponible=false;
    public Buffer() {
    }

    public synchronized char recoger(){
        while(!disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        disponible=false;
        notify();
        return contenido;
    }

    public synchronized void poner(char valor){
        while(disponible){
            try{
                wait();
            }catch(InterruptedException ex){}
        }
        contenido=valor;
        disponible=true;
        notify();
    }
}
```

La salida del programa será ahora la adecuada, tal como se muestra en la figura inferior. El productor y el consumidor están ya coordinados.



Cargando imágenes

El retraso en la carga de las imágenes y otros recursos desde un servidor remoto es una situación que experimentamos continuamente. El lenguaje Java dispone de una librería de clases que trata con eficacia este problema.

El modelo productor-consumidor tiene en cuenta la progresiva carga de las imágenes y supone que el productor genera los pixels de la imagen y el consumidor la muestra. Java utiliza también el concepto de filtro que permite cambiar el aspecto de la imagen a medida que pasa desde el productor al consumidor.

El modelo productor/consumidor tiene muchas ventajas entre las que cabe destacar la modularidad y la interacción asíncrona entre productor y consumidor. Esto último significa, que una vez que se ha conectado el productor con el consumidor, el productor notifica al consumidor solamente cuando hay información adicional disponible, mientras tanto, el applet o la aplicación pueden seguir haciendo otro trabajo.

Los productores representan la fuente de los datos situados al otro lado de la red. Los consumidores representan los applets o aplicaciones situados a este lado de la red. Su trabajo conjunto permite enviar los recursos desde un lado hacia el otro.

El interface *ImageProducer* tiene los siguientes métodos:

- `public void addConsumer(ImageConsumer ic);`
- `public boolean isConsumer(ImageConsumer ic);`
- `public void removeConsumer(ImageConsumer ic);`

- `public void startProduction(ImageConsumer ic);`
- `public void requestTopDownLeftRightResend(ImageConsumer ic);`

La clase que describe un objeto productor ha de implementar el interface *ImageProducer*. Como vemos los métodos requieren que se les pase un objeto cuya clase implemente el interface *ImageConsumer*. Ya que un productor no tiene sentido sin su consumidor asociado. Un productor puede tener múltiples consumidores que se añaden mediante *addConsumer* y se eliminan mediante *removeConsumer*.

La clase que describe el objeto consumidor (normalmente es uno) ha de implementar el interface *ImageConsumer*, que es más complejo.

- `public void setDimensions(int width, int height);`
- `public void setProperties(Hashtable props);`
- `public void setColorModel(ColorModel model);`
- `public void setHints(int hintflags);`
- `public void setPixels(int x, int y, int w, int h, ColorModel model, byte pixels[], int off, int scansize);`
- `public void setPixels(int x, int y, int w, int h, ColorModel model, int pixels[], int off, int scansize);`
- `public void imageComplete(int status);`

Veamos el proceso de carga de una imagen proveniente de la Red.

1. El productor *ImageProducer* comienza a leer la imagen. Lo primero que lee es la anchura y la altura de la imagen. Notifica al consumidor *ImageConsumer* de la dimensión de la imagen llamando al método *setDimension*.
2. A continuación, el productor lee el mapa del color de la imagen. El productor determina el tipo de modelo de color que usa la imagen, y se lo comunica al consumidor llamando al método *setColorModel*.
3. El productor llama al método *setHints* del consumidor para comunicarle como pretende enviarle los pixels de la imagen. Los posibles valores son del parámetro *hintflags* de esta función son

| Nombre | Significado |
|---------------------|--|
| RANDOMPIXELORDER=1 | No hace mención de la forma en la que se envían los pixels |
| TOPDOWNLEFTRIGHT=2 | Para dibujar la imagen de arriba hacia abajo, desde la izquierda a la derecha |
| COMPLETESCANLINES=4 | Se envían filas completas de pixels |
| SINGLEPASS=8 | El envío de los pixels de la imagen se hace en una sola llamada a <i>setPixels</i> |
| SINGLEFRAME=16 | Una sola imagen |

4. El productor empieza a "producir" los pixels, llamado a la función *setPixels* del consumidor para enviarle la imagen. Esto se puede hacer a lo largo de muchas llamadas especialmente si se envía línea por línea de la imagen *COMPLETESCANLINES*. O puede hacerse en una única llamada si se hace en un solo paso *SINGLEPASS*.

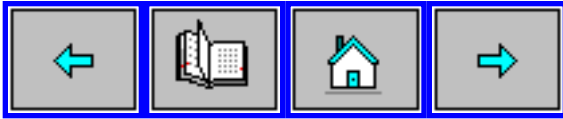
5. Finalmente, el productor llama al método *imageComplete* del consumidor para comunicarle que la imagen ha sido ya enviada. Si hay algún fallo durante este proceso, por ejemplo, se corta la comunicación con el servidor remoto, el parámetro *status* toma el valor `IMAGEERROR`. Cuando todos los pixels de la imagen se ha enviado con éxito el parámetro *status* toma el valor `STATICIMAGEDONE`

Nota: para un estudio de los filtros consultar uno de los libros

David M. Geary. *Graphic Java 2. Volume I. AWT*. Java Series. Sun Microsystems Press (1998)

David M. Geary. *Graphic JAVA 1.1. Mastering the AWT, second edition*. Sun Microsystems (1997).

Ejercicio

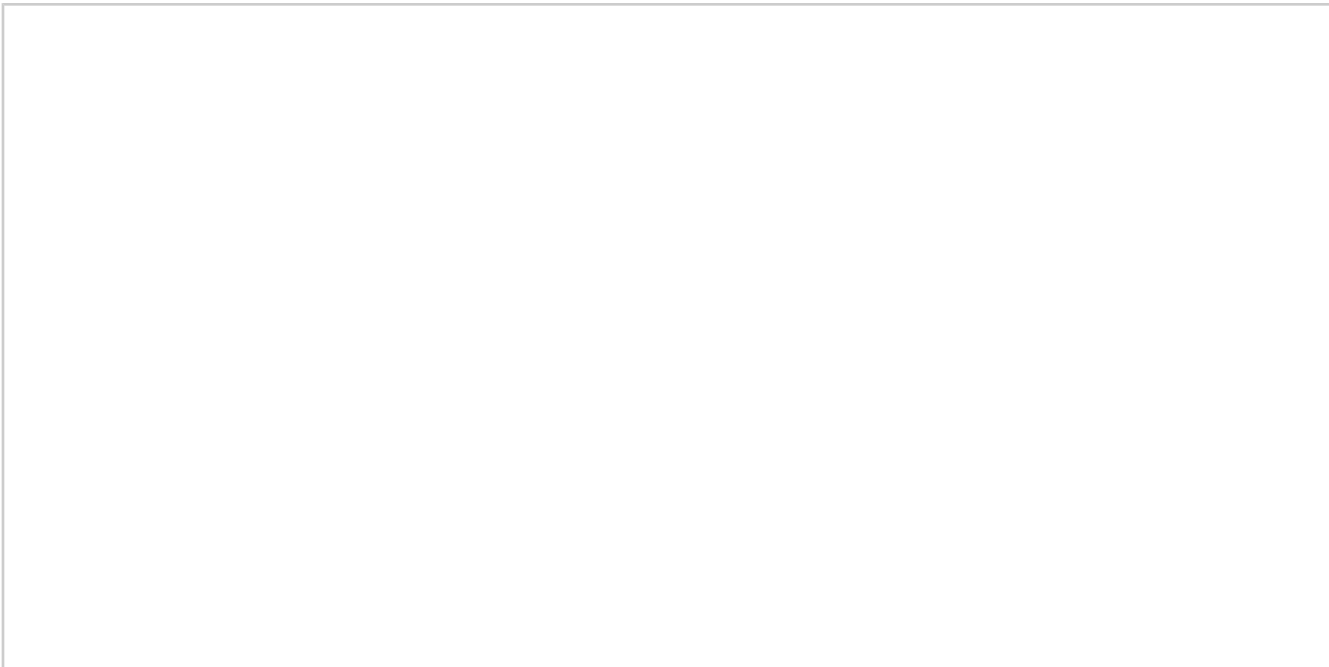


Subprocesos (threads)

[Diseño](#)

[El código fuente](#)

Vemos en el applet, varios círculos aumentando y disminuyendo de radio, de forma independiente unos de los otros y a distinto ritmo. Este ejemplo nos da la ilusión de un conjunto de subprocesos que se ejecutan a la vez.



Diseño

El proyecto consta de dos clases

La clase denominada *Circulo*

La clase [derivará de la clase base *Thread*](#)

y tendrá los siguientes miembros dato

- el color
- el radio medio
- el radio actual
- el tiempo de pausa (argumento de *sleep*)
- una variable de tipo **boolean** que indique si el radio crece o decrece
- una referencia al applet

y las funciones miembro siguientes:

- la función miembro *run*
- la función que dibuja un círculo del color especificado

Como criterio se establece que el círculo crece hasta que su radio actual sea el doble que su radio medio, y decrece hasta que su radio sea igual a 1.

La clase que describe el applet

Se reservará espacio para un array que guarde un máximo de 10 círculos

En la función miembro *init* se crearán los objetos de la clase *Circulo* y se guardarán en el array.

En la función miembro *start* se pondrán en marcha los subprocesos descritos por la clase *Circulo*.

En la función miembro *stop*, se pararán todos los subprocesos descritos por la clase *Circulo*.

Se redefinirá *update*, dibujándose todos los círculos empleando la técnica del [double-buffer](#).

Criterios

El radio medio de los círculos será un número aleatorio comprendido entre 5 y 25

Se elegirá al azar un color de un array colores y se pintará el círculo de dicho color.

El tiempo de pausa (argumento de *sleep*) será un número al azar entre 50 y 550

La posición del círculo estará dada por dos números al azar

- la abcisa será un número al azar determinado por la anchura del applet menos un margen de 20 unidades
- la ordenada será un número al azar determinado por la altura del applet menos un margen de 20 unidades

Estos datos se pasan al constructor de la clase *Circulo* cuando se crean los objetos de dicha clase.

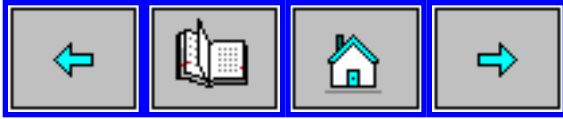
Un número al azar comprendido entre 0 y 1 se puede generar con la función [*random*](#) de la clase *Math*, o bien, se puede emplear la clase [*Random*](#).

El código fuente



procesos: [Circulo.java](#), [ProcesosApplet.java](#)

Moviendo una figura por el área de trabajo de una ventana



[Subprocesos \(threads\)](#)

[La primera aproximación al problema de la animación](#)

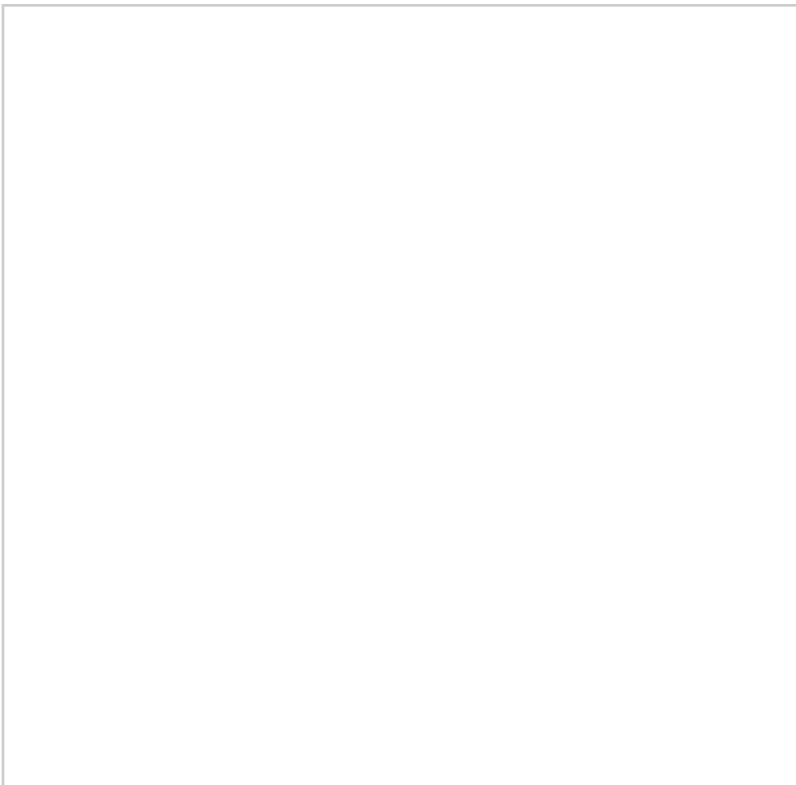
[Eliminando el parpadeo](#)

[Insertando un retardo en el bucle sin fin](#)

La primera aproximación al problema de la animación



anima1: [AnimaApplet1.java](#)



La clase que describe el applet *AnimaApplet1* deriva de *Applet* y ha de [implementar el interface *Runnable*](#) y definir la función *run*, además de otras funciones cuyas tareas se explicarán a lo largo de esta página.

Además, es necesario crear un subproceso, es decir, un objeto de la clase *Thread*. Creamos este objeto en la redefinición de la [función *start*](#) miembro de la clase que describe el applet. La función miembro *start* se llama después de *init* cuando la página que contiene el applet aparece en el navegador. Se crea el objeto *anima* de la clase *Thread*, desde dicho objeto se llama a su función miembro *start* para poner en marcha el subproceso.

```
public class AnimaApplet1 extends Applet implements Runnable {
    Thread anima;
    //...
    public void start(){
        if(anima ==null){
            anima=new Thread(this);
            anima.start();
        }
    }
}
```

A continuación, se ejecuta la función miembro *run*, que consta de un bucle indefinido **while**, que llama a la función *mover*.

```
public void run() {
    while (true) {
        mover();
    }
}
```

La función *mover* actualiza la posición de la figura que se mueve y comprueba que no supera los límites de su confinamiento. Por último, llama a *paint* para dibujar la figura en su nueva posición. Como vemos en el código, cuando la pelota alcanza los contornos del applet, rebota.

```
void mover() {
    x += dx;
    y += dy;
    if (x >= (anchoApplet-radio) || x <= radio) dx*= -1;
    if (y >= (altoApplet-radio) || y <= radio) dy*= -1;
    repaint();      //llama a paint
}
```

Dibujamos la pelota de color rojo en su nueva posición

```
public void paint (Graphics g) {
    g.setColor(Color.red);
    g.fillOval(x-radio, y-radio, 2*radio, 2*radio);
}
```

El movimiento de la pelota [lo paramos](#) cuando abandonamos la página que contiene el applet, se llama entonces a la función *stop* miembro de la clase que describe el applet. En dicha función miembro, se para el subproceso *anima*, desde este objeto se llama a la función *stop* miembro de la clase *Thread*. Por último, se asigna a *anima* el valor **null**.

```
public void stop(){
    if(anima!=null){
        anima.stop();
        anima=null;
    }
}
```

El código completo de este ejemplo, es el siguiente

```
package animal;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class AnimaApplet1 extends Applet implements Runnable {
    Thread anima;
    int radio=10;           //radio de la pelota
    int x, y;               //posición del centro de la pelota
    int dx = 1;             //desplazamientos
    int dy = 1;
    int anchoApplet;
    int altoApplet;

    public void init () {
        anchoApplet=getSize().width;           //dimensiones del applet
        altoApplet=getSize().height;
        x=anchoApplet/4;                       //posición inicial de partida
        y=altoApplet/2;
    }
}
```

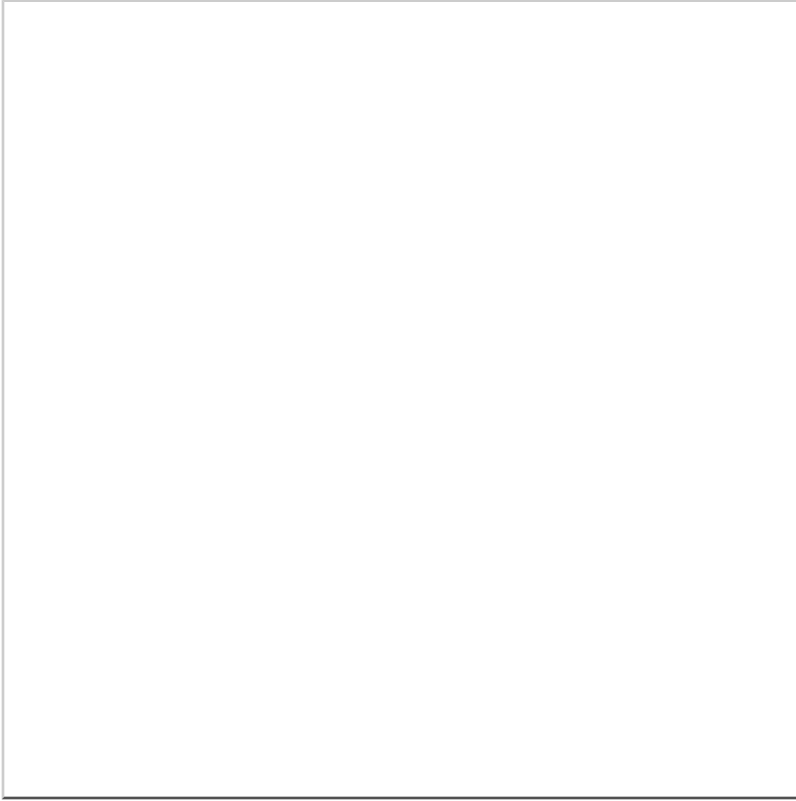
```

public void start(){
    if(anima ==null){
        anima=new Thread(this);
        anima.start();
    }
}
public void stop(){
    if(anima!=null){
        anima.stop();
        anima=null;
    }
}
public void run() {
    while (true) {
        mover();
    }
}
void mover() {
    x += dx;
    y += dy;
    if (x >= (anchoApplet-radio) || x <= radio) dx*= -1;
    if (y >= (altoApplet-radio) || y <= radio) dy*= -1;
    repaint();      //llama a update
}
public void paint (Graphics g) {
    g.setColor(Color.red);
    g.fillOval(x-radio, y-radio, 2*radio, 2*radio);
}
}

```

Nuestra primera aproximación a la animación no es nada brillante. Existen dos problemas: la pelota se mueve muy rápidamente y en segundo lugar, se produce un molesto parpadeo causado, por el borrado de la ventana del applet y vuelta a pintarlo de nuevo con la pelota en una nueva posición. Como hemos explicado, *repaint* no llama directamente a *paint* sino a *update*, que borra la ventana y a continuación llama a *paint*.

Eliminando el parpadeo

**anima2:** [AnimaApplet2.java](#)

La solución más simple para eliminar el parpadeo en la animación es la de [redefinir la función *update*](#) en la clase que describe el applet. Lo primero que hace *update* es borrar la ventana y luego, llama al método *paint*. La parte del código que borra la ventana es la que causa el parpadeo. Por lo tanto, hemos de redefinir *update* en la clase que describe el applet. Si dibujamos en la ventana sin borrarla previamente eliminamos el parpadeo.

```
public void update(Graphics g) {  
    paint(g);  
}
```

La solución a este problema es parcial, ya que no se borra el fondo y en la ventana del applet la pelota aparece en todas las posiciones por las que ha pasado a lo largo de su movimiento. Esta situación es conveniente, cuando queremos que se muestren las sucesivas posiciones por las que ha pasado el móvil, para darnos una idea acerca de su trayectoria.

El [double-buffer](#) es la solución a muchos de los problemas asociados con la animación. En vez de dibujar directamente en la ventana del applet dibujamos en un buffer intermedio (contexto gráfico en memoria). Cuando es el momento de actualizar la animación lo que hacemos es volcar lo dibujado desde el contexto en memoria a la ventana del applet en una simple y muy rápida operación de transferencia. Luego, volvemos a dibujar en el contexto gráfico en memoria, lo volcamos a la ventana, y así sucesivamente.

```

public void update(Graphics g){
    if(gBuffer==null){
        imag=createImage(anchoApplet, altoApplet);
        gBuffer=imag.getGraphics();
    }
    gBuffer.setColor(getBackground());
    gBuffer.fillRect(0,0, anchoApplet, altoApplet);
//dibuja la pelota
    gBuffer.setColor(Color.red);
    gBuffer.fillOval(x-radio, y-radio, 2*radio, 2*radio);
//transfiere la imagen al contexto gráfico del applet
    g.drawImage(imag, 0, 0, null);
}

```

Como vemos en el código, es muy importante que el contexto gráfico en memoria tenga las dimensiones de la ventana del applet. Como estamos trabajando en un contexto en memoria no hay que preocuparse por los efectos de borrarlo antes de dibujar sobre dicho contexto. De hecho, es el primer paso que hay que hacer cuando empleamos esta técnica conocida por el nombre de double-buffer.

Una vez borrado *gBuffer*, se dibuja sobre dicho contexto la pelota en la nueva posición. Finalmente, se transfiere la imagen creada *imag* desde la memoria al contexto gráfico *g* de la ventana del applet, mediante la función *drawImage*. Fijarse que el método *paint* no se llama ahora desde *update*.

Insertando un retardo en el bucle sin fin

El segundo problema que observamos al ejecutar el applet *AnimaApplet1*, es que las posiciones de la pelota parecen aleatorias en la ventana del applet. Precisamos introducir un retardo en el bucle sin fin **while**.

La clase *Thread* dispone de una función miembro *sleep* que hace que el subproceso detenga su ejecución durante un número de milisegundos indicado en su argumento. Dicha función, solamente se puede llamar dentro de un bloque **try...catch**, ya que puede producirse una [excepción](#) del tipo *InterruptedException*.

```

public void run() {
    while (true) {
//...
        try{
            Thread.sleep(200);
        }catch(InterruptedException ex){

```

```

        break;
    }
}
}

```

Se puede mejorar el código escribiendo la función *run* de la forma en la que se indica en el siguiente listado.

```

package anima2;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class AnimaApplet2 extends Applet implements Runnable {
    Thread anima;
    int radio=10;           //radio de la pelota
    int x, y;               //posición del centro de la pelota
    int dx = 1;             //desplazamientos
    int dy = 1;
    int anchoApplet;
    int altoApplet;
    int retardo=80;
    //Doble buffer
    Image imag;
    Graphics gBuffer;

    public void init () {
        setBackground(Color.white);
        anchoApplet=getSize().width;           //dimensiones del applet
        altoApplet=getSize().height;
        x=anchoApplet/4;                       //posición inicial de partida
        y=altoApplet/2;
    }

    public void start(){
        if(anima ==null){
            anima=new Thread(this);
            anima.start();
        }
    }

    public void stop(){
        if(anima!=null){

```



```

        anima.stop();
        anima=null;
    }
}
public void run() {
    long t=System.currentTimeMillis();
    while (true) {
        mover();
        try{
            t+=retardo;
            Thread.sleep(Math.max(0, t-System.currentTimeMillis()));
        }catch(InterruptedException ex){
            break;
        }
    }
}
void mover(){
    x += dx;
    y += dy;
    if (x >= (anchoApplet-radio) || x <= radio) dx*= -1;
    if (y >= (altoApplet-radio) || y <= radio) dy*= -1;
    repaint();          //llama a update
}
public void update(Graphics g){
    if(gBuffer==null){
        imag=createImage(anchoApplet, altoApplet);
        gBuffer=imag.getGraphics();
    }
    gBuffer.setColor(getBackground());
    gBuffer.fillRect(0,0, anchoApplet, altoApplet);
//dibuja la pelota
    gBuffer.setColor(Color.red);
    gBuffer.fillOval(x-radio, y-radio, 2*radio, 2*radio);
//transfiere la imagen al contexto gráfico del applet
    g.drawImage(imag, 0, 0, null);
}

    public void paint (Graphics g) {
//se llama la primera vez que aparece el applet
    }
}

```

El control de la animación



[Subprocesos \(threads\)](#)

[El diseño del applet](#)

[Los botones que controlan la animación](#)

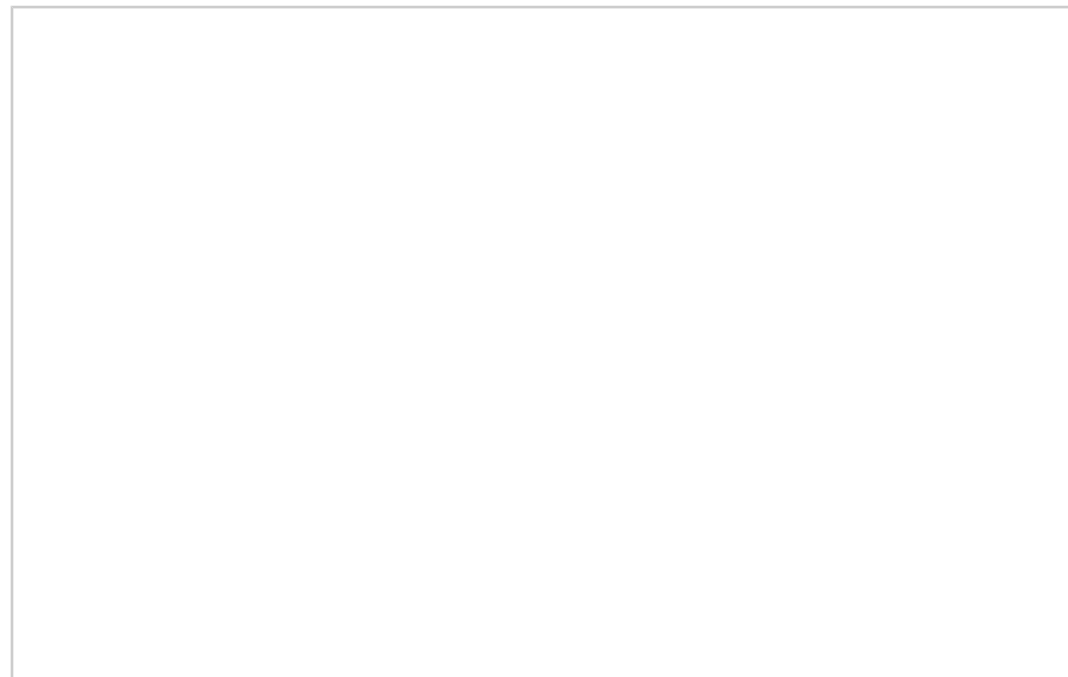
[La clase que describe el canvas](#)

Finalizamos el estudio del movimiento de una figura por el área de trabajo del applet, poniendo un panel con tres botones sobre el applet cuyo cometido será el de controlar la animación: empezar la animación, establecer una pausa o continuar la animación en estado de pausa, y finalmente, parar la animación.



anima3: [AnimaApplet3.java](#), [MiCanvas.java](#)

El diseño del applet



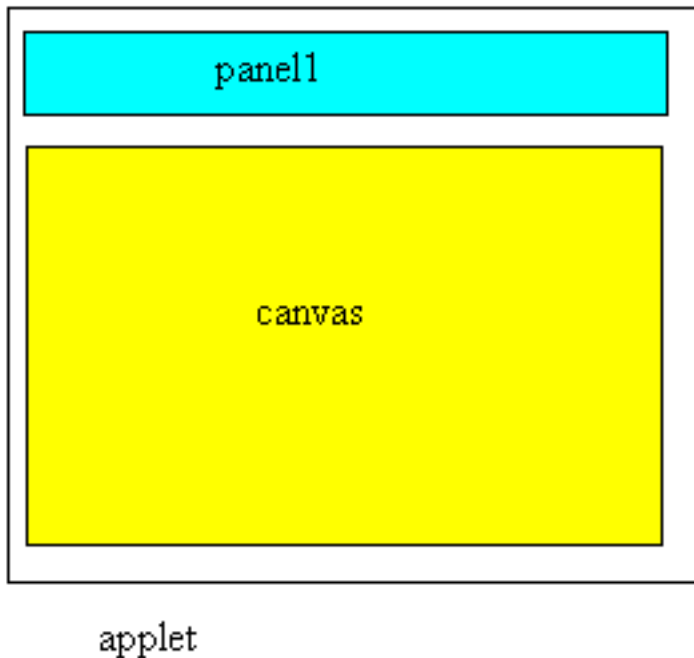


En la página anterior, el [movimiento de la pelota](#) se inicia cuando aparece la página que contiene el applet en la ventana del navegador, es decir, cuando se llama a la función *start* miembro de la clase que describe el applet. El movimiento se para cuando la página desaparece de la ventana del navegador, es decir, se llama a la función *stop* miembro de la clase que describe el applet.

En algunas aplicaciones, necesitamos que el usuario [controle la animación](#), es decir, que pueda iniciarla pulsando un botón, suspenderla pulsando otro botón, reanudarla, o que pueda pararla. Para este propósito, diseñamos un applet en cuya parte superior se dispone un panel en el que se sitúan tres botones titulados Empieza, Pausa y Para, y un canvas en la parte inferior ocupando la mayor parte de la superficie del applet.

Los pasos para crear el applet son los siguientes:

Cear el applet, y en el modo de diseño (pestaña **Design**) situar un panel en la parte superior del applet y tres botones sobre el panel.



Establecer [FlowLayout](#) como gestor de diseño del panel, de modo que los botones quedan centrados en el panel.

Establecer [BorderLayout](#) como gestor de diseño del applet, de modo que el panel quede al norte (NORTH)

Añadir al proyecto una clase derivada de [Canvas](#) denominada *MiCanvas*

En el modo código fuente (pestaña **Source**), crear un objeto de la clase *MiCanvas* y añadir dicho objeto al applet en la posición central (CENTER).

```
this.add(panell, BorderLayout.NORTH);
this.add(canvas, BorderLayout.CENTER);
```

En modo diseño, hacer doble-clic sobre cada uno de los botones. JBuilder genera el código de la función respuesta ([Anonymous Adapter](#)), asociando mediante *addActionListener* el control botón con un objeto de una clase anónima que implementa el interface *ActionListener* y que define *actionPerformed*.

Los botones que controlan la animación

Cuando se pulsa el botón titulado Empieza se llama a la función respuesta *btnEmpezar_actionPerformed*. La tarea que realiza esta función es similar a la [función start](#) miembro de la clase que describe el applet: crea un subproceso u objeto *anima* de la clase *Thread*, si no existe, y lo pone en marcha, llamando a su función miembro *start*.

```
void btnEmpezar_actionPerformed(ActionEvent e) {
//...
    bPausa=true;
    if(anima ==null){
        anima=new Thread(this);
        anima.start();
    }
}
```

Cuando se pulsa el botón Para, se llama a la función respuesta *btnPara_actionPerformed*, que realiza la misma tarea que la [función stop](#) miembro de la clase que describe el applet: para el subproceso *anima* llamando a su función miembro *stop*.

```
void btnPara_actionPerformed(ActionEvent e) {
    if(anima!=null){
        anima.stop();
        anima=null;
    }
}
```

Cuando se pulsa el botón titulado Pausa se llama a la función respuesta *btnPausa_actionPerformed*. La variable de control *bPausa* de tipo **boolean** que inicialmente toma el valor **true**, cambia a **false** y a la vez, el título del botón cambia de Pausa a Continua. Al pulsar de nuevo el botón, *bPausa* que vale **false** cambia a **true**, y el título del botón vuelve a ser Pausa.

```
void btnPausa_actionPerformed(ActionEvent e) {
    if(bPausa==true){
        btnPausa.setLabel("Continua");
        bPausa=false;
    }else{
        btnPausa.setLabel("  Pausa  ");
        bPausa=true;
    }
}
```

La definición de la [función run](#) es similar a la estudiada en la página anterior. La diferencia estriba en que el movimiento de la pelota tiene lugar en el canvas, y que solamente se mueve (se llama a la función miembro *mover*) si *bPausa* toma el valor **true**, es decir, no se ha pulsado el botón titulado Pausa.

```
public void run() {
    long t=System.currentTimeMillis();
    while (true) {
        if(bPausa){
            canvas.mover();
        }
        try{
            t+=retardo;
            Thread.sleep(Math.max(0, t-System.currentTimeMillis()));
        }catch(InterruptedException ex){
            break;
        }
    }
}
```

La clase que describe el canvas

El movimiento de la pelota no tiene lugar en la ventana del applet, sino en un canvas. Copiamos del ejemplo anterior AnimaApplet2.java la [definición de mover](#) y la [redefinición de update](#) y la pegamos en la clase que describe el canvas denominada *MiCanvas*.

La función miembro *inicio* que obtiene las dimensiones del canvas y la posición inicial de la pelota, de

forma similar a la función *init* del applet que vimos en la página anterior. La función *inicio* miembro de *MiCanvas* nos permitirá en otros programas inicializar el objeto *canvas* una vez creado, pasándole, por ejemplo, la posición inicial de la pelota que el usuario ha podido introducir en controles de edición.

El código completo de la clase que describe el canvas, es el siguiente.

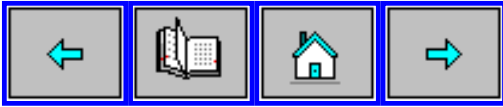
```
public class MiCanvas extends Canvas {
    int radio=10;           //radio de la pelota
    int x, y;               //posición del centro de la pelota
    int dx = 1;             //desplazamientos
    int dy = 1;
    int anchoCanvas;
    int altoCanvas;
    //Doble buffer
    Image imag;
    Graphics gBuffer;

    public MiCanvas() {
        setBackground(Color.white);
    }
    void inicio(){
    //dimensiones del canvas
        anchoCanvas=getSize().width;
        altoCanvas=getSize().height;
    //posición inicial de partida
        x=anchoApplet/4;
        y=altoApplet/2;
    }

    public void update(Graphics g){
        if(gBuffer==null){
            imag=createImage(anchoCanvas, altoCanvas);
            gBuffer=imag.getGraphics();
        }
        gBuffer.setColor(getBackground());
        gBuffer.fillRect(0,0, anchoCanvas, altoCanvas);
    //dibuja la pelota
        gBuffer.setColor(Color.red);
        gBuffer.fillOval(x-radio, y-radio, 2*radio, 2*radio);
    //transfiere la imagen al contexto gráfico del canvas
        g.drawImage(imag, 0, 0, null);
    }
    void mover() {
```

```
        x += dx;
        y += dy;
        if (x >= (anchoCanvas-radio) || x <= radio) dx*= -1;
        if (y >= (altoCanvas-radio) || y <= radio) dy*= -1;
        repaint();          //llama a update
    }
    public void paint (Graphics g) {
//se llama la primera vez que aparece el applet
    }
}
```

Una figura inmóvil que cambia de aspecto con el tiempo



[Subprocesos \(threads\)](#)

[Pasos para crear una animación](#)

[Cargando y mostrando las imágenes](#)

[La clase *MediaTracker*](#)

Aunque 12 imágenes por segundo son suficientes para producir la ilusión de movimiento, en el cine se emplean del orden de 24 y en la televisión del orden de 30. El número de imágenes por segundo que se pueden producir y mostrar con un ordenador dependen de su potencia de cálculo.

Pasos para crear una animación

Resumimos los pasos necesarios para crear una animación.

1. La clase que describe [el applet implementa el interface *Runnable*](#).

```
public class AnimaApplet4 extends Applet implements Runnable {  
    //...  
}
```

2. Se crea un subproceso, un objeto de la clase *Thread* y se pone en marcha llamando a su función miembro *start*. Dicho subproceso se puede crear y poner en marcha cuando aparece la página que contiene al applet en el navegador, o en respuesta a la pulsación de un botón titulado Empieza.

```
if (anima == null) {  
    anima = new Thread(this);  
    anima.start();  
}
```

3. Se para el subproceso llamando a su función miembro *stop*. Esta tarea se puede realizar cuando la página que contiene el applet desaparece del navegador, o en respuesta a la pulsación de un botón titulado Para.


```
if (anima != null) {  
    anima.stop();  
    anima = null;  
}
```

4. Ya que la clase que describe el applet implementa el interface *Runnable*, ha de definir la función miembro *run*, que tiene la siguiente definición

```
public void run() {  
    long t=System.currentTimeMillis();  
    while (true) {  
        //tarea a realizar ....mover una pelota, etc.  
        try{  
            t+=retardo;  
            Thread.sleep(Math.max(0, t-System.currentTimeMillis()));  
        }catch(InterruptedException ex){  
            break;  
        }  
    }  
}
```

5. Para evitar el parpadeo se redefine la función *update*, que emplea la técnica conocida por [double-buffer](#). Esta técnica requiere cuatro pasos:

- Crea un contexto gráfico en memoria *gBuffer*, y una imagen *imag* de las mismas dimensiones que el componente.
- Borra dicho contexto con el color de fondo del componente.
- Dibuja sobre dicho contexto gráfico en memoria *gBuffer*
- Transfiere la imagen en memoria *imag* al contexto gráfico del componente *g*, mediante *drawImage*.

```
public void update(Graphics g) {  
    if (gBuffer == null) {  
        imag = createImage(ancho, alto);  
        gBuffer = imag.getGraphics();  
    }  
  
    gBuffer.setColor(getBackground());  
    gBuffer.fillRect(0, 0, ancho, alto);  
    gBuffer.setColor(Color.black);  
  
    //aquí se dibuja sobre el contexto gráfico en memoria gBuffer  
    //....  
  
    g.drawImage(imag, 0, 0, null);  
}
```

Cargando y mostrando las imágenes

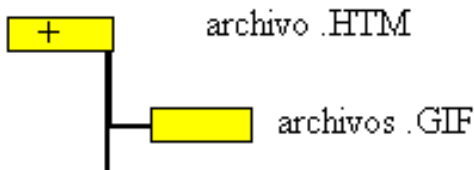


anima4: [AnimaApplet4.java](#), [0.gif](#), [1.gif](#), [2.gif](#), [3.gif](#), [4.gif](#), [5.gif](#), [6.gif](#), [7.gif](#), [8.gif](#), [9.gif](#)



En este ejemplo, vamos a ver lo específico de cada tipo de animación. En este caso es un contador. Los números del cero al nueve, se representan por pequeñas imágenes que se han guardado en archivos .GIF.

El primer paso, será cargar las imágenes mediante *getImage* y guardarlas en un array de objetos de la clase *Image*. Las imágenes las hemos situado en un subdirectorío denominado contadorGif por debajo de la ubicación del archivo HTM que guarda la página que contiene el applet. El nombre de los archivos como podemos apreciar en la porción de código es 1.gif, 2.gif ... 9.gif



```
for (int i = 0; i < 10; i++) {  
    numeros[i] = getImage(getDocumentBase(), "contadorGif/" + i + ".gif");  
}
```

En el bucle **while** de la función miembro *run* se incrementa el contador de imágenes. Cuando el contador *indice* sobrepasa el número nueve comienza con el cero, y así se repite el ciclo.

```
public void run() {  
    while (true) {  
        if (++indice > numeros.length) {  
            indice = 0;  
        }  
        repaint();  
        //...  
    }  
}
```

Cuando hay subprocesos corriendo, no podemos sustituir las sentencias

```
if (++indice > numeros.length) {  
    indice = 0;  
}
```

Por las sentencias

```
    indice++;
    if (indice>= numeros.length){
        indice = 0;
    }
```

Sino por las sentencias

```
    synchronized (this) {
        indice++;
        if (indice>= numeros.length) {
            indice=0;
        }
    }
```

El significado de la palabra reservada [synchronized](#) lo hemos estudiado en este capítulo.

Finalmente, *repaint* llama a *update* para dibujar la imagen guardada en el elemento *indice* del array *numeros* en el contexto gráfico en memoria *gBuffer* empleando la técnica del double-buffer.

```
public void update(Graphics g) {
    //...
    gBuffer.drawImage(numeros[indice], 0, 0, this);
    //...
}
```

La clase *MediaTracker*

Como podemos apreciar al visitar una página web, una imagen estática grande se va mostrando a medida que se va cargando. En una animación con un conjunto de imágenes esto no es posible, tenemos que disponer de todas las imágenes antes de correr (*run*) la animación.

El retraso en la transmisión es el tiempo que transcurre hasta que un determinado objeto ha sido transferido a través de la red Internet. La clase *MediaTracker* nos ayuda a resolver los problemas relacionados con el retraso en la transmisión de imágenes, sonido u otros elementos multimedia.

Para disponer de todas las imágenes antes de empezar la animación, en *init* creamos un objeto *tracker* de la clase *MediaTracker*.

```
tracker = new MediaTracker(this);
```

A medida que vamos cargando las imágenes mediante *getImage*, las vamos añadiendo al objeto *tracker*, mediante *addImage*. El segundo argumento, 0, es el identificador de un grupo de imágenes.

```
for (int i = 0; i < 10; i++) {  
    numeros[i] = getImage(getDocumentBase(), "contadorGif/" + i + ".gif");  
    tracker.addImage(numeros[i], 0);  
}
```

Nos aseguramos de que todas las imágenes, cuyo identificador es 0, se ha cargado mediante las siguientes líneas de código

```
try {  
    tracker.waitForID(0);  
} catch (InterruptedException ex) {  
    return;  
}
```

El código fuente completo, es el siguiente

```
public class AnimaApplet4 extends Applet implements Runnable {  
    Image[] numeros = new Image[10];  
    Thread  anima;  
    MediaTracker  tracker;  
    int retardo = 800;  
    int indice = 0;  
    //doble buffer  
    Image imag;  
    Graphics gBuffer;  
  
    public void init() {  
        // carga las imágenes  
        tracker = new MediaTracker(this);  
        for (int i = 0; i < 10; i++) {  
            numeros[i] = getImage(getDocumentBase(), "contadorGif/" + i + ".gif");  
            tracker.addImage(numeros[i], 0);  
        }  
    }  
  
    public void start() {  
        if (anima == null) {  
            anima = new Thread(this);  
            anima.start();  
        }  
    }  
  
    public void stop() {  
        if (anima != null) {  
            anima.stop();  
            anima = null;  
        }  
    }  
}
```

```

    }

    public void run() {
//han de estar completamente cargadas las imágenes antes de empezar a correr la
animación
        try {
            tracker.waitForID(0);
        } catch (InterruptedException ex) {
            return;
        }
// corre la animación
        long t = System.currentTimeMillis();
        while (true) {
            synchronized (this) {
                indice++;
                if (indice>=numeros.length) {
                    indice=0;
                }
            }
            repaint();

            try {
                t+=retardo;
                Thread.sleep(Math.max(0, t - System.currentTimeMillis()));
            } catch (InterruptedException e) {
                break;
            }
        }
    }

    public void update(Graphics g) {
        int ancho=getSize().width;
        int alto=getSize().height;
        if (gBuffer == null) {
            imag = createImage(ancho, alto);
            gBuffer = imag.getGraphics();
        }

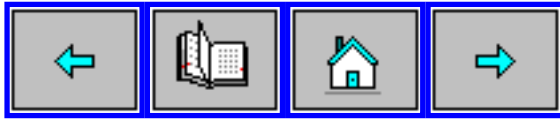
        gBuffer.setColor(getBackground());
        gBuffer.fillRect(0, 0, ancho, alto);
        gBuffer.setColor(Color.black);
// muestra las imágenes
        gBuffer.drawImage(numeros[indice], 0, 0, this);
// Transfiere la imagen off a al contexto gráfico del applet
        g.drawImage(imag, 0, 0, null);
    }

    public void paint(Graphics g) {
        Font font = new Font("TimesRoman", Font.BOLD, 16);

```

```
g.setFont(font);
FontMetrics fm = g.getFontMetrics(font);
String texto;
if ((tracker.statusID(0, true) == MediaTracker.LOADING)) {
//se están cargando las imágenes
    texto="Cargando las imágenes ....";
    g.drawString(texto, (getSize().width - fm.stringWidth(texto)) / 2,
((getSize().height - fm.getHeight()) / 2) + fm.getAscent());
}
}
}
```

Error en las medidas directas



[Ejemplos completos](#)

[Planteamiento del problema](#)

[Diseño del applet](#)

[Una clase para el tratamiento de los datos](#)

[Conclusiones](#)

[El código fuente](#)

Planteamiento del problema

Si al tratar de determinar una magnitud por [medida directa](#) realizamos varias medidas con el fin de corregir los errores aleatorios, y los resultados obtenidos son $x_1, x_2, x_3 \dots x_n$, se adopta como mejor estimación del valor verdadero el valor medio $\langle x \rangle$ que viene dado por

$$\langle x \rangle = \frac{\sum_{i=1}^n x_i}{n}$$

De acuerdo con la teoría de Gauss de los errores, que supone que estos se producen por causas aleatorias, se toma como la mejor estimación del error, para el valor experimental, obtenido como valor medio de un conjunto n de medidas, el llamado error cuadrático Δx definido por

$$\Delta x = \sqrt{\frac{\sum_{i=1}^n (x_i - \langle x \rangle)^2}{n(n-1)}}$$

El resultado del experimento se expresa como $\langle x \rangle \pm \Delta x$

Diseño del *applet*

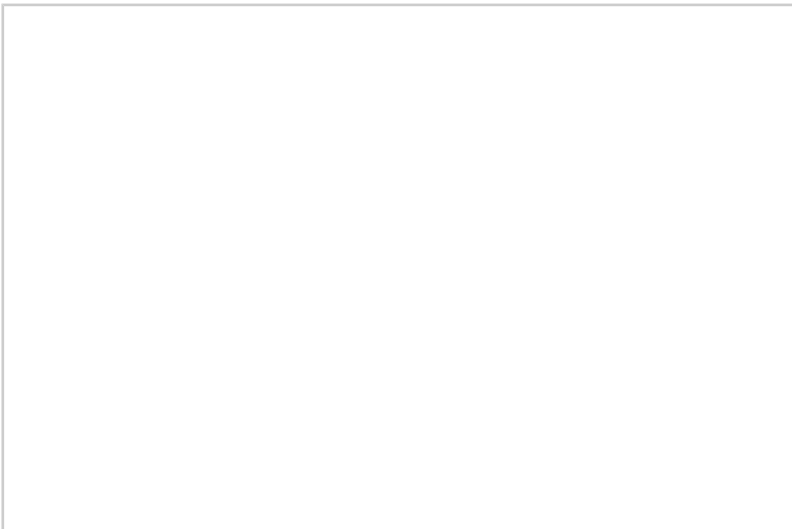
En primer lugar, hemos de pensar en identificar y separar, los distintos elementos que entran en un proyecto, en este caso, el interfaz o medio de comunicación entre el usuario y el programa, y el procedimiento de tratamiento de los datos.

Nuestro proyecto consistirá básicamente en dos archivos: el primero describirá el interfaz mediante una clase derivada (o subclase) de la clase *Applet*. Y el segundo, describirá el procedimiento numérico mediante una clase que denominaremos *Error*.

Disposición de los componentes en el *applet*

Antes de decidir qué controles se van a utilizar y su disposición en la ventana del applet, hemos de pensar cómo va a ser la interacción entre el usuario y el programa. En primer lugar, introducirá los datos, a continuación pulsará un botón, y finalmente, aparecerán los resultados (la medida y el error).

- Los datos se introducirán en un control área de texto, un simple editor multilínea. Cada línea de texto contendrá un dato numérico
- Se mostrarán los resultados en controles de edición no editables. Delante de los controles pondremos una etiqueta (*Label*) para señalar qué resultado se refiere a la medida y cual al error.



En el applet, se muestra la disposición de los controles, a la izquierda el control área de texto, a la derecha los controles de edición y los botones titulados **Calcular** y **Borrar**.

Hay muchas maneras de disponer los controles, de acuerdo con el gusto personal y la experiencia del programador. En este caso, hemos dividido la ventana del applet en dos partes, la parte izquierda la ocupa el control área de texto *tDatos* , y la parte derecha un panel *Panel1* que contiene al resto de los controles. Se ha empleado [GridLayout](#) como gestor de diseño para estos dos componentes.

```
this.setLayout(gridLayout1);
this.add(tDatos, null);
this.add(Panell1, null);
```

Para disponer el resto de los controles sobre el *Panel1*, se ha empleado el gestor de diseño [GridBagLayout](#). Se crean dos objetos, uno *gbc1* de la clase *GridBagConstraints*, y otro *gbl1* de la clase *GridBagLayout*

```
GridBagLayout gbl1 = new GridBagLayout();
GridBagConstraints gbc1=new GridBagConstraints();
```

Se establece el gestor de diseño para este panel, mediante la función miembro *setLayout*

```
Panell1.setLayout(gbl1);
```

A continuación, se sitúan los controles sobre dicho panel, utilizando las distintas propiedades (*constraints*) que definen este complicado gestor de diseño.

```
//panel 1
//primera fila
    Panell1.setLayout(gbl1);
    gbc1.anchor=GridBagConstraints.WEST;
    gbc1.gridwidth=1;
    gbc1.insets=new Insets(5,0,5,0);
    Panell1.add(label1, gbc1);
    gbc1.gridwidth=GridBagConstraints.REMAINDER;
    Panell1.add(tMedio, gbc1);

//segunda fila
    gbc1.anchor=GridBagConstraints.WEST;
    gbc1.gridwidth=1;
    gbc1.insets=new Insets(5,0,5,0);
    Panell1.add(label2, gbc1);
    gbc1.gridwidth=GridBagConstraints.REMAINDER;
    Panell1.add(tError, gbc1);

//tercera fila
    gbc1.anchor=GridBagConstraints.CENTER;
```

```

gbc1.gridwidth=1;
gbc1.insets=new Insets(25,0,5,0);
Panel1.add(btnCalcular, gbc1);
gbc1.gridwidth=GridBagConstraints.REMAINDER;
Panel1.add(btnBorrar, gbc1);

```

En los [controles de edición](#) *tMedio* y *tError* se muestran los resultados del cálculo, por lo que no precisan ser editados o modificados por el usuario, para ello se pone su propiedad *Editable* en *false*.

```
tMedio.setEditable(false);
```

Respuesta a las acciones del usuario.

Una vez que se ha diseñado el interfaz, se definen las [funciones respuesta a las acciones del usuario sobre los botones](#). Para cada uno de los botones se crea un objeto de una clase anónima interna que maneja las acciones del usuario sobre cada botón.

```

btnCalcular.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        btnCalcular_actionPerformed(e);
    }
});

```

A continuación, se define la función respuesta *btnCalcular_actionPerformed* que realiza varias tareas:

1. Lee los datos que se han introducido en el control área de texto *tDatos*,

```
String entrada=tDatos.getText();
```

2. Crea el objeto *error* de la clase *Error* e inicializa sus miembros dato.

```

Error error=new Error();
error.setDatos(entrada);

```

2. Llama desde este objeto a la función miembro *calcular* que calcula el valor medio y el error absoluto.

```
error.calcular();
```

4. Muestra los resultados del cálculo en los controles de edición *tMedio* y *tError*, para ello, se formatean los resultados mostrándose hasta con 4 decimales mediante [la función *Math.round*](#), y se

convierten los números decimales en su representación textual, objetos de la clase *String*. La función *Math.round* devuelve un entero que se ha de promocionar a un **double** (casting), para poder efectuar la división.

```
(double)Math.round(error.valorMedio*10000)/10000;
```

Alternativamente, se podría emplear la función *Math.floor* que devuelve un *double*

```
Math.floor(error.media*10000)/10000;
```

La diferencia estriba en que *Math.floor* expresa el número 0.45638021, tomando cuatro cifras decimales como 0.4563, mientras que empleando la función *Math.round* se obtiene una mejor aproximación de la última cifra 0.4564 como cabría esperar.

Una clase para el tratamiento de los datos

Denominaremos *Error* a la clase que emplearemos para realizar el tratamiento de los datos. Dicha clase tendrá como miembros, el número n de datos que se va a tratar, un array x que guardará dichos datos numéricos, y finalmente, dos miembros que guardarán los resultados denominados *valorMedio* y *errorAbsoluto*.

La función miembro *setDatos*, inicializa el array x , y la función miembro *calcular* realiza el cálculo del valor medio y del error absoluto de acuerdo con las fórmulas mencionadas en el primer apartado.

Los datos se introducen en el control área de texto *tDatos*. Leemos los datos introducidos mediante la función *getText* que devuelve un string. La inicialización del array x se realiza a través del objeto de la clase *String* que se le pasa a la función miembro *setDatos*.

La clase [*StringTokenizer*](#) nos rompe el texto extraído del control área de texto en trozos (tokens) que se guardan en el string local *subTexto*, representando cada trozo un dato que hemos introducido. El separador, un retorno de carro, se introduce en el segundo argumento de su constructor.

```

public class Error {
//otros miembros...
    boolean setDatos(String texto){
        StringTokenizer t=new StringTokenizer(texto,"\n");
        n=t.countTokens();
        if (n<3) return false;
        x=new double[n];
        int i=0;
        while(t.hasMoreTokens()){
            String subTexto=(String)t.nextToken();
            x[i]=Double.valueOf(subTexto.trim()).doubleValue();
            i++;
        }
        return true;
    }
}

```

La medida y el error

Una vez creado e inicializado el array x . La función miembro *calcular* no reviste dificultad alguna, ya que para definirla basta traducir las fórmulas matemáticas a código.

```

public class Error {
//otros miembros...
    void calcular(){
        valorMedio=0.0;
        for(int i=0; i<n; i++){
            valorMedio+=x[i]/n;
        }
        double desviacion=0.0;
        for(int i=0; i<n; i++){
            desviacion+=(x[i]-valorMedio)*(x[i]-valorMedio);
        }
        errorAbsoluto=Math.sqrt(desviacion/(n*(n-1)));
    }
}

```

Los miembros dato *valorMedio* y *errorAbsoluto*, guardan los resultados. Para que los valores que guardan sean conocidos desde la clase que describe el applet, se pueden emplear dos alternativas, definir dos funciones miembro *getMedia* y *getError*, que devuelvan dichos datos o bien, no declarar privados (**private**) a los miembros dato *valorMedio* y *errorAbsoluto*

```
error.valorMedio;  
error.getMedia();
```

La segunda opción, está de acuerdo con el espíritu de la Programación Orientada a Objetos, una de cuyas características básicas es la encapsulación, que trata de restringir en lo posible el [acceso a la información que guardan los objetos](#) de dicha clase mediante ciertos modificadores denominados **private**, **protected**, **public** o **default** (cuando no se pone modificador).

Conclusiones

Varios son los aspectos de la programación Java tratados en este capítulo, entre los que cabe destacar

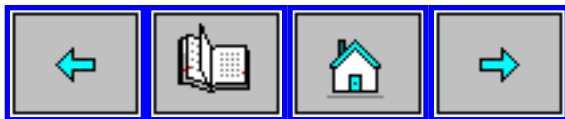
- La identificación y la separación de los distintos elementos que entran en un proyecto.
- El diseño del interfaz gráfico de usuario (GUI)
- La definición de las funciones respuesta a las acciones del usuario sobre los controles.
- Trabajar con la clase *StringTokenizer*, para extraer una colección de datos numéricos guardados en un string.
- La diferencia entre las funciones *Math.floor* y *Math.round* a la hora de presentar los resultados con un número prefijado de decimales.
- Traducir las fórmulas matemáticas a código

El código fuente



error: [ErrorApplet.java](#), [Error.java](#)

Conversión de unidades



[Ejemplos completos](#)

[Introducción](#)

[Un grupo de botones de radio](#)

[Disposición de los controles empleando el gestor *GridBagLayout*](#)

[Respuesta a la pulsación sobre un botón](#)

[Verificación de los datos introducidos en controles de edición.](#)

[Respuesta a las acciones sobre un grupo de controles botón de radio.](#)

[El código fuente](#)

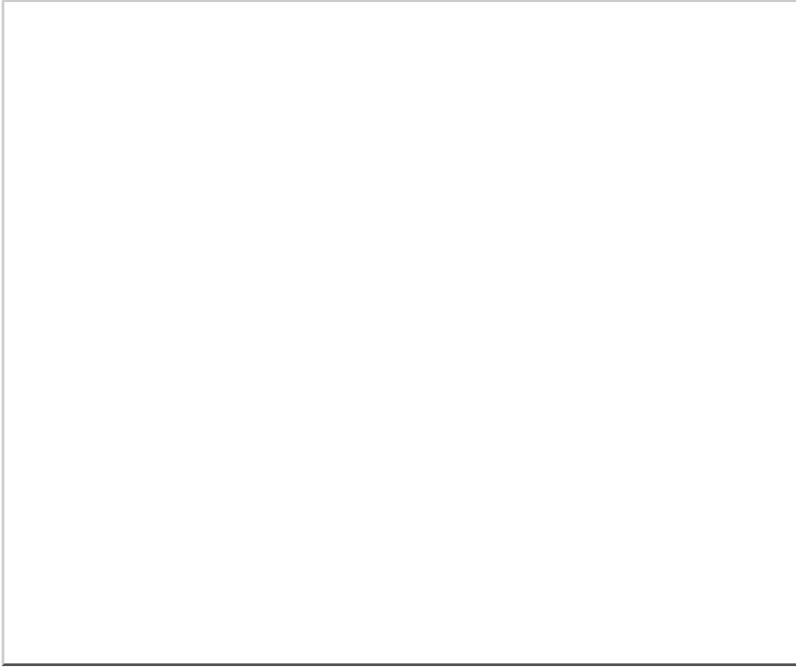
Introducción

El applet que se explica en esta página web, tiene como objetivo convertir cantidades expresadas en unidades empleadas habitualmente en Termodinámica en cantidades expresadas en unidades del Sistema Internacional. Dicho applet se emplea en el capítulo [Termodinámica](#) del Curso Interactivo de Física.

Para pasar desde el Sistema Internacional de Unidades al sistema ordinario de unidades empleado en Termodinámica, se introduce la cantidad a convertir en el control de edición situado en la parte superior izquierda del applet. Si se selecciona la unidad de origen pulsando el botón de radio situado en el panel izquierdo, la unidad de destino queda automáticamente seleccionada, salvo en el caso de la energía en la que hay una doble opción, atmósferas por litro (por defecto) o calorías. Finalmente, se pulsa el botón titulado >>>. La cantidad convertida aparece en el control de edición situado en la parte superior derecha del applet.

Para convertir desde el sistema ordinario de unidades empleado en Termodinámica al Sistema Internacional, se procede de modo inverso. Se introduce la cantidad a convertir en el control de edición situado en la parte superior derecha del applet, se elige la unidad de origen pulsando en el botón de radio

correspondiente a dicha unidad, la unidad de destino queda automáticamente seleccionada. Finalmente, se pulsa el botón titulado <<<<. La cantidad convertida aparece en el control de edición situado en la parte superior izquierda del applet.



Un grupo de botones de radio

Para [crear un grupo de botones de radio](#), se procede del siguiente modo

Se crea un array de objetos *Checkbox*

```
Checkbox chkIzq[ ]=new Checkbox[4];
```

Se crea un objeto de la clase *CheckboxGroup*

```
CheckboxGroup chkGrupoIzq=new CheckboxGroup( );
```

En *init* o bien *jbInit* se crean cada uno de los botones de radio

```
for(int i=0; i<4; i++){  
    chkIzq[i]=new Checkbox( );  
}
```

Se pone una etiqueta a cada uno de los botones de radio

```
chkIzq[0].setLabel("Energía (J)");
chkIzq[1].setLabel("Presión (Pa)");
chkIzq[2].setLabel("Volumen (m3)");
chkIzq[3].setLabel("Temperatura (°K)");
```

Se asocia cada botón de radio a su grupo

```
for(int i=0; i<4; i++){
    chkIzq[i].setCheckboxGroup(chkGrupoIzq);
}
```

Se establece el estado inicial del grupo de botones de radio (recuérdese que sólo uno de los botones de radio de un grupo puede estar activado, *checked*)

```
chkGrupoIzq.setSelectedCheckbox(chkIzq[0]);
```

Disposición de los controles empleando el gestor *GirdBagLayout*

En el applet se han situado dos controles de edición, dos grupos de botones de radio y dos botones. Para situar estos controles se ha empleado el gestor de diseño [*GridBagLayout*](#), el más versátil y a la vez más complicado de la AWT.

La disposición de los botones de radio no es simétrica, lo que añade una mayor dificultad al manejo de este gestor de diseño. Los botones correspondientes a la misma clase de unidad deben de estar uno enfrente del otro. Esto ocurre con el volumen, la presión y la temperatura, pero no ocurre con la energía. En el Sistema Internacional de Unidades la energía se expresa en Julios, pero en Termodinámica se expresa en atmósferas por litro y en calorías. El gestor de diseño *GridBagLayout* nos permite poner un botón de radio en la primera columna enfrente de dos botones de radio en la segunda columna. Las líneas de código que lo hacen posible están marcadas en negrita.


```
this.add(Panel2, BorderLayout.CENTER);  
//Panel2  
Panel2.setLayout(gbl1);  
gbc1.anchor=GridBagConstraints.CENTER;  
gbc1.insets=new Insets(5,0,10,0);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(labelControl1, gbc1);  
  
gbc1.anchor=GridBagConstraints.CENTER;  
gbc1.insets=new Insets(0,0,5,0);  
gbc1.gridwidth=1;  
Panel2.add(tIzquierda, gbc1);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(tDerecha, gbc1);  
  
gbc1.anchor=GridBagConstraints.WEST;  
gbc1.insets=new Insets(0,5,0,5);  
gbc1.gridwidth=1;  
Panel2.add(chkIzq[0], gbc1);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(chkDcha[0], gbc1);  
  
gbc1.gridx=1;  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
gbl1.setConstraints(chkDcha[1], gbc1);  
Panel2.add(chkDcha[1]);  
  
gbc1.gridwidth=1;  
gbc1.gridx=GridBagConstraints.RELATIVE;  
Panel2.add(chkIzq[1], gbc1);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(chkDcha[2], gbc1);  
  
gbc1.gridwidth=1;  
Panel2.add(chkIzq[2], gbc1);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(chkDcha[3], gbc1);  
  
gbc1.gridwidth=1;  
Panel2.add(chkIzq[3], gbc1);  
gbc1.gridwidth=GridBagConstraints.REMAINDER;  
Panel2.add(chkDcha[4], gbc1);  
  
gbc1.anchor=GridBagConstraints.CENTER;
```

```

gbc1.gridwidth=1;
gbc1.insets=new Insets(10,0,5,0);
Panel2.add(btnIzquierda, gbc1);
gbc1.gridwidth=GridBagConstraints.REMAINDER;
Panel2.add(btnDerecha, gbc1);

```

Respuesta a la pulsación sobre un botón

El compilador JBuilder nos genera automáticamente, el nombre de la función respuesta a la [acción del usuario sobre un botón](#).

```

btnDerecha.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        btnDerecha_actionPerformed(e);
    }
});

```

//.....

El código de la función respuesta *btnDerecha_actionPerformed*, realiza las siguientes tareas:

- Obtiene el texto del [control de edición](#) situado a la derecha, lo convierte en un número decimal de doble precisión, y lo guarda en la variable local *dcha*.

```
double dcha=Double.valueOf(tDerecha.getText()).doubleValue();
```

- Obtiene el nombre de la magnitud a convertir, a partir del [botón de radio del grupo](#) de la derecha actualmente en estado activado.

```
Checkbox seleccionado=chkGrupoDcha.getSelectedCheckbox();
```

- Calcula la cantidad correspondiente a la magnitud en el sistema de unidades de la izquierda y lo guarda en la variable local *izq*.

```

double izq;
if(i==4){
    izq=dcha+factorDchaIzq[i];
}

```

```

    }else{
        izq=dcha*factorDchaIzq[i];
    }

```

- Muestra dicha cantidad en el control de edición situado a la izquierda

```
tIzquierda.setText(String.valueOf(izq));
```

```

void btnDerecha_actionPerformed(ActionEvent e) {
    double dcha=Double.valueOf(tDerecha.getText()).doubleValue();
    Checkbox seleccionado=chkGrupoDcha.getSelectedCheckbox();
    int i=0;
    for(i=0; i<5; i++){
        if (seleccionado.equals(chkDcha[i])){
            break;
        }
    }
    double izq;
    if(i==4){
        izq=dcha+factorDchaIzq[i];
    }else{
        izq=dcha*factorDchaIzq[i];
    }
    tIzquierda.setText(String.valueOf(izq));
}

```

Verificación de los datos introducidos en controles de edición

Para verificar que se introduce un número en el control de edición se siguen los mismos pasos que se explicaron en la página titulada [verificación de la información que introduce el usuario en un control de edición](#)

Definimos una clase que implementa el interface *FocusListener* y redefinimos la función miembro *focusLost* en la que estamos interesados.

```

class ValidaDouble extends FocusAdapter{
    public void focusLost(FocusEvent ev){
        TextField tEntrada=(TextField)(ev.getSource());
        try{
            Double.valueOf(tEntrada.getText()).doubleValue();
        }catch(NumberFormatException e){
            tEntrada.requestFocus();
            tEntrada.selectAll();
        }
    }
}

```

El objeto *ev* de la clase *FocusEvent* guarda toda la información relativa al suceso, y en particular, la fuente o el control que lo ha generado, dicha información se extrae mediante la función *getSource*.

Una vez que sabemos el control de edición *tEntrada* que ha generado el suceso, se obtiene el texto, un objeto de la clase *String*, que se ha introducido en dicho control mediante la función *getText*. Posteriormente, se convierte el texto (los caracteres) en un número en doble precisión mediante la función *doubleValue*.

Si el proceso de conversión falla, por que se han introducido caracteres que no son números, o una coma en vez de un punto, se produce un excepción del tipo *NumberFormatException*. En este caso, el control de edición vuelve a tomar el foco *requestFocus*, y se selecciona todos los caracteres introducidos para que puedan ser borrados de una vez pulsando la tecla Retroceso.

```

tEntrada.requestFocus();
tEntrada.selectAll();

```

Asociamos el control de edición *tIzquierda* con el objeto *valDouble* de la clase *ValidaDouble* que gestiona los sucesos que se producen en dicho control de edición.

```

ValidaDouble valDouble=new ValidaDouble();
tIzquierda.addFocusListener(valDouble);

```

Respuesta a las acciones sobre un grupo de controles botón de radio

Para responder a las acciones sobre un control *Checkbox* se ha de implementar el interface *ItemListener*. La clase que implementa este interface redefine la función *itemStateChanged*. La información acerca del suceso viene encapsulada en el objeto *ev* de la clase *ItemEvent*, véase la [respuesta a las acciones del usuario sobre un grupo de botones de radio](#). En particular, la función miembro *getSource*, extrae de *ev* el control *Checkbox* que ha sido activado.

```
public class ListenerIzq implements ItemListener{
    public void itemStateChanged(ItemEvent ev){
        Checkbox seleccionado=(Checkbox)ev.getSource();
        int i=0;
        for(i=0; i<4; i++){
            if (seleccionado.equals(chkIzq[i])){
                break;
            }
        }
        chkDcha[i+1].setState(true);
    }
}
```

Una vez que conocemos qué magnitud de la izquierda que ha sido seleccionada, *itemStateChanged* activa el botón de radio correspondiente a la misma magnitud en la parte derecha del applet.

Tendremos que asociar ahora los controles *Checkbox* de la parte izquierda del applet con el objeto la clase *ListenerIzq* que maneja las acciones del usuario sobre dichos controles.

```
for(int i=0; i<4; i++){
    chkIzq[i].addItemListener(listenerIzq);
}
```

Todo lo que se ha descrito para los botones de radio situados en la parte izquierda del applet se repite con pequeñas modificaciones para los botones de radio situados en la parte derecha del applet.

El código fuente



[ConversionApplet.java](#)

Diagramas



[Ejemplo completos](#)

[Diseño del applet](#)

[La jerarquía de clases](#)

[La clase que describe el canvas](#)

[El código fuente](#)

Este ejemplo es muy instructivo, ya que se incluyen muchos aspectos que hemos estudiado: establecer una jerarquía de clases, el enlace dinámico, la respuesta a las acciones del usuario sobre los controles, el control canvas, la clase *StringTokenizer* que nos permite dividir un string en substrings, el diseño del applet empleando la técnica de paneles anidados. Como en otros ejemplos que hemos comentado es importante para el mantenimiento del código identificar las distintas partes que entran en el proyecto.



Diseño del applet

El applet está diseñado de modo que en la parte izquierda se introducen los datos en un control área de texto. Se introduce un dato, se pulsa la tecla Retorno o Enter, luego se introduce otro dato y así sucesivamente. Cada dato está en una fila y todos ellos en una única columna.

En la parte derecha, se sitúa un canvas en el que tiene lugar la representación gráfica de los datos, bien sea en forma de tarta o

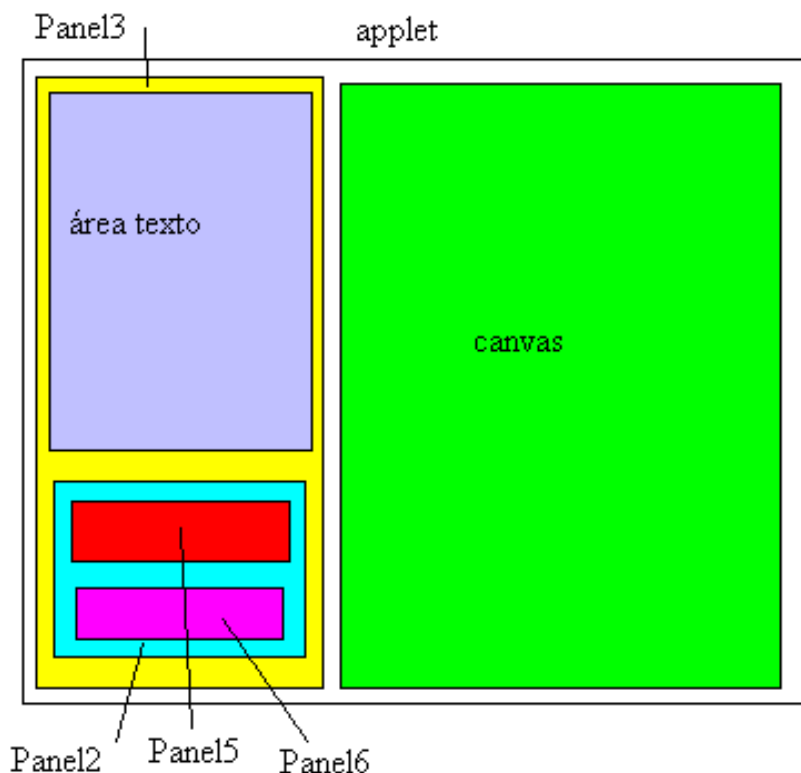
en forma de barras.

Debajo del control área de texto, se sitúan dos botones uno titulado Gráfica para procesar los datos introducidos y otro Borrar, para eliminar los datos del control área de texto, y prepararlo para introducir otros nuevos.

Debajo de estos botones, se sitúan un grupo de dos botones de radio que permite seleccionar el tipo de gráfico: diagrama de barras o gráfico en forma de tarta.

Para disponer los distintos controles se emplea la aproximación de paneles anidados tal como puee verse en la figura. Sobre el applet se sitúa el *Panel3* y el *canvas*. Sobre el *Panel3* se sitúa el control área de texto y el *Panel2*. Sobre el *Panel2* el *Panel5* y el *Panel6*. Se aplica el gestor de diseño [BorderLayout](#). Sobre el *Panel5* se colocan dos botones titulados Gráfica y Borrar, y sobre el *Panel6* los botones de radio titulados Barras y Tarta. Se disponen los controles sobre sus respectivos paneles aplicando el gestor de diseño [FlowLayout](#).

Todas estas operaciones se realizan con suma facilidad en el modo de diseño del IDE de JBuilder, si antes tenemos una idea previa de como va a quedar el applet, o disponemos de un esquema similar al de la figura adjunta.



Respuesta a las acciones del usuario sobre los controles

En modo diseño, se hace doble-clic sobre los botones Gráfica y Borrar. El IDE asocia cada botón a un [objeto de una clase anónima](#) que implementa el interface *ActionListener*. El programador solamente necesita escribir el código entre las llaves de apertura y cierre de la función respuesta cuyo nombre ha generado JBuilder.

En la función respuesta *btnGráfica_actionPerformed* a la pulsación sobre el botón titulado Gráfica, se obtiene el texto del [control área de texto](#) mediante *getText*, y se guarda en el string local *texto*. Se obtiene también, el [botón de radio](#) que está activado mediante la función *getSelectedCheckbox* de la clase *CheckboxGroup*. Finalmente, le pasamos al *canvas* los datos introducidos en el área de texto en forma de un string único, y el tipo de gráfico que se va a representar, un entero cuyo valor es cero o la constante *Grafico.BARRAS* o un uno, o la constante *Grafico.TARTA*. Ambas constantes son miembros estáticos

públicos de la clase *Grafico* que describiremos más adelante.

```
void btnGrafica_actionPerformed(ActionEvent e) {
    String texto=tDatos.getText();
    Checkbox radio=chkGrupo.getSelectedCheckbox();
    int tipo=(radio.equals(chkBarra)) ? Grafico.BARRAS : Grafico.TARTA;
    canvas.setDatos(texto, tipo);
}
```

La definición de la función respuesta *btnBorrar_actionPerformed* a la acción de pulsar sobre el botón titulado Borrar es muy simple, borra el texto del área de texto.

```
void btnBorrar_actionPerformed(ActionEvent e) {
    tDatos.setText(" ");
}
```

La jerarquía de clases

Podíamos diseñar una clase independiente que describiese cada uno de los tipos de gráficos. Pero es mucho más elegante establecer [una jerarquía de clases](#), ya que las clases *GraficoTarta* y *GarficoBarras* comparten los mismos datos y solamente se diferencian en el modo en el que presentan dichos datos. Por lo tanto, definimos una clase base abstracta denominada *Grafico*, que describe las características comunes a *GraficoTarta* y *GraficoBarras*, y declara una función abstracta *dibuja* que se define en cada una de las clases derivadas.

La clase base *Grafico* declara una serie de miembros dato, el más importante es el array *valores* que guarda los datos introducidos en el control área de texto, y el número *n* de datos. Para representar gráficamente los datos es necesario definir el área de la representación gráfica: un origen *x* e *y*, y unas dimensiones *ancho* y *alto*. Alternativamente, podemos encapsular estos cuatro datos en un objeto de la clase *Rectangle*.

Otros miembros de esta clase son constantes, se refieren al tipo de gráfico TARTA y BARRAS, al número máximo de datos MAXDATOS que es igual al máximo número de colores disponibles (esta restricción puede ser fácilmente levantada para representar cualquier número de datos). Finalmente, un array con los [colores](#) disponibles para representar los datos.

En el constructor de la clase *Grafico* se inicializan los miembros dato no estáticos. Se halla el valor máximo de los datos introducidos y se dividen entre el valor máximo. Se transforma así un conjunto de datos en otro equivalente cuyos valores están comprendidos entre 0.0 y 1.0.


```

public abstract class Grafico {
    public static final int TARTA=1;
    public static final int BARRAS=0;
    protected int x, y;                //posición del origen
    protected int ancho, alto;         //dimensiones del gráfico
    public static final int MAXDATOS=10;
    protected double[] valores=new double[MAXDATOS];
    protected int n;                   //número de valores
    //colores del gráfico
    protected static final Color colores[]={new Color(255,0,0), new Color(0, 255,0),
new Color(0, 0, 255),
        new Color(0, 255, 255), new Color(255, 0, 255), new Color(255, 255, 0),
        new Color(128, 0, 0), new Color(255, 128, 0),
        new Color(128, 128, 255), new Color(255, 0, 128)};

    public Grafico(int x, int y, int ancho, int alto, double[] valores) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
        this.valores=valores;
        n=valores.length;
        double maximo=0.0;
        for (int i=0; i<n; i++) {
            if (Math.abs(valores[i])>maximo){
                maximo = Math.abs(valores[i]);
            }
        }
        for (int i=0; i<n; i+=1) {
            valores[i] = valores[i]/maximo;
        }
    }
    public abstract void dibuja(Graphics g);
}

```

Las clases derivadas de *Grafico* definen la función *dibuja* declarada abstracta en la clase base. Las clases derivadas no declaran nuevos miembros dato, por lo que el constructor de la clase derivada se limita a llamar al constructor de la clase base.

Diagrama de barras

La clase derivada *GarficoBarras* hereda los miembros dato de la clase base y define un constructor, que se limita a llamar al constructor de la clase base.

La función miembro *dibuja*, divide el *ancho* del área de la representación gráfica entre los *n* datos, y lo guarda en la variable local *dx*, pero dibuja una barra cuya anchura *anchoBarra* que es las tres cuartas partes de *dx*. La altura de la barra *i* es proporcional a *valores[i]*, la altura de la barra que corresponde al valor máximo es igual al *alto* del área de la representación gráfica. Para dibujar las barras en el contexto gráfico *g* con apariencia plana se llama a la función *fillRect*, para que tenga un aspecto tridimensional, más estético, se llama a *fill3DRect*.

```

class GraficoBarras extends Grafico {
    public GraficoBarras(int x, int y, int ancho, int alto, double[] valores) {
        super (x, y, ancho, alto, valores);
    }
    public void dibuja (Graphics g) {
        int anchoBarra=3*ancho/(4*n);
        int dx = ancho/n;
        int h;
        for(int i=0; i<n; i++) {
            g.setColor (colores[i]);
            h=(int)(valores[i]*alto);
            g.fillRect (x+dx*i, y-h, anchoBarra, h, true);
        }
    }
}

```

Diagrama de tarta

La clase derivada *GraficoTarta* es similar a *GraficoBarras* solamente se diferencia en la definición de la función *dibuja*.

Para dibujar un diagrama de tarta, necesitamos conocer la suma de todos los valores que guarda el array *valores*. Para hallar el ángulo de cada porción de la tarta se hace una regla de tres: si al valor total le corresponden 360° a un valor concreto de índice *i*, *valores[i]*, le corresponde el valor que se guarda en la variable local *incAngulo*. Cada porción de la tarta se dibuja en el contexto gráfico *g* llamando a la función *fillArc*

```

class GraficoTarta extends Grafico {
    public GraficoTarta(int x, int y, int ancho, int alto, double[] valores) {
        super (x, y, ancho, alto, valores);
    }
    public void dibuja (Graphics g) {
        double total = 0.0;
        for (int i=0; i<n; i+=1) {
            total += valores[i];
        }
        int angulo = 0, incAngulo;
        for(int i=0; i<n; i++) {
            incAngulo=(int)(360.0*valores[i]/total);
            g.setColor (colores[i]);
            g.fillArc (x, y-alto, ancho, alto, angulo, incAngulo);
            angulo+=incAngulo;
        }
    }
}

```

La clase que describe el canvas

Definimos una clase denominada *MiCanvas* [derivada de Canvas](#), y creamos un objeto *canvas* de dicha clase en la función *init* de la clase que describe el applet. El *canvas* se añade al applet y se sitúa en la parte dercha del mismo.

```
this.add(Panel3, BorderLayout.WEST);
this.add(canvas, BorderLayout.CENTER);
```

En la [función repuesta a la pulsación sobre el botón](#) titulado Gráfica se pasa al *canvas*, por medio de la función miembro *setDatos*, información sobre los datos introducidos en un único string, y el tipo de gráfica (tarta o barras) para representar dichos datos.

La primera tarea de la función miembro *setDatos*, consistirá en romper el string texto, en un número de substrings igual al número de datos introducidos, y [convertir los strings resultantes en números](#) del tipo **double**. Para este propósito empleamos la clase [StringTokenizer](#) en la forma explicada en la página dedicada al estudio de esta clase.

El segundo paso, consiste en crear un objeto gráfico, cuya conducta sea la de representarse en el canvas. Dependiendo del tipo de gráfico elegido, se crea un objeto de la clase *GraficoBarras* o *GarficoTarta*. El valor devuelto por **new** al crear un [objeto de la clase derivada](#) se guarda en la variable *grafico* de la clase base *Grafico*.

```
if(tipo==Grafico.BARRAS){
    grafico=new GraficoBarras(anchoCanvas/8, 7*altoCanvas/8,
        3*anchoCanvas/4, 3*altoCanvas/4, datos);
}else{
    grafico=new GraficoTarta(anchoCanvas/8, 7*altoCanvas/8,
        3*anchoCanvas/4, 3*altoCanvas/4, datos);
}
```

Finalmente, se llama a *paint* para dibujar los gráficos. Ahora bien, hemos de tener cuidado, *paint* se llama cuando se crea el applet y se muestra en una ventana del navegador, cuando aún no hemos introducido ningún dato, ni hemos creado un objeto *grafico*, en esta circunstancia si escribimos

```
public void paint(Graphics g){
    grafico.dibuja(g);
}
```

El miembro dato *grafico* no ha sido aún inicializado, obteniendo un error cuando llama a la función *dibuja*. En la consola vemos el nombre de la [excepción NullPointerException](#) que ha sido lanzada. Para evitar este error definimos *paint* de la siguiente forma.

```
public void paint(Graphics g){
    if(grafico!=null){
        grafico.dibuja(g);
    }
}
```

El código completo de la clase *MiCanvas* que describe el control canvas es, el siguiente

```

public class MiCanvas extends Canvas {
    Grafico grafico;          //objeto gráfico

    public MiCanvas() {
        setBackground(Color.white);
    }
    void setDatos(String texto, int tipo){
        StringTokenizer t=new StringTokenizer(texto, "\n");
        int n=t.countTokens();
        double[] datos=new double[n];
        int i=0;
        while(t.hasMoreTokens()){
            String subTexto=(String)t.nextToken();
            if(i>=Grafico.MAXDATOS) break;
            datos[i]=Double.valueOf(subTexto.trim()).doubleValue();
            i++;
        }
//anchura y altura del canvas
        int anchoCanvas=getSize().width;
        int altoCanvas=getSize().height;
//objeto gráfico
        if(tipo==Grafico.BARRAS){
            grafico=new GraficoBarras(anchoCanvas/8, 7*altoCanvas/8, 3*anchoCanvas/4,
3*altoCanvas/4, datos);
        }else{
            grafico=new GraficoTarta(anchoCanvas/8, 7*altoCanvas/8, 3*anchoCanvas/4,
3*altoCanvas/4, datos);
        }
        repaint();
    }

    public void paint(Graphics g){
        if(grafico==null){
            Font font = new Font("TimesRoman", Font.BOLD, 16);
            g.setFont(font);
            FontMetrics fm = g.getFontMetrics(font);
            String texto="Introducir los datos";
            g.drawString(texto, (getSize().width - fm.stringWidth(texto)) / 2,
((getSize().height - fm.getHeight()) / 2) + fm.getAscent());
        }else{
            grafico.dibuja(g);
        }
    }
}

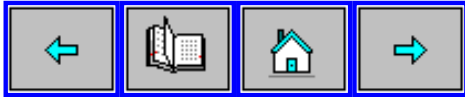
```

El código fuente



grafico1: [Grafico.java](#), [GraficoApplet1.java](#) [MiCanvas.java](#)

Representación gráfica de una función



[Ejemplos completos](#)

[Diseño del applet](#)

[Respuesta a las acciones del usuario sobre los controles](#)

[El área de la representación gráfica](#)

[Definición de la función](#)

[La clase que describe el canvas](#)

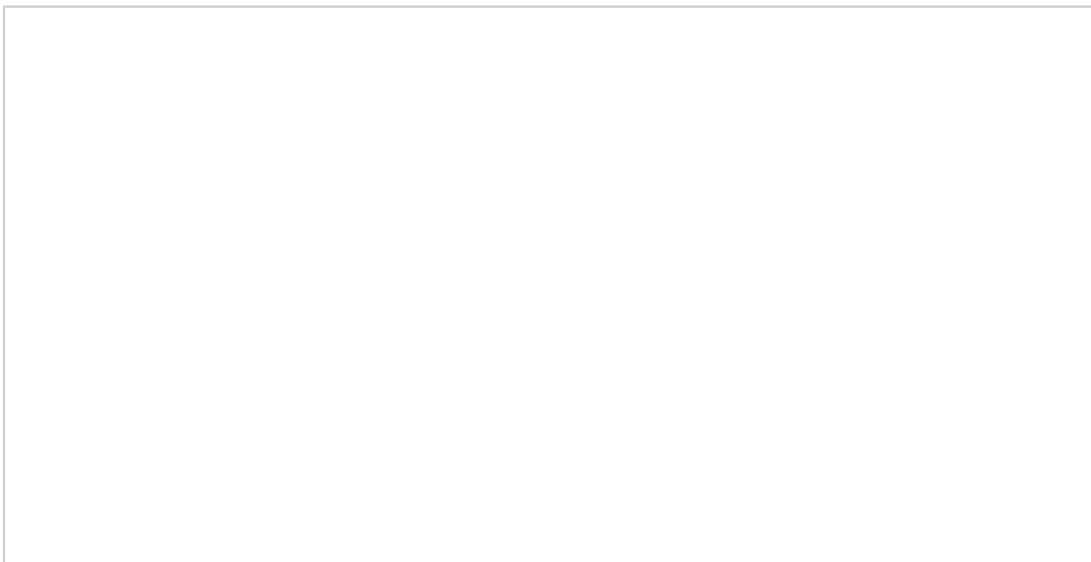
[El código fuente](#)

El objetivo de esta página es la de describir los pasos para representar una función. Sea la función

$$\frac{dn}{dv} = 4 \pi N \left(\frac{m}{2 \pi k T} \right)^{\frac{3}{2}} v^2 \exp \left(- \frac{mv^2}{2kT} \right)$$

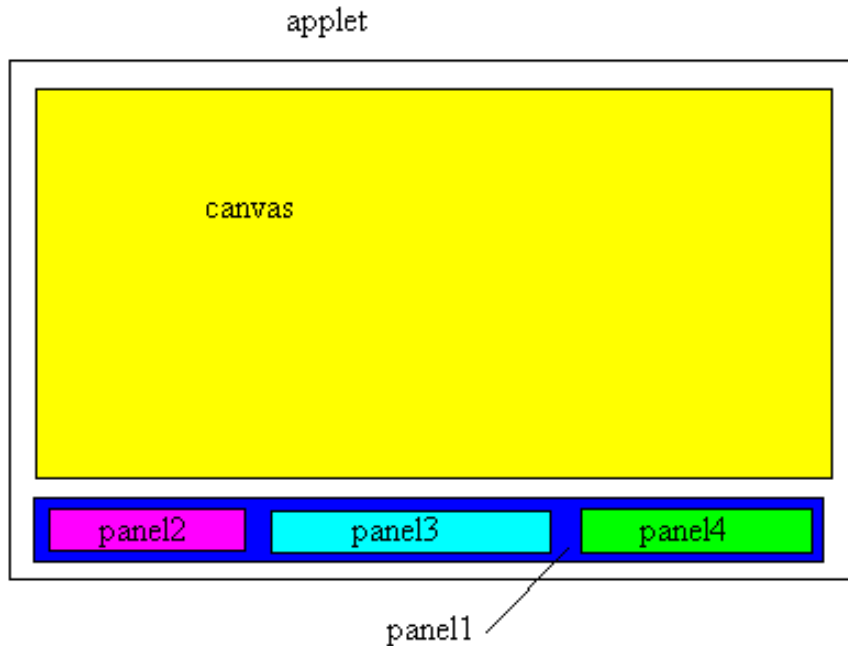
que es la fórmula de Maxwell para la [distribución de velocidades de las moléculas](#) de un gas ideal. Nos da el número dn de moléculas que se mueven con una velocidad comprendida entre v y $v+dv$ independientemente de la dirección del movimiento, cuando el gas ideal está a una temperatura T . m es la masa de las moléculas y k es la constante de Boltzmann $1.38 \cdot 10^{-23}$ J/K.

Queremos representar en el eje Y de las ordenadas el número dn de moléculas que se mueven con una velocidad comprendida entre v y $v+dv$, y en el eje X la velocidad de las moléculas, luego cambiaremos de temperatura o de gas y representaremos la nueva distribución en otro color.



Diseño del applet

Como en muchos otros applets se han creado dos clases, la que deriva de *Applet* y la que [deriva de Canvas](#). El applet describe el interfaz o comunicación entre el usuario y el programa, y en el canvas se muestran los resultados en forma de representación gráfica.



Se dispone el canvas en la parte superior del applet, y los controles sobre un panel en la parte inferior. Para disponer los controles se han empleado la aproximación de paneles anidados, como puede verse en el modo diseño del IDE de JBuilder.

Sobre el panel inferior se disponen tres paneles: en el panel de la izquierda se sitúan un control etiqueta (*Label*) y un control selección (*Choice*). En el panel central se dispone un control etiqueta (*Label*) y un control de edición (*Edit*). En el panel situado a la derecha se sitúan dos botones. El diseño del interfaz es muy rápido, si lo tenemos previamente trazado sobre papel o nos imaginamos la disposición de los controles en el applet.

Ponemos un control selección (*Choice*) en vez de un control lista para ahorrar el máximo espacio posible. Un control selección tiene una altura similar a un control de edición o a un botón, y se pueden disponer sin desentonar en una barra que incluya estos controles.

El [control selección \(Choice\)](#) contiene los nombre de varios gases ideales, y se inicializa de la siguiente forma

```
Choice chGases = new Choice();
//...
String str[]={"Hidrógeno (H2)", "Oxígeno (O2)", "Nitrógeno (N2)",
             "Helio (He)", "Neón (Ne)", "Argón (Ar)"};
for(int i=0; i<str.length; i++){
    chGases.addItem(str[i]);
}
```

El primer elemento de la lista aparecerá seleccionado por defecto. Inicializamos también el control de edición, de este modo el usuario, puede hacerse una idea acerca de los valores que puede introducir en dicho control

```
TextField tTemperatura = new TextField();
//...
tTemperatura.setColumns(5);
tTemperatura.setText("500");
```

Respuesta a las acciones del usuario sobre los controles

Haciendo doble-clic sobre cada uno de los botones JBuilder asocia el botón con un objeto una [clase anónima](#) que gestiona las acciones del usuario sobre dichos controles. El programador solamente tiene que preocuparse de introducir el código entre las llaves de apertura y cierre de la función respuesta.

Cuando pulsamos el botón titulado Gráfica se llama a la función respuesta *btnGrafica_actionPerformed*, se lee el texto del control de edición *tTemperatura* y [se convierte el string en un número](#), que se guarda en la variable local *temperatura*. Posteriormente, leemos el nombre del gas seleccionado en el control de selección *chGases*. En vez del nombre, obtenemos el índice mediante *getSelectedIndex*, el número entero lo guardamos en la variable local *indice*. Finalmente le pasamos estos dos datos al *canvas* mediante la llamada a *setNuevo*.

```
void btnGrafica_actionPerformed(ActionEvent e) {
    double temperatura=Double.valueOf(tTemperatura.getText()).doubleValue();
    int indice=chGases.getSelectedIndex();
    canvas.setNuevo(temperatura, indice);
}
```

Al pulsar en el botón Borrar se llama a la función respuesta *btnBorrar_actionPerformed*. Dentro de ella se hace una llamada a la función *paint* de la clase derivada de *Canvas* que borra los gráficos dibujados en su área de trabajo.

```
void btnBorrar_actionPerformed(ActionEvent e) {
    canvas.repaint();
}
```

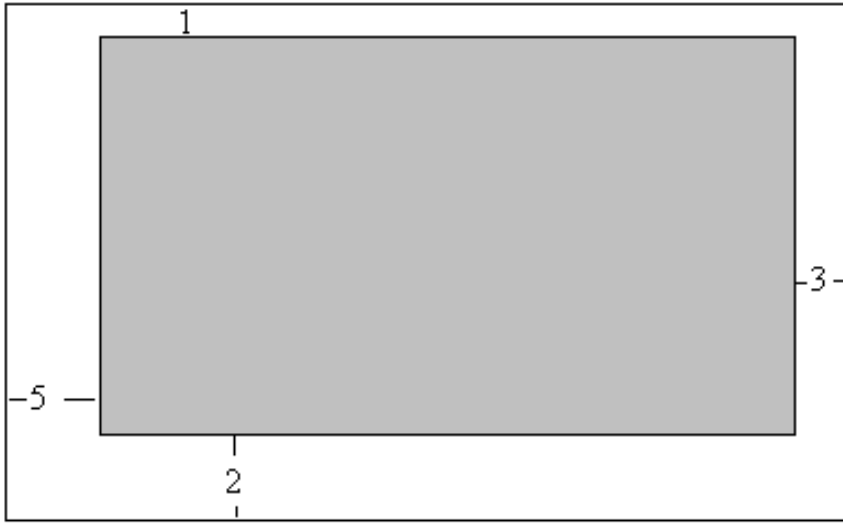
El área de la representación gráfica

Para representar gráficamente una función hemos de establecer un origen y unos ejes. Primero, obtenemos la anchura y la altura del *canvas* mediante *getSize*

```
anchoCanvas=getSize().width;
altoCanvas=getSize().height;
```

El área y los márgenes de la representación gráfica

A continuación determinamos el área del canvas en el que se va a representar la gráfica. No es conveniente aprovechar toda la superficie del canvas, sino dejar un margen alrededor, y espacio suplementario suficiente a la izquierda y en la parte inferior para poner etiquetas a las divisiones de los ejes.



Para situar el origen tomamos como referencia en vez del pixel, la medida de la anchura y la [altura de un carácter](#) de la fuente de texto por defecto.

```
charAlto=g.getFontMetrics().getHeight();
charAncho=g.getFontMetrics().stringWidth("0");
```

Situamos el origen en la parte inferior izquierda del canvas, a 5 caracteres hacia la derecha y dos hacia arriba

```
orgX=5*charAncho;
orgY=altoCanvas-2*charAlto;
```

Dejamos un margen en la parte superior, y en la parte derecha.

Las escalas

La determinación de las escalas es lo que más tiempo lleva si queremos que las distintas representaciones gráficas se vean con claridad. Habitualmente, la determinación de una escala es el resultado de muchas pruebas.

En el eje horizontal vamos a representar las velocidades entre 0 y 3000 m/s, y en el eje vertical representamos la distribución entre 0 y 20 tomado arbitrariamente un valor de N (número de partículas) igual a 10000.

La determinación de una escala es una simple regla de tres, a la anchura del área de representación gráfica (véase la figura) $\text{anchoCanvas} - \text{origenX} - \text{margenDerecho}$ le corresponde 3000. Luego la escala horizontal es

```
escalaX=(double)(anchoCanvas-orgX-3*charAncho)/3000;
```

A la altura del área de la representación gráfica $\text{altoCanvas} - \text{margenSuperior} - \text{margenInferior}$ le corresponde 20. Luego, la escala vertical es

```
escalaY=(double)(altoCanvas-3*charAlto)/20;
```

Una vez determinados el origen y las escalas, podemos proyectar cualquier punto (x, y) del espacio real en un punto del área de la representación gráfica (xI, yI) . Otros sistemas como Windows disponen de funciones que realizan una proyección (mapping) entre estos dos sistemas de coordendas.

```
xI=orgX+(int)(x*escalaX);
yI=orgY-(int)(y*escalaY);
```

Recuérdese, que los argumentos de las funciones gráficas son enteros, por lo que es preciso realizar una conversión (casting) de **double** a **int**.

Los ejes y las divisiones

Una vez establecido el origen y las escalas, dibujamos los ejes y les ponemos etiquetas

```
//eje horizontal
g.drawLine(orgX-charAncho, orgY, anchoCanvas, orgY);
g.drawString("v(m/s)", anchoCanvas-4*charAncho, orgY);
//eje vertical
g.drawLine(orgX, 0, orgX, altoCanvas-charAlto);
g.drawString("dn/dv", orgX+charAncho, charAlto);
```

A continuación, ponemos divisiones al eje horizontal. Tres divisiones grandes en las posiciones que corresponden a 1000, 2000 y 3000 m/s. Subdividimos cada una de ellas en cinco, de modo que tenemos divisiones pequeñas a 200, 400, 600, 800; 1200, 1400, 1600, 1800 m/s, etc. Para mayor claridad ponemos solamente etiquetas a las divisiones grandes. Dichas etiquetas se centran horizontalmente con la correspondiente división mayor. En letra negrita señalamos la parte del código que centra horizontalmente una etiqueta en una división mayor, cuya posición es xI .

```
for(int i=0; i<=3; i++){
    xI=orgX+(int)(1000*i*escalaX);
    g.drawLine(xI, orgY+charAncho/2, xI, orgY-charAncho/2);
    String str=String.valueOf(i*1000);
    g.drawString(str, xI-g.getFontMetrics().stringWidth(str)/2,
                  orgY+charAlto);
    if(i==3) break;
    for(int j=1; j<5; j++){
        xI=orgX+(int)((1000*i+(double)(1000*j)/5)*escalaX);
        g.drawLine(xI, orgY+charAncho/4, xI, orgY-charAncho/4);
    }
}
```

La división grande tiene una longitud igual a la altura de un carácter y la división pequeña tiene una longitud igual a mitad de dicha altura

Hacemos lo mismo con el eje vertical. Ponemos divisiones grandes en 5, 10, 15 y 20, y subdividimos cada intervalo en 5. De modo, que tenemos divisiones pequeñas a 1, 2, 3, 4, 6, 8, 9, ...Se ponen etiquetas en las divisiones grandes y se centran verticalmente con la correspondiente división mayor. En letra negrita marcamos la parte del código que centra verticalmente una etiqueta en una división mayor.

```

for(int i=0; i<=20; i+=5){
    y1=orgY-(int)(i*escalaY);
    g.drawLine(orgX+charAncho/2, y1, orgX-charAncho/2, y1);
    String str=String.valueOf(i);
    g.drawString(str, orgX-g.getFontMetrics().stringWidth(str)-
        charAncho/2, y1+charAlto/2-descent);
    if(i==20) break;
    for(int j=1; j<5; j++){
        y1=orgY-(int)((i+(double)(j))*escalaY);
        g.drawLine(orgX+charAncho/4, y1, orgX-charAncho/4, y1);
    }
}

```

Como se observa en el código, se emplea funciones de [la clase *FontMetrics*](#) para obtener la altura de un carácter, la anchura de una cadena de caracteres, y el parámetro *descent* de la fuente de texto actual. La porción de código que realiza esta tarea es la siguiente.

```

FontMetrics fm=g.getFontMetrics();
int charAlto=fm.getHeight();
int charAncho=fm.stringWidth("0");
int descent=fm.getDescent();

```

Definición de la función

Tenemos que trasladar la expresión matemática a código

$$\frac{dn}{dv} = 4 \pi N \left(\frac{m}{2 \pi k T} \right)^{\frac{3}{2}} v^2 \exp \left(-\frac{mv^2}{2kT} \right)$$

Proporcionamos la masa en unidades de masa atómica de cada uno de los gases que aparecen en el control selección

| Gas | masa (u.m.a) |
|-----------------------------|--------------|
| Hidrógeno (H ₂) | 2 |
| Oxígeno (O ₂) | 32 |
| Nitrógeno (N ₂) | 28 |
| Helio (He) | 4 |
| Neon (Ne) | 10 |
| Argon (Ar) | 18 |

Hallamos el cociente $m/2kT$ y lo guardamos en una variable denominada *cociente*. Una unidad de masa atómica vale $1.672 \cdot 10^{-27}$ kg y la constante de Boltzmann vale $1.38 \cdot 10^{-23}$ J/K.

```

cociente=(1.67e-27*masa[indice])/(2*temperatura*1.38e-23);

```

Definimos la función a representar. Para cada valor de v , devuelve la proporción de partículas de un gas ideal cuyas velocidades están comprendidas entre v y $v+dv$. Dentro de la definición de la función usamos las funciones de [la clase Math](#): *Math.pow* para hallar la potencia de exponente $3/2$ y *Math.exp* para hallar la potencia del número e .

```
double f(double v){
    double y=4*N*Math.PI*Math.pow(cociente/Math.PI, 1.5)*
        Math.exp(-cociente*v*v)*v*v;
    return y;
}
```

Para representar la función simplemente hacemos un bucle **for**, desde el valor v inicial hasta el valor v final, con un determinado paso dv . La elección del paso es importante para que la función se vea como continua. Como la resolución de la ventana es limitada por las dimensiones vertical y horizontal del área de la representación gráfica, la elección de un paso muy pequeño, no tendría efecto visual, ya que los puntos muy próximos se superpondrían sobre el mismo pixel. La elección de un paso grande dará lugar a que la curva continua se transforme en una polilínea u unión de un conjunto de segmentos rectilíneos.

La representación gráfica de una curva se realiza en dos pasos. En primer lugar, se determina las coordenadas del punto inicial. Se calcula el nuevo punto, se une mediante una recta el punto inicial y el actual, a continuación, el punto actual se convierte en el punto inicial para el siguiente paso.

```
//punto inicial
    int x1=orgX, y1=orgY, x2, y2;
    g.setColor(color[nGrafica]);

    for(double v=0; v<3000; v+=10){
//punto actual
        y2=orgY-(int)(funcion(v)*escalaY);
        x2=orgX+(int)(v*escalaX);
        g.drawLine(x1, y1, x2, y2);
//el punto actual se convierte en inicial
        x1=x2; y1=y2;
    }
```

La clase que describe el canvas

Todo lo que se refiere a la representación gráfica lo describimos mediante la clase *MiCanvas* derivada de [Canvas](#). Creamos un *canvas*, u objeto de la clase *MiCanvas* y la ponemos por encima del panel que contiene los controles.

```
this.add(panell, BorderLayout.SOUTH);
this.add(canvas, BorderLayout.CENTER);
```

Cuando se pulsa el botón titulado Gráfica, se llama a la función *setNuevo*, y le pasa el índice del gas ideal que ha sido seleccionado y el valor de la temperatura que se ha introducido en el control de edición. En dicha función, se calcula el cociente $m/2kT$ y se guarda en una variable denominada *cociente*, que se empleará en la definición de la función *f*. Esta función miembro, se encarga también de determinar el índice del color *nGrafica* con el que se dibujará la gráfica. Finalmente, llama a la función que dibuja la gráfica *dibujaFuncion*.

```
void setNuevo(double temperatura, int indice){
```

```

cociente=(1.67e-27*masa[indice])/(2*temperatura*1.38e-23);
nGrafica++;
if(nGrafica>13){
    nGrafica=0;
}
dibujaFuncion();
}

```

Ahora tenemos una disyuntiva, si queremos dibujar una función cada vez, ponemos el código que la dibuja en la redefinición de *paint*, y en *setNuevo* hacemos una llamada a *paint* mediante *repaint*. Véase los ejemplos de la primera página que estudia el [control Canvas](#).

Nuestro propósito es el de dibujar varias curvas y poder comparar las distintas representaciones gráficas. Por ejemplo, la distribución de velocidades de las moléculas de un gas a distintas temperaturas, o bien, la distribución de las velocidades de las moléculas de varios gases ideales a la misma temperatura. Para realizar esta tarea, emplearemos la aproximación que se sugiere en la segunda página dedicada al estudio del [control Canvas](#).

La función *dibujaFuncion* obtiene el [contexto gráfico](#) *g* del componente mediante *getGraphics* y dibuja la función *f* sobre dicho contexto, en el color dado por el índice *nGrafica* del array *color*. No debemos de olvidarnos de liberar los recursos asociados al contexto gráfico *g* mediante *dispose*, cuando se obtiene con la función *getGraphics*. No es necesario hacerlo cuando el contexto *g*, nos lo suministra la función *paint* o *update*, en su único parámetro.

```

void dibujaFuncion(){
    Graphics g=getGraphics();
    int x1=orgX, y1=orgY, x2, y2;
    g.setColor(color[nGrafica]);

    for(double v=0; v<3000; v+=10){
        y2=orgY-(int)(f(v)*escalaY);
        x2=orgX+(int)(v*escalaX);
        g.drawLine(x1, y1, x2, y2);
        x1=x2; y1=y2;
    }
    g.dispose();
}

```

Redefinimos la función *paint*, para dibujar los ejes. La función *paint* se llama cuando se crea el applet o cuando se redimensiona, o bien, indirectamente a través de *repaint*. Cuando aparece el applet se crea el objeto *canvas*, y se llama a su función miembro *paint*, se determina el origen y se dibujan los ejes, preparando el applet para que el usuario seleccione un gas, introduzca un valor para la temperatura y pulse el botón titulado Gráfica.

```

public void paint(Graphics g){
    origen(g);
    dibujaEjes(g);
}

```

Cuando se han acumulado varias gráficas y no se distingue bien entre ellas se pulsa el botón titulado Borrar, que hace una llamada a *paint*, para preparar al applet para dibujar nuevas funciones con nuevos datos.

```

void btnBorrar_actionPerformed(ActionEvent e) {
    canvas.repaint();
}

```

El código completo de la clase *MiCanvas*, es el siguiente

```
package grafica;

import java.awt.*;

public class MiCanvas extends Canvas {
//anchura y altura del canvas
    int anchoCanvas, altoCanvas;
//origenes
    int orgY, orgX;
//escalas
    double escalaX, escalaY;
//masas de las moléculas en u.m.a
    final int masa[]={2, 32, 28, 4, 10, 18};
    double cociente;
//número de partículas
    final int N=10000;
//colores de las funciones
    int nGrafica=-1;
    static final Color color[]={new Color(255,0,0), new Color(0, 255,0), new
Color(0, 0, 255),
        new Color(0, 255, 255), new Color(255, 0, 255), new Color(255, 255, 0),
        new Color(128, 0, 0), new Color(255, 128, 0), new Color(0, 64, 64),
        new Color(128, 128, 255), new Color(128, 0, 64), new Color(255, 0, 128),
        new Color(192, 192, 192), new Color(128, 128, 0)};

    public MiCanvas() {
        setBackground(Color.white);
    }
    void setNuevo(double temperatura, int indice){
        cociente=(1.67e-27*masa[indice])/(2*temperatura*1.38e-23);
        nGrafica++;
        if(nGrafica>13){
            nGrafica=0;
        }
        dibujaFuncion();
    }

    void origen(Graphics g){
        anchoCanvas=getSize().width;
        altoCanvas=getSize().height;
        FontMetrics fm=g.getFontMetrics();
        int charAlto=fm.getHeight();
        int charAncho=fm.stringWidth("0");
//orígenes
        orgX=5*charAncho;
        orgY=altoCanvas-2*charAlto;
//escalas
        escalaX=(double)(anchoCanvas-orgX-3*charAncho)/3000;
        escalaY=(double)(altoCanvas-3*charAlto)/20;
    }
}
```

```

void dibujaEjes(Graphics g){
    FontMetrics fm=g.getFontMetrics();
    int descent=fm.getDescent();
    int charAlto=fm.getHeight();
    int charAncho=fm.stringWidth("0");
//borra el canvas
    g.setColor(getBackground());
    g.fillRect(0,0, anchoCanvas, altoCanvas);
    g.setColor(Color.black);
//eje horizontal
    g.drawLine(orgX-charAncho, orgY, anchoCanvas, orgY);
    g.drawString("v(m/s)", anchoCanvas-4*charAncho, orgY);
    int x1, y1;
    for(int i=0; i<=3; i++){
        x1=orgX+(int)(1000*i*escalaX);
        g.drawLine(x1, orgY+charAncho/2, x1, orgY-charAncho/2);
        String str=String.valueOf(i*1000);
        g.drawString(str, x1-fm.stringWidth(str)/2, orgY+charAlto);
        if(i==3) break;
        for(int j=1; j<5; j++){
            x1=orgX+(int)((1000*i+(double)(1000*j)/5)*escalaX);
            g.drawLine(x1, orgY+charAncho/4, x1, orgY-charAncho/4);
        }
    }

//eje vertical
    g.drawLine(orgX, 0, orgX, altoCanvas-charAlto);
    g.drawString("dn/dv", orgX+charAncho, charAlto);
    for(int i=0; i<=20; i+=5){
        y1=orgY-(int)(i*escalaY);
        g.drawLine(orgX+charAncho/2, y1, orgX-charAncho/2, y1);
        String str=String.valueOf(i);
        g.drawString(str, orgX-fm.stringWidth(str)-charAncho/2, y1+charAlto/2-
descent);
        if(i==20) break;
        for(int j=1; j<5; j++){
            y1=orgY-(int)((i+(double)(j))*escalaY);
            g.drawLine(orgX+charAncho/4, y1, orgX-charAncho/4, y1);
        }
    }
}

double f(double v){
    double y=4*N*Math.PI*Math.pow(cociente/Math.PI, 1.5)*Math.exp(-cociente*v*v)*v*v;
    return y;
}

void dibujaFuncion(){
    Graphics g=getGraphics();
    int x1=orgX, y1=orgY, x2, y2;
    g.setColor(color[nGrafica]);

    for(double v=0; v<3000; v+=10){
        y2=orgY-(int)(f(v)*escalaY);

```

```
        x2=orgX+(int)(v*escalaX);
        g.drawLine(x1, y1, x2, y2);
        x1=x2; y1=y2;
    }
    g.dispose();
}

public void paint(Graphics g){
    origen(g);
    dibujaEjes(g);
}
}
```

El código fuente



grafica: [GraficaApplet.java](#), [MiCanvas.java](#)

Un programa de dibujo simple



[Ejemplos completos](#)

[Diseño del applet](#)

[Manejo del ratón](#)

[Guardar las figuras en memoria](#)

[La clase que describe las figuras](#)

[La jerarquía de clases](#)

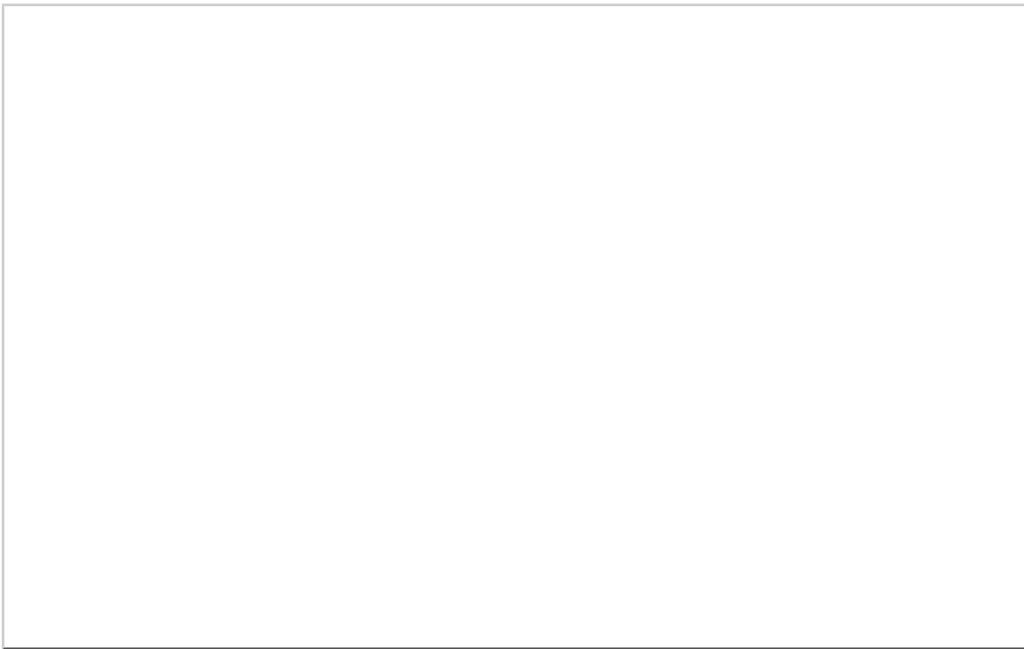
[Conclusiones](#)

[El código fuente](#)

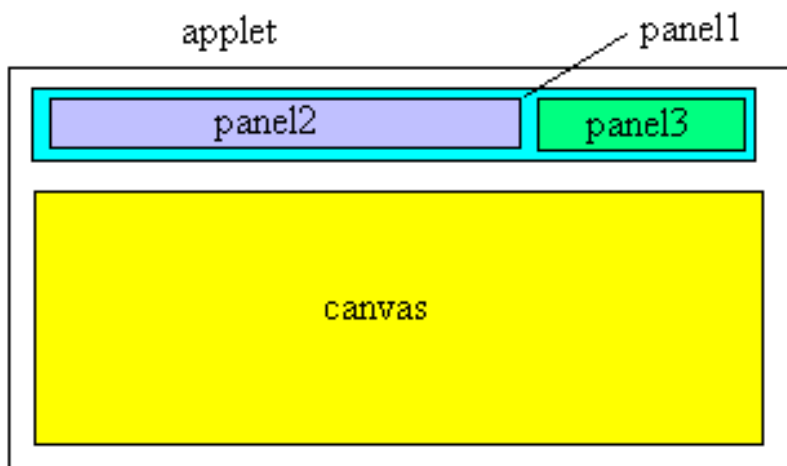
Vamos a estudiar un programa de dibujo muy sencillo que consta de una barra de herramientas constituida por tres botones de radio que corresponden a una línea, una elipse, un rectángulo, y un botón que borra el área de trabajo del canvas. El propósito del programa es el de simular un programa de dibujo con una caja de herramientas.

Otra de las particularidades del programa, es la creación de clases para realizar las distintas tareas. La clase que describe el applet, en la que se disponen los controles, la clase que describe el canvas que nos permite dibujar figuras sobre su área de trabajo, y la clase que describe las distintas figuras.

Podemos finalmente, optar por una clase que describe todas las figuras o bien crear una jerarquía de clases, que descendan de una clase base abstracta, y definir en las clases derivadas la función que la dibuja en un contexto gráfico.



Diseño del applet



El applet consta de paneles anidados en la parte superior en el que se sitúan los botones, y un control canvas en la parte inferior, y que ocupa la mayor parte del applet.

```
public void init() {  
    //...  
    this.add(panel1, BorderLayout.NORTH);  
    this.add(canvas, BorderLayout.CENTER);  
}
```

Una vez situados los controles sobre el applet definimos la respuesta a las acciones del usuario sobre los botones. Para ello vamos a seguir la aproximación que tomamos en el estudio a las [acciones sobre un grupo de botones de radio](#), en la que un único objeto de una clase que implementa el interface *ItemListener* maneja los sucesos que se originan al pulsar sobre dichos botones.

```

class ItemRadio implements ItemListener{
    private MiCanvas canvas;
    public ItemRadio(MiCanvas canvas){
        this.canvas=canvas;
    }
    public void itemStateChanged(ItemEvent ev){
        String s=(String)ev.getItem();
        if(s.equals("Línea")){
            canvas.setFigura(Figura.LINEA);
        }else if(s.equals("Rectángulo")){
            canvas.setFigura(Figura.RECTANGULO);
        }else if(s.equals("Elipse")){
            canvas.setFigura(Figura.ELIPSE);
        }
    }
}

```

La clase *ItemRadio* implementa el interface *ItemListener* y define la función *itemStateChanged*. Esta función necesita acceder al objeto *canvas* de la clase *MiCanvas*, por lo que se lo pasamos en el constructor de la clase. Mediante *getItem* extraemos del objeto *ev* de la clase *ItemEvent* el título del botón de radio que ha sido pulsado, véase el apartado [Información acerca de los sucesos](#).

Creamos un objeto *item* de la clase *ItemRadio* y asociamos mediante *addItemListener* cada uno de los botones de radio con el objeto que registra las acciones sobre dichos controles.

```

ItemRadio item=new ItemRadio(canvas);
chkLinea.addItemListener(item);
chkElipse.addItemListener(item);
chkRectangulo.addItemListener(item);

```

Hacemos doble-clic sobre el botón titulado Borrar en modo diseño, JBuilder genera el código que asocia dicho botón con un objeto de una [clase anónima](#) que implementa el interface *ActionListener*. El programador solamente se tiene que preocupar de añadir el código a la función respuesta *btnBorrar_actionPerformed* cuyo nombre ha generado el Entorno Integrado.

```

void btnBorrar_actionPerformed(ActionEvent ev){
    canvas.borrar();
}

```

Manejo del ratón

Para crear un control canvas definimos una [clase derivada de *Canvas*](#). Dicha clase la llamaremos *MiCanvas*, que redefinirá la función *paint*, y definirá otras funciones miembro.

La clase que describe el canvas ha de responder a las acciones de pulsar el botón izquierdo del ratón, cuando se comienza a dibujar la figura en una determinada posición; de arrastrar el ratón, cuando se proporciona un tamaño a la figura; y de liberar el botón izquierdo del ratón, cuando terminamos de dibujar la figura.

Hemos estudiado las distintas aproximaciones, para responder a estas acciones. Una de ellas, es que la clase que describe el [canvas implemente los interfaces *MouseListener* y *MouseMotionListener*](#). Otra opción, es la de usar el [código generado por JBuider](#) cuando en el modo diseño se selecciona el *canvas*, y en el correspondiente panel Events se hace doble-clic sobre los editores asociados a *mousePressed*, *mouseReleased* y *mouseDragged*.

Vamos a seguir una tercera opción, que es la de crear dos clases internas denominas *RatonPoner* y *RatonMover* que derivan de *MouseAdapter* y de *MouseMotionAdapter*, y que redefinen las funciones *mousePressed*, *mouseReleased* y *mouseDragged*, respectivamente.

```
class RatonPoner extends MouseAdapter{
    public void mousePressed(MouseEvent ev){
        empiezaDibujar(ev.getPoint());
    }
    public void mouseReleased(MouseEvent ev){
        terminaDibujar();
        figuras.addElement(figActual);
    }
}

class RatonMover extends MouseMotionAdapter{
    public void mouseDragged(MouseEvent ev){
        dibuja(ev.getPoint());
    }
}
```

En el constructor de *MiCanvas* se asocia el productor de los sucesos asociados al ratón, el canvas (**this**) con objetos de las clases *RatonPoner* y *RatonMover*.

```
public MiCanvas(int tipo) {
    addMouseListener(new RatonPoner());
    addMouseMotionListener(new RatonMover());
}
```

```
}
```

Para dibujar una figura se requieren tres acciones

- Pulsar el botón izquierdo del ratón
- Arrastrar el ratón con el botón izquierdo pulsado
- Dejar de pulsar el botón izquierdo del ratón

Cuando se pulsa el botón izquierdo del ratón, se llama a la función *mousePressed*, y esta llama a la función *empiezaDibujar* miembro de *MiCanvas*. En esta función se obtiene el contexto gráfico del componente *gc* mediante *getGraphics* y se establece en dicho contexto el modo de dibujo XOR mediante *setXORMode*. Finalmente, se dibuja el punto inicial llamando a la función *dibujar* miembro de la clase *Figura*, para dibujar la figura actualmente seleccionada.

```
void empiezaDibujar(Point p){
    ini=p;
    fin=p;
    gc=getGraphics();
    gc.setXORMode(getBackground());
    figActual.dibujar(gc, ini, fin);
}
```

Cuando se arrastra el ratón permaneciendo el botón izquierdo pulsado se llama a *mouseDragged*, y esta llama a la función miembro *dibuja* de *MiCanvas*, para dibujar la figura con unas dimensiones determinadas. En el modo XOR, la primera sentencia borra la figura dibujada previamente y a continuación dibuja la nueva figura con las dimensiones proporcionadas por el parámetro *p* (punto final).

Los miembros dato *ini* y *fin* (objetos de la clase *Point*), guardan los puntos inicial (esquina superior izquierda) y final (esquina inferior derecha) de la figura que se ha dibujado previamente. El valor de *ini* no cambia y se establece al pulsar el botón izquierdo del ratón, mientras que el valor *fin*, se actualiza con el valor que nos suministra el objeto *ev* de la clase *MouseEvent*. Cuando [se mueve el ratón](#), se llama a la función respuesta *mouseDragged* y ésta llama a *dibuja*.

```
void dibuja(Point p){
    figActual.dibujar(gc, ini, fin);           //borra la figura previa
    fin=p;
    figActual.dibujar(gc, ini, fin);           //dibuja la figura actual
}
```

Cuando se libera el botón izquierdo del ratón, se llama a la función *mouseReleased*, y ésta llama a la función miembro *terminaDibujar* de *MiCanvas*. Dicha función, borra la última figura dibujada, establece el modo de dibujo por defecto con *setPaintMode*, y dibuja la figura final en este modo. Finalmente, nos acordamos de liberar los [recursos asociados al contexto gráfico](#) *gc*, mediante la llamada a *dispose*.

```

void terminaDibujar(){
    figActual.dibujar(gc, ini, fin);
    gc.setPaintMode();
    figActual.dibujar(gc, ini, fin);
    gc.dispose();
}

```

Guardar las figuras en memoria

Cuando se pulsa un botón de radio cuyo nombre es el de una figura se llama a la función respuesta *itemStateChanged* de la clase *ItemRadio*. Esta función llama a *setFigura* miembro de *MiCanvas*, y le pasa el tipo de figura, un valor constante para cada tipo de figura definido en la clase *Figura* que veremos más adelante.

```

public void itemStateChanged(ItemEvent ev){
    String s=(String)ev.getItem();
    if(s.equals("Línea")){
        canvas.setFigura(Figura.LINEA);
    }else if(s.equals("Rectángulo")){
        canvas.setFigura(Figura.RECTANGULO);
    }else if(s.equals("Elipse")){
        canvas.setFigura(Figura.ELIPSE);
    }
}

```

En la función miembro *setFigura* de la clase *MiCanvas* se crea un objeto *figActual* de la clase *Figura* llamando a uno de sus constructores.

```

public void setFigura(int tipo){
    this.tipo=tipo;
    figActual=new Figura(tipo);
}

```

Para guardar un número indeterminado de figuras empleamos [la clase Vector](#). Creamos un objeto *figuras* de dicha clase llamando a uno de sus tres constructores. Cuando la figura se termina de dibujar, al soltar el botón izquierdo del ratón, se añade a la colección mediante *addElement*.

```

public void mouseReleased(MouseEvent ev){
    terminaDibujar();
    figuras.addElement(figActual);
}

```

```

    }
}

```

Para dibujar todas las figuras guardadas en el vector *figuras*, se redefine *paint*.

```

public void paint(Graphics g){
    for(int i=0; i<figuras.size(); i++){
        Figura fig=(Figura)figuras.elementAt(i);
        fig.dibujar(g);
    }
}

```

Para acceder a todos los elementos de la colección se puede emplear [un bucle **while**](#) o bien, un bucle **for**, si obtenemos antes el número de elementos que se han guardado en *figuras* mediante la función miembro *size* de la clase *Vector*.

Una vez obtenida un objeto de la clase *Figura* mediante *elementAt*, se dibuja la figura en el contexto gráfico *g*, llamando a la función miembro *dibujar* de la clase *Figura*.

Cuando se pulsa al botón titulado Borrar se llama a la función *borrar*, se eliminan todas las figuras guardadas en el vector *figuras* mediante *removeAllElements*. Se vuelve a dibujar el canvas llamando a la función miembro *paint*, que no dibuja nada.

```

public void borrar(){
    figuras.removeAllElements();
    repaint();
}

```

El código completo de la clase *MiCanvas* que describe el canvas es, el siguiente

```

public class MiCanvas extends Canvas {
    private int tipo=Figura.LINEA;
    private Vector figuras=new Vector();
    private Figura figActual;
    private Graphics gc;
    private Point ini, fin;
    public MiCanvas(int tipo) {
        this.tipo=tipo;
        addMouseListener(new RatonPoner());
        addMouseMotionListener(new RatonMover());
        figActual=new Figura();
    }
    public void setFigura(int tipo){

```

```

        this.tipo=tipo;
        figActual=new Figura(tipo);
    }
    public void borrar(){
        figuras.removeAllElements();
        repaint();
    }
    private void empiezaDibujar(Point p){
        ini=p;
        fin=p;
        gc=getGraphics();
        gc.setXORMode(getBackground());
        figActual.dibujar(gc, ini, fin);
    }
    private void dibuja(Point p){
        figActual.dibujar(gc, ini, fin);
        fin=p;
        figActual.dibujar(gc, ini, fin);
    }
    private void terminaDibujar(){
        figActual.dibujar(gc, ini, fin);
        gc.setPaintMode();
        figActual.dibujar(gc, ini, fin);
        gc.dispose();
    }
    public void paint(Graphics g){
        for(int i=0; i<figuras.size(); i++){
            Figura fig=(Figura)figuras.elementAt(i);
            fig.dibujar(g);
        }
    }
    //...
}

```

La clase que describe las figuras

La clase *Figura* tiene tres miembros datos constantes que indican el tipo de figura. Se [accede a dichos miembros estáticos](#) poniendo el nombre de la clase *Figura*, seguido del nombre del miembro estático, separados por un punto


```
canvas.setFigura(Figura.LINEA);
```

La clase *Figura* tiene tres constructores, el constructor por defecto, y dos constructores explícitos. También tiene dos funciones miembro con el mismo nombre *dibujar*. La primera versión, se llama mientras se está dibujando la figura, y la segunda versión se llama cuando se dibujan todas las figuras en la redefinición de *paint*.

La primera versión de *dibujar*, actualiza los valores de los miembros dato *ini* y *fin*, que guardan los puntos extremo superior izquierdo de la figura, y extremo inferior derecho de la misma, después llama a la función correspondiente de la clase *Graphics* para dibujar el tipo de figura que se guarda en el miembro dato *tipo*. Para dibujar una línea se llama a *drawLine*, para dibujar un rectángulo a *drawRect*, y para dibujar una elipse a *drawOval*.

Se ha de tener cuidado al dibujar un rectángulo o una elipse, ya que los dos primeros parámetros indican las coordenadas de la esquina superior izquierda y los dos últimos la anchura y la altura, ambas cantidades positivas. Pero con el ratón podemos dibujar en la dirección que queramos, de modo que la anchura o altura podrían ser cantidades negativas, y la esquina superior izquierda podría ser la inferior derecha. Tenemos que escribir el código que trate esta situación empleando las funciones estáticas [*min* y *abs* miembros de la clase *Math*](#).

La segunda versión de *dibujar*, se limita a llamar a la primera versión, pasándole los valores que guardan los miembros dato *ini* y *fin*, cuando se ha terminado de dibujar la figura.

El código completo de la clase *Figura* es, el siguiente

```
public class Figura {
    public static final int LINEA=0;
    public static final int RECTANGULO=1;
    public static final int ELIPSE=2;
    private int tipo;
    private Point ini, fin;
//figura final
    public Figura(int tipo, Point ini, Point fin) {
        this.tipo=tipo;
        this.ini=ini;
        this.fin=fin;
    }
    public Figura(){
        this(LINEA, new Point(), new Point());
    }
    public Figura(int tipo){
        this(tipo, new Point(), new Point());
    }
}
```

```
//mientras se dibuja la figura
public void dibujar(Graphics g, Point ini, Point fin){
    this.ini=ini;
    this.fin=fin;
    if(tipo==LINEA){
        g.drawLine(ini.x, ini.y, fin.x, fin.y);
    }else{
        int ancho=Math.abs(ini.x-fin.x);
        int alto=Math.abs(ini.y-fin.y);
        int iniX=Math.min(ini.x, fin.x);
        int iniY=Math.min(ini.y, fin.y);
        if(tipo==ELIPSE){
            g.drawOval(iniX, iniY, ancho, alto);
        }else{
            g.drawRect(iniX, iniY, ancho, alto);
        }
    }
}

//figura final
public void dibujar(Graphics g){
    dibujar(g, ini, fin);
}
}
```

La jerarquía de clases

Ya hemos estudiado la [jeraquía de clases de las figuras planas](#), y se ha definido la función *area* que calcula el área de cada uno de los tipos de figuras. Vamos a hacer algo parecido, pero definiendo en las clases derivadas la función *dibujar* que dibuja una figura concreta en un contexto gráfico.

La clase base *Figura* tiene como miembros dato los puntos *ini* (esquina superior izquierda) y *fin* (esquina inferior derecha) de la figura, y como funciones miembro, *actualizar*, que actualiza los miembros dato *ini* y *fin*, a medida que se va dibujando la figura con el ratón.

```
protected void actualizar(Point ini, Point fin){
    this.ini=ini;
    this.fin=fin;
}
```

La función miembro *dibujar*, dibuja la figura final que se ha guardado en el vector *figuras*, y que se llama en

paint cuando se dibujan todas las figuras en el contexto gráfico *g*.

```
public void dibujar(Graphics g){
    dibujar(g, ini, fin);
}
```

Se declara abstracta la primera versión de *dibujar*, que se define en cada una de las clases derivadas, esta función se llama mientras se está dibujando la figura.

```
public abstract void dibujar(Graphics g, Point ini, Point fin);
```

La definición de la clase base *Figura* y de las clases derivadas *Linea*, *Elipse* y *Rectangulo*, es el siguiente

```
public abstract class Figura {
    protected Point ini, fin;
    public Figura(Point ini, Point fin) {
        this.ini=ini;
        this.fin=fin;
    }
    public Figura(){
        this(new Point(), new Point());
    }
    protected void actualizar(Point ini, Point fin){
        this.ini=ini;
        this.fin=fin;
    }
}

//mientras se dibuja la figura
public abstract void dibujar(Graphics g, Point ini, Point fin);

//figura final
public void dibujar(Graphics g){
    dibujar(g, ini, fin);
}

}

class Linea extends Figura{
    public Linea(Point ini, Point fin) {
        super(ini, fin);
    }
    public Linea(){
        this(new Point(), new Point());
    }
    public void dibujar(Graphics g, Point ini, Point fin){
```

```

        actualizar(ini, fin);
        g.drawLine(ini.x, ini.y, fin.x, fin.y);
    }
}

class Rectangulo extends Figura{
    public Rectangulo(Point ini, Point fin) {
        super(ini, fin);
    }
    public Rectangulo(){
        this(new Point(), new Point());
    }
    public void dibujar(Graphics g, Point ini, Point fin){
        actualizar(ini, fin);
        int ancho=Math.abs(ini.x-fin.x);
        int alto=Math.abs(ini.y-fin.y);
        int iniX=Math.min(ini.x, fin.x);
        int iniY=Math.min(ini.y, fin.y);
        g.drawRect(iniX, iniY, ancho, alto);
    }
}

class Elipse extends Figura{
    public Elipse(Point ini, Point fin) {
        super(ini, fin);
    }
    public Elipse(){
        this(new Point(), new Point());
    }
    public void dibujar(Graphics g, Point ini, Point fin){
        actualizar(ini, fin);
        int ancho=Math.abs(ini.x-fin.x);
        int alto=Math.abs(ini.y-fin.y);
        int iniX=Math.min(ini.x, fin.x);
        int iniY=Math.min(ini.y, fin.y);
        g.drawOval(iniX, iniY, ancho, alto);
    }
}

```

La clase *MiCanvas*, no sufre apenas variación salvo en la función miembro *setFigura*. Se crea un objeto de la clase derivada dependiendo del tipo de figura seleccionada al pulsar uno u otro botón. El objeto de la clase derivada se guarda en una variable *figActual* de la clase base *Figura*. Cuando se llama desde *figActual* a la función *dibujar*, ¿a cuál de las tres funciones *dibujar* se llama: la que dibuja una línea, la que dibuja un

rectángulo, o la que dibuja una elipse?. Esto es lo que estudiamos en el capítulo dedicado a la [herencia y el polimorfismo](#).

```
public class MiCanvas extends Canvas{
    private Figura figActual;
    public MiCanvas() {
        figActual=new Linea();
    }
    public void setFigura(int tipo){
        switch(tipo){
            case 0:
                figActual=new Linea();
                break;
            case 1:
                figActual=new Rectangulo();
                break;
            case 2:
                figActual=new Elipse();
                break;
            default:
                figActual=new Linea();
                break;
        }
    }
}
//otras funciones miembro
}
```

Conclusiones

En este ejemplo hemos visto los siguientes temas:

1. Respuesta a las acciones del usuario sobre un grupo de botones de radio, y un botón.
2. Respuesta a los sucesos asociados al ratón
3. Creación de clases para distintos propósitos:
 - la clase que describe el applet
 - la clase que describe un canvas
 - la clase o la jerarquía de clases que describen las figuras
4. En este último caso, podemos crear una jerarquía de clases y aplicar de forma elegante las características de la herencia y del polimorfismo del lenguaje Java.

5. El modo de dibujo XOR
6. Guardar objetos en un vector, u objeto de la clase *Vector*.

El código fuente

Las dos versiones del programa

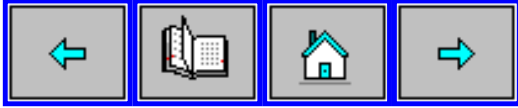


draw: [Figura.java](#), [MiCanvas.java](#), [DrawApplet.java](#)



draw1: [Figura.java](#), [MiCanvas.java](#), [DrawApplet1.java](#)

El movimiento de los planetas



[Ejemplos completos](#)

[Introducción](#)

[Fundamentos físicos y procedimientos numéricos](#)

[Movimiento del planeta](#)

[El estado del móvil](#)

[La comunicación entre el usuario y el programa](#)

[Relación entre los objetos de las distintas clases](#)

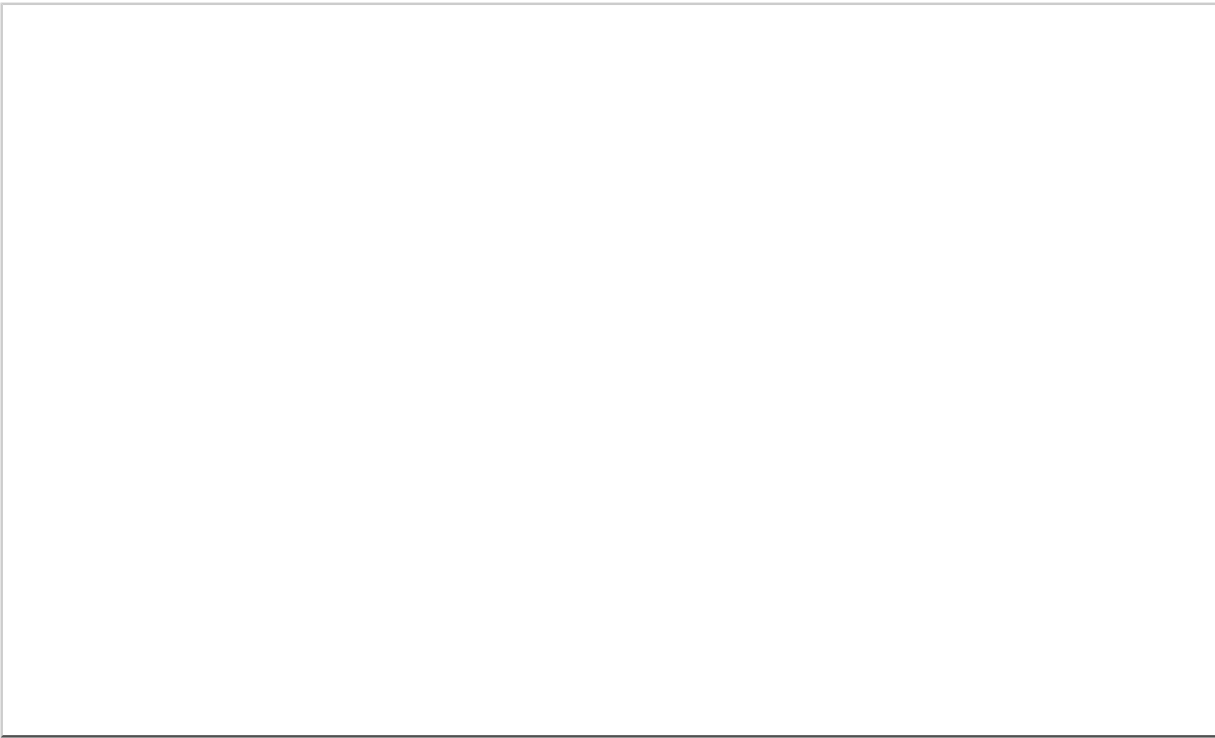
[Conclusiones](#)

[Bibliografía](#)

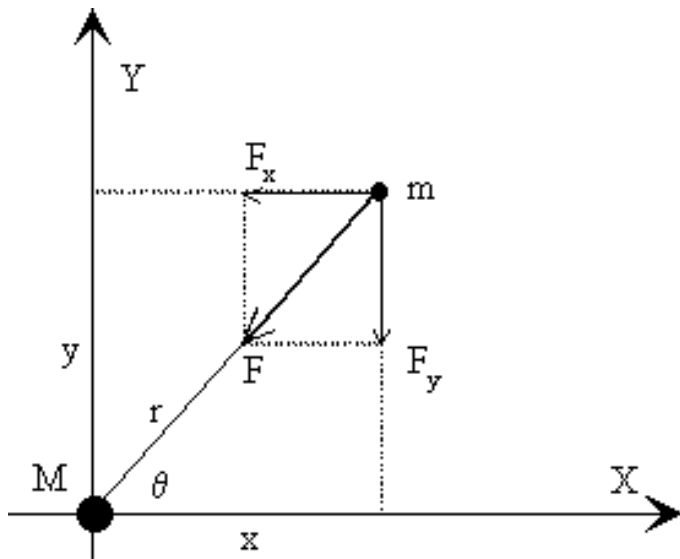
[El código fuente](#)

Introducción

El objetivo del programa interactivo, o applet que se explica en esta página va a consistir en mostrar de forma animada el [movimiento de un planeta](#), pudiendo el usuario detener su movimiento en cualquier instante para tomar datos de su posición y de su velocidad.



Fundamentos físicos y procedimientos numéricos.



Uno de los ejemplos más interesantes de resolución de un sistema de ecuaciones diferenciales de segundo orden es la descripción del movimiento planetario, el cual tiene una solución analítica sencilla en coordenadas polares. La trayectoria seguida por un planeta es una cónica, normalmente una elipse, en uno de cuyos focos está el centro fijo de fuerzas, el Sol.

En la figura, se muestra la fuerza que ejerce el Sol sobre un planeta, inversamente proporcional al cuadrado de las distancias que separan sus centros, y también se muestran sus componentes rectangulares

Teniendo en cuenta que la fuerza que ejerce el Sol sobre un planeta viene descrita por la ley de la Gravitación Universal

$$F = \frac{GMm}{r^2}$$

donde M es la masa del Sol, m la masa del planeta y r la distancia entre el centro del Sol y del planeta. Las componentes de la aceleración del planeta serán

$$a_x = -\frac{F_x}{m} = -\frac{F \cos \theta}{m} = -\frac{GM}{r^3} x$$

$$a_y = -\frac{F_y}{m} = -\frac{F \sin \theta}{m} = -\frac{GM}{r^3} y$$

Escalas

Uno de los problemas del tratamiento numérico con ordenador, es la de reducir el problema a números simples e inteligibles por el usuario de un vistazo. Las masa de los planetas y del Sol son números muy grandes: la masa de la Tierra es $5.98 \cdot 10^{24}$ kg., y $1.98 \cdot 10^{30}$ kg. la del Sol. La distancia media entre la Tierra y el Sol es también muy grande $1.49 \cdot 10^{11}$ m. y la constante G es muy pequeña $6.67 \cdot 10^{-11}$ en el Sistema Internacional de Unidades. Podemos simplificar el problema numérico, refiriéndonos a un hipotético Sol cuya masa sea tal que el producto $GM=1$, o bien que se ha cambiado la escala de los tiempos de modo que se cumpla esa igualdad. Teniendo en cuenta que la aceleración es la derivada segunda de la posición, el movimiento del planeta queda descrito por el siguiente sistema de dos ecuaciones diferenciales de segundo orden

$$\frac{d^2 x}{dt^2} = -\frac{x}{r^3}$$

$$\frac{d^2 y}{dt^2} = -\frac{y}{r^3}$$

Existen numerosos métodos de resolución de problemas en las que las fuerzas son centrales y conservativas, los más sencillos acumulan en cada paso el error inherente a todo procedimiento numérico, haciendo que el planeta describa una espiral en vez de una elipse que es la trayectoria esperada. El procedimiento elegido denominado de Runge-Kutta es estable y relativamente fácil de programar.

Procedimiento numérico de Runge-Kutta

En el Curso de Procedimientos Numéricos en Lenguaje Java, se estudia el procedimiento de Runge-Kutta, para resolver un [sistema de dos ecuaciones diferenciales de segundo orden](#). En este caso, creamos una clase denominada *Planeta* con una función miembro denominada *resolver* que nos proporciona la posición de la partícula cada intervalo de tiempo dt , para poder así dibujar la trayectoria del planeta en el contexto gráfico del canvas

```

public class Planeta {
    private double r;          //distancia al centro de fuerzas
    private double a, b, c, d;
    private double l1, l2, l3, l4;
    private double k1, k2, k3, k4;
    private double m1, m2, m3, m4;
    private double q1, q2, q3, q4;
    private double dt;          //intervalo

    public double t;
    public double x;
    public double y;
    public double Vx;
    public double Vy;

    Planeta(double x, double y, double Vx, double Vy, double t, double dt){
        this.x=x;
        this.y=y;
        this.Vx=Vx;
        this.Vy=Vy;
        this.dt=dt;
        this.t=t;
    }

    public void resolver(){
        //resolución del sistema de ecuaciones diferenciales de segundo orden
        //por el procedimiento de Runge-Kutta
        a=x; b=y; c=Vx; d=Vy;
        l1=c*dt; q1=d*dt;
        r=Math.sqrt(a*a+b*b);
        k1=-a*dt/r/r/r; m1=-b*dt/r/r/r;
        a=x+l1/2; b=y+q1/2; c=Vx+k1/2; d=Vy+m1/2;
        l2=c*dt; q2=d*dt;
        r=Math.sqrt(a*a+b*b);
        k2=-a*dt/r/r/r; m2=-b*dt/r/r/r;
        a=x+l2/2; b=y+q2/2; c=Vx+k2/2; d=Vy+m2/2;
        l3=c*dt; q3=d*dt;
        r=Math.sqrt(a*a+b*b);
        k3=-a*dt/r/r/r; m3=-b*dt/r/r/r;
        a=x+l3; b=y+q3; c=Vx+k3; d=Vy+m3;
        l4=c*dt; q4=d*dt;
        r=Math.sqrt(a*a+b*b);
        k4=-a*dt/r/r/r; m4=-b*dt/r/r/r;
        //valores de las variables en el instante t+dt
        t+=dt;
        x+=(l1+2*l2+2*l3+l4)/6;
        y+=(q1+2*q2+2*q3+q4)/6;
    }
}

```

```

        Vx+=(k1+2*k2+2*k3+k4)/6;
        Vy+=(m1+2*m2+2*m3+m4)/6;
    }
}

```

La clase *Planeta* tiene una función miembro pública denominada *resolver*, que calcula la posición del móvil en el instante $t+dt$ cuando se conoce su posición del móvil en el instante t , es decir, resuelve numéricamente el sistema de dos ecuaciones diferenciales de segundo orden.

Movimiento del planeta

La clase que describe el applet implementa el [interface *Runnable*](#) y define el método *run* tal como se aprecia en el cuadro que viene a continuación.

```

public class PlanetaApplet extends Applet implements Runnable{
//...
    public void run(){
        while(true){
            try{
                hilo.sleep(20);
            }catch(InterruptedException e){}
            if(bMover){
                canvas.mover();
            }
        }
    }
    void btnEmpieza_actionPerformed(ActionEvent e) {
        double x=Double.valueOf(tPosicion.getText()).doubleValue();
        double Vy=Double.valueOf(tVelocidad.getText()).doubleValue();
        btnPausa.setLabel("  Pausa  ");
        canvas.nuevo(x, Vy);
        if(hilo==null){
            hilo=new Thread(this);
            hilo.start();
        }
        bMover=true;
    }
    void btnPausa_actionPerformed(ActionEvent e) {
        if(bMover==true){
            bMover=false;
            btnPausa.setLabel("Continua");
        }else{

```

```

        btnPausa.setLabel( "    Pausa    " );
        bMover=true;
    }
}

```

Cuando se pulsa el botón titulado "Empieza" se crea un subproceso o thread y se establece el estado inicial (posición inicial y velocidad inicial) de la partícula. Cuando se pulsa el botón "Pausa" cambia su título a "Continúa" y cambia de pausa (**false**) a movimiento (**true**) y viceversa.

En la [función respuesta a la pulsación sobre el botón](#) titulado "Empieza", se crea el subproceso *hilo* de la clase *Thread* y se pone en marcha, llamando a *start*. La función miembro *run* tiene un bucle que se ejecuta indefinidamente, y que llama a la función *mover* que mueve al planeta en el área de trabajo del *canvas*. Además, le hemos añadido la característica de que el planeta detenga su movimiento para que el usuario examine el valor de las variables que describen su estado. El miembro *bMover* se encarga de esta tarea. Si *bMover* guarda **true** se mueve el planeta, en caso contrario se salta la sentencia que llama a la función *mover*.

Representación de la trayectoria

La representación de la trayectoria que sigue el planeta se lleva a cabo en una [clase derivada de Canvas](#) denominada *MiCanvas*, en la que se redefine la función miembro *paint*. La función *paint* de la clase base se limita a borrar el área de trabajo del componente, por lo que redefinimos dicha función en la clase derivada para que pinte el origen y los ejes del movimiento.

La función *nuevo* inicializa los miembros dato del componente cada vez que se comienza una nueva "experiencia", creando un objeto *planeta* de la clase *Planeta* y determinando su posición *x1* e *y1* sobre el área de trabajo del *canvas*.

La función miembro *mover* se llama desde la función miembro *run*, obtiene el [contexto gráfico](#) del componente mediante *getGraphics*, se calcula la nueva posición del planeta, y se dibuja en dicho contexto una línea recta entre las dos posiciones consecutivas del móvil. La posición final *x2*, *y2* será la posición inicial *x1* e *y1* en la siguiente llamada a la función *mover*.

La función *mover* se llama muchísimas veces, por lo que no debemos de olvidarnos de llamar desde el contexto gráfico *g* a *dispose*, para liberar los recursos asociados a *g*.

```

void nuevo(double x, double Vy){
    this.x=x;
    this.Vy=Vy;
    t=0.0;
    Vx=0.0;
    y=0.0;
    planeta=new Planeta(x, y, Vx, Vy, t, dt);
    x1=orgX+(int)(x*escala);
    y1=orgY;
    nOrbita++;
    if(nOrbita>13){
        nOrbita=0;
    }
    double mAngular=x*Vy;
    double energia=Vy*Vy/2-2/x;
    parent.estado.setConstantes(energia, mAngular);
}
void mover(){
    int x2, y2;
    Graphics g=getGraphics();
//calcula la nueva posición del planeta
    planeta.resolver();
//muestra los valores de las variables
    parent.estado.setEstado(planeta.x, planeta.y, planeta.Vx, planeta.Vy,
planeta.t);
    g.setColor(color[nOrbita]);
//situa al planeta en la nueva posición
    x2=orgX+(int)(escala*planeta.x);
    y2=orgY-(int)(escala*planeta.y);

    g.drawLine(x1, y1, x2, y2);
    x1=x2; y1=y2;
    g.dispose();
}
public void paint(Graphics g){
    origenEscalas(g);
    dibujaEjes(g);
}
}

```

El estado del móvil

Para conocer los valores de la posición, y velocidad del móvil en cada instante, se pueden mostrar en la parte superior del área de trabajo del *canvas*, convirtiendo los [valores numéricos a un string](#) mediante la función miembro *valueOf* de la clase *String*, y posteriormente dibujando el texto en el contexto gráfico del componente mediante *drawString*. Para mostrar el valor de la abscisa x en el punto de coordenadas 20, 30 del contexto gráfico g escribimos.

```
g.drawString("X: "+String.valueOf(x), 20, 30);
```

También se puede escribir alternativamente

```
g.drawString("X: "+x, 20, 30);
```

ya que se convierte automáticamente el número x , en un string.

No resulta adecuado, mostrar todas las cifras decimales del número x del tipo predefinido **double**, ya que el procedimiento numérico de cálculo no está libre de errores. Basta en principio, con dos números decimales, para ello podemos emplear dos alternativas una de las cuales emplea la [función estática *floor* de la clase *Math*](#).

```
this.x=(double)((int)(x*100))/100;
this.x=Math.floor(x*100)/100;.
```

Dado que la posición y velocidad del móvil cambian con el tiempo, sería preciso mostrar sus valores en una zona del contexto gráfico g del *canvas*, borrar dicha zona y volver a mostrar dichos valores en la nueva posición del móvil. Este proceso da lugar a un parpadeo que es molesto para el usuario. La solución viene dada por la creación de un nuevo componente derivado de la clase *Canvas* que se ha denominado *Estado*.

El *double-buffer*

En dicho componente mostramos los valores cambiantes de la posición y velocidad del móvil empleando una técnica denominada [double-buffer](#) que se emplea en la mayor parte de las animaciones. El concepto de double-buffer es realmente simple, dibujamos en un contexto en memoria y cuando la imagen esté terminada movemos el bloque entero de datos desde la memoria al contexto gráfico del componente en una única y muy rápida operación.

El primer paso, consiste en redefinir el método *update*. El método por defecto borra el componente con el color del fondo y llama a la función *paint*. Podemos cambiar la conducta por defecto, definiendo nuestro propio método *update* que ya no borra el área de trabajo del componente, sino que llama a la función *paint*.

```
public void update(Graphics g){
    paint(g);
}
```

Primero, se crea un área en memoria, cuyas dimensiones son las del componente *d.width* y *d.height*, respectivamente, y se obtiene su contexto gráfico *gBuffer*.

```

    imag=createImage(d.width, d.height);
    gBuffer=imag.getGraphics();

```

Se borra dicha área de memoria con el color de fondo del componente

```

    gBuffer.setColor(getBackground());
    gBuffer.fillRect(0,0, d.width, d.height);

```

Se dibujan en dicho contexto en memoria textos, gráficos o imágenes sin afectar para nada el área de trabajo del componente.

```

    muestraValores(gBuffer);

```

Se transfiere el bloque entero de memoria al contexto gráfico del componente.

```

public void paint(Graphics g){
    //...
    g.drawImage(imag, 0, 0, null);
}

```

Finalmente, todo el proceso se realiza mediante una única llamada a la función *repaint* usualmente desde el método *run* del thread activo. En nuestro caso indirectamente, una vez que se han actualizado los valores de las variables que definen el estado del móvil.

```

void setEstado(double x, double y, ...){
    this.x=Math.floor(x*100)/100;
    //...
    repaint();
}

```

```

void setEstado(double x, double y, double Vx, double Vy, double t){
//dos cifras decimales
    this.x=Math.floor(x*100)/100;
    this.y=Math.floor(y*100)/100;
    this.Vx=Math.floor(Vx*100)/100;
    this.Vy=Math.floor(Vy*100)/100;
    this.t=Math.floor(t*100)/100;
    repaint();
}
void muestraValores(Graphics g){
    int cAlto=g.getFontMetrics().getHeight();
    int cAncho=g.getFontMetrics().stringWidth("0");
    g.setColor(Color.black);
    g.drawString("Tiempo", 0, 6*cAlto);
    g.drawString(String.valueOf(t), cAncho, 7*cAlto);
}

```

```

        g.drawString("Posición", 0, 9*cAlto);
        g.drawString("X: "+x, cAncho, 10*cAlto);

//...
    }
    public Dimension getPreferredSize(){
        return new Dimension(80, 300);
    }
    public void update(Graphics g){
        paint(g);
    }
    public void paint(Graphics g){
        Dimension d=getSize();
        if((gBuffer==null)|| (d.width!=dim.width)|| (d.height!=dim.height)){
            dim=d;
            imag=createImage(d.width, d.height);
            gBuffer=imag.getGraphics();
        }
        gBuffer.setColor(getBackground());
        gBuffer.fillRect(0,0, d.width, d.height);
        muestraValores(gBuffer);
        g.drawImage(imag, 0, 0, null);
    }
}

```

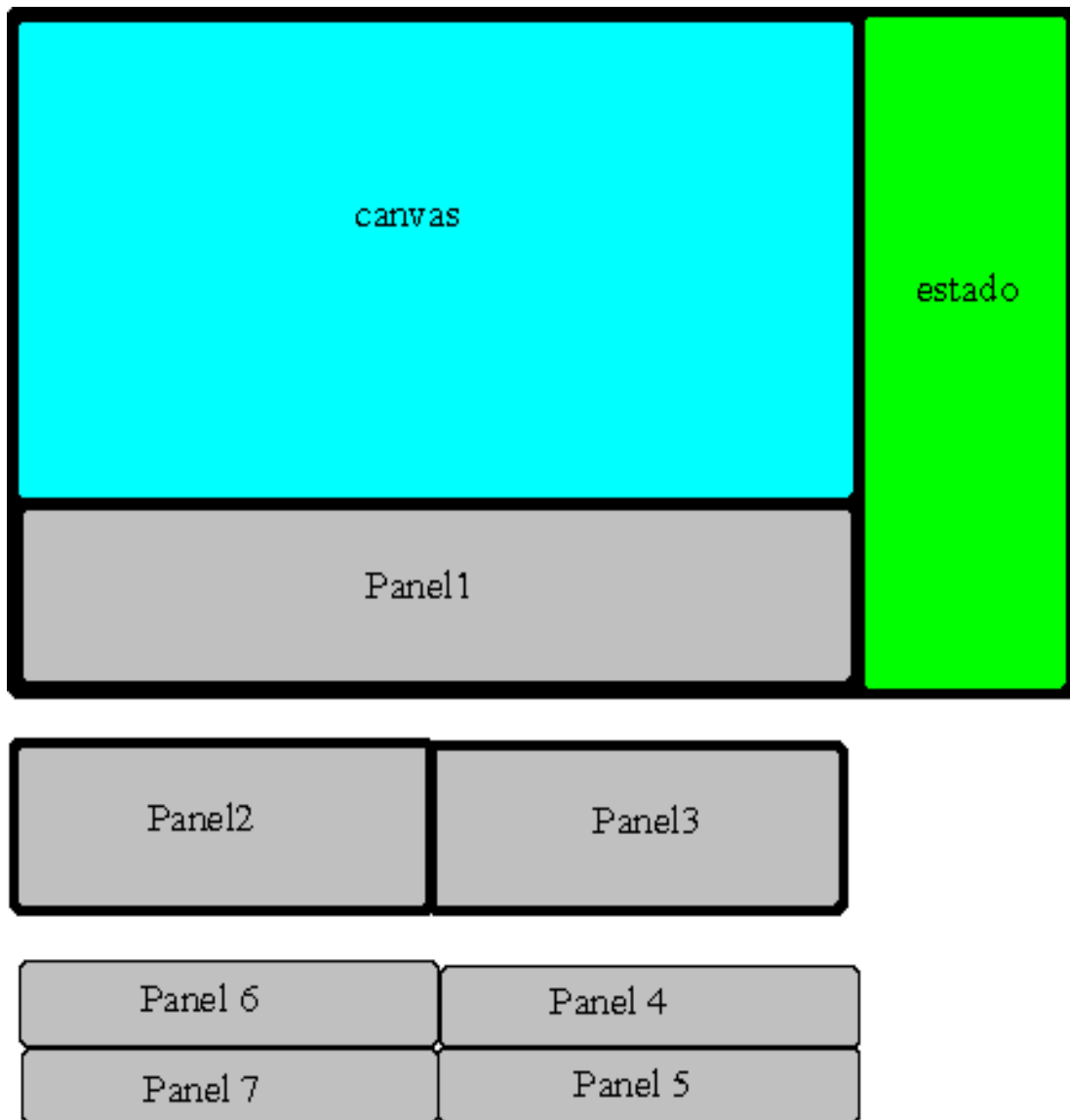
La comunicación entre le usuario y el programa

La comunicación entre el usuario y el programa se realiza mediante la disposición en la superficie del applet de controles y componentes, que permiten introducir los valores iniciales, controlar la evolución del sistema y observar su comportamiento de forma gráfica o animada.

Para introducir los valores iniciales de la posición y velocidad del móvil, se puede hacer con el ratón o bien con controles estándar. En el primer caso, se determina la posición inicial en el lugar en el que se pulsa el botón izquierdo del ratón, y la velocidad se determina por la longitud que se arrastra el ratón manteniendo pulsado el botón izquierdo hasta que se libera.

Mediante controles, se puede optar por controles de edición (*TextField*) o por barras de desplazamiento (*Scrollbar*). En el primer caso, sería necesario [verificar que la entrada es correcta](#), que los caracteres son numéricos y que el valor introducido está en el intervalo adecuado. La segunda alternativa es más segura ya que no es necesario verificar nada, solamente es necesario convertir el desplazamiento del dedo de la barra en valores aceptables haciendo un cambio de escala.

Ahora queda la tarea de disponer los controles y los componentes en la superficie del applet.



En vez de emplear un gestor de diseño complicado como *GridBagLayout*, se ha empleado la aproximación de paneles anidados tal como se muestra en la figura.

Se emplea el gestor [*BorderLayout*](#) para disponer el *canvas* (CENTER) el *estado* (EAST) y el *Panel1* (SOUTH). Sobre el *Panel1* situamos el *Panel2* (CENTER) y el *Panel3* (EAST). Sobre el *Panel2* situamos el *Panel6* (CENTER) y el *Panel7* (SOUTH). Por otra parte, sobre el *Panel3* situamos el *Panel4* (CENTER) y el *Panel5* (SOUTH).

Ahora empleamos el gestor [*FlowLayout*](#) para situar los controles sobre los paneles: en cada uno de los paneles 6 y 7 colocamos una etiqueta (*Label*) y un control de edición (*TextField*), alineados a la izquierda.

En el *Panel4* colocamos los botones titulados "Empieza" y "Pausa" centrados, y finalmente en el *Panel5* colocamos el botón "Borra" con la misma alineación.

Todas estas operaciones se realizan rápidamente con el ratón en el modo de diseño de JBuilder. Los componentes

canvas y *estado* se añaden manualmente en el modo código fuente, y podemos ver el resultado inmediatamente en el modo de diseño, gracias al sistema en el que los cambios en el código se reflejan en el diseño, y cambios en el diseño se reflejan en el código fuente.

Cuando hayamos terminado el diseño, procedemos a definir las [funciones respuesta a las acciones del usuario sobre los botones](#). En el panel de diseño hacemos doble-clic sobre cada uno de los botones. JBuilder genera automáticamente el código acorde con el nuevo modelo denominado *Delegation-based event model* para la gestión de los sucesos (events) generados por el usuario o por el sistema. El programador solamente tiene que definir la función respuesta cuyo nombre ha generado JBuilder.

Cuando se pulsa el botón titulado "Empieza" se leen los controles de edición y se transforma el texto en valor numérico, que se guarda en las variables locales x y Vx . Dichos valores se pasan al *canvas*, y se crea el subproceso denominado *hilo*.

Cuando se actúa sobre el botón titulado "Pausa" se modifica el valor de la variable *bMover*, y se cambia alternativamente el título del botón.

Cuando las trayectorias no se vean con claridad, se pulsa el botón titulado "Borrar" para llamar a la función *paint* desde el objeto *canvas* que borra el área de trabajo de dicho componente y vuelve a dibujar el origen y los ejes.

```
void btnBorrar_actionPerformed(ActionEvent e) {
    canvas.repaint();
}
```

Relación entre los objetos de las distintas clases

Se crean dos objetos *canvas* y el *estado* en la clase *PlanetaApplet1* para disponerlos sobre la superficie del applet.

```
public class PlanetaApplet1 extends Applet {
    MiCanvas canvas;
    Estado estado;

    public void init(){
        canvas=new MiCanvas(this);
        estado=new Estado();
        this.add(estado, BorderLayout.EAST);
        this.add(canvas, BorderLayout.CENTER);
        //...
    }
    //...
}
```

Por otra parte, en la clase *MiCanvas* se creará un objeto *planeta* de la clase *Planeta*, para moverlo por su área de

trabajo.

```
void nuevo(double x, double Vy){
    //...
    planeta=new Planeta(x, y, Vx, Vy, t, dt);
}
```

La clase *MiCanvas* precisa acceder a la función miembro *setEstado* de la clase *Estado* para mostrar en su área de trabajo el estado del planeta en movimiento. Esto se puede hacer a través de la referencia que mantiene del *applet* y que se pasa en el constructor de la clase *MiCanvas*.

```
public class MiCanvas extends Canvas {
    PlanetaApplet parent;

    //...
public MiCanvas(PlanetaApplet p) {
    parent=p;
}
void mover(){
    //...
    parent.estado.setEstado(planeta.x, planeta.y, planeta.Vx, planeta.Vy,
planeta.t);
}
```

Conclusiones

En este estudio se ha puesto de manifiesto varios aspectos importantes en la programación Java:

1. La organización del código, creando clases para las distintas tareas:
La clase *Planeta* que calcular las posiciones sucesivas del planeta.
La clase *MiCanvas* que representa gráficamente la trayectoria.
La clase *Estado* que muestra el valor de las variables que describen la dinámica de dicho cuerpo en movimiento.
La clase *PlanetaApplet* derivada de *Applet* que implementa el interface *Runnable* cuya tarea es la de poner en marcha la aplicación, mover el planeta a intervalos fijos de tiempo, y posibilitar la comunicación entre el usuario y el programa, por medio de los controles que se disponen en su área de trabajo.
2. La disposición de los controles y componentes sobre la superficie del applet, empleando el diseño visual de *JBUILDER*.
3. El uso de subprocesos que posibilitan la programación multitarea.

4. La representación gráfica de la trayectoria de un móvil, y finalmente el uso de la técnica conocida como double-buffer para evitar el molesto parpadeo que se produce al mostrar y borrar información sucesivamente.

Bibliografía

B.P. Demidowitsch, I. A. Maron, E. S. Schuwalowa. *Métodos numéricos de análisis*. Editorial Paraninfo (1980)

Angel Franco García. *Creación de applets educativos: El movimiento de los planetas*. RPP nº 42, Julio-Agosto 1998.

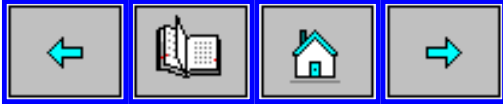
Pedro Agulló Soliveres. *Curso de Java (VII) Programación concurrente en Java (I)*. RPP septiembre de 1997.

David M. Geary. *Graphic JAVA 1.1. Mastering the AWT, second edition*. Sun Microsystems (1997).

El código fuente

 planeta: [PlanetaApplet.java](#), [MiCanvas.java](#), [Estado.java](#), [Planeta.java](#)

Introducción a los JavaBeans

[JavaBeans](#)

[Definición de JavaBean](#)

[Propiedades](#)

En la industria electrónica como en otras industrias se está acostumbrado a utilizar componentes para construir placas, tarjetas, etc. En el campo del software la idea es la misma. Se puede crear un interfaz de usuario en un programa Java en base a componentes: paneles, botones, etiquetas, caja de listas, barras de desplazamiento, diálogos, menús, etc.

Si se ha utilizado Delphi o Visual Basic, ya estamos familiarizados con la idea de componente, aunque el lenguaje de programación sea diferente. Existen componentes que van desde los más simples como un botón hasta otros mucho más complejos como un calendario, una hoja de cálculo, etc.

La primeros componentes que tuvieron gran éxito fueron los VBX (Visual Basic Extension), seguidos a continuación por los componentes OCX (OLE Custom Controls). Ahora bien, la principal ventaja de los JavaBeans es que son independientes de la plataforma.

Muchos componentes son visibles cuando se corre la aplicación, pero no tienen por qué serlo, solamente tienen que ser visibles en el momento de diseño, para que puedan ser manipulados por el Entorno de Desarrollo de Aplicaciones (IDE).

Podemos crear una aplicación en un IDE seleccionando los componentes visibles e invisibles en una paleta de herramientas y situarlas sobre un panel o una ventana. Con el ratón unimos los sucesos (events) que genera un objeto (fuente), con los objetos (listeners) interesados en responder a las acciones sobre dicho objeto. Por ejemplo, al mover el dedo en una barra de desplazamiento (fuente de sucesos) con el ratón, se cambia el texto (el número que indica la posición del dedo) en un control de edición (objeto interesado en los sucesos generados por la barra de desplazamiento).

Definición de JavaBean

Un JavaBean o bean es un componente hecho en software que se puede reutilizar y que puede ser manipulado visualmente por una herramienta de programación en lenguaje Java.

Para ello, se define un interfaz para el momento del diseño (design time) que permite a la herramienta de programación o IDE, interrogar (query) al componente y conocer las propiedades (**properties**) que define y los tipos de sucesos (**events**) que puede generar en respuesta a diversas acciones.

Aunque los beans individuales pueden variar ampliamente en funcionalidad desde los más simples a los más complejos, todos ellos comparten las siguientes características:

- **Introspection:** Permite analizar a la herramienta de programación o IDE como trabaja el bean
- **Customization:** El programador puede alterar la apariencia y la conducta del bean.
- **Events:** Informa al IDE de los sucesos que puede generar en respuesta a las acciones del usuario o del sistema, y también los sucesos que puede manejar.
- **Properties:** Permite cambiar los valores de las propiedades del bean para personalizarlo (customization).
- **Persistence:** Se puede guardar el estado de los beans que han sido personalizados por el programador, cambiando los valores de sus propiedades.

En general, un bean es una clase que obedece ciertas reglas:

- Un bean tiene que tener un constructor por defecto (sin argumentos)
- Un bean tiene que tener persistencia, es decir, implementar el [interface](#) *Serializable*.
- Un bean tiene que tener introspección (**introspection**). Los IDE reconocen ciertas pautas de diseño, nombres de las funciones miembros o métodos y definiciones de las clases, que permiten a la herramienta de programación mirar dentro del bean y conocer sus propiedades y su conducta.

Propiedades

Una propiedad es un atributo del JavaBean que afecta a su apariencia o a su conducta. Por ejemplo, un botón puede tener las siguientes propiedades: el tamaño, la posición, el título, el color de fondo, el color del texto, si está o no habilitado, etc.

Las propiedades de un bean pueden examinarse y modificarse mediante métodos o funciones miembro, que acceden a dicha propiedad, y pueden ser de dos tipos:

- **getter method:** lee el valor de la propiedad
- **setter method:** cambia el valor de la propiedad.

Un IDE que cumpla con las especificaciones de los JavaBeans sabe como analizar un bean y conocer sus propiedades. Además, crea una representación visual para cada uno de los tipos de propiedades, denominada editor de propiedades, para que el programador pueda modificarlas fácilmente en el momento del diseño.

Cuando un programador, coge un bean de la paleta de componentes y lo deposita en un panel, el IDE muestra el bean sobre el panel. Cuando seleccionamos el bean aparece una hoja de propiedades, que es una lista de las propiedades del

bean, con sus editores asociados para cada una de ellas.

El IDE llama a los métodos o funciones miembro que empiezan por **get**, para mostrar en los editores los valores de las propiedades. Si el programador cambia el valor de una propiedad se llama a un método cuyo nombre empieza por **set**, para actualizar el valor de dicha propiedad y que puede o no afectar al aspecto visual del bean en el momento del diseño.

Las especificaciones JavaBeans definen un conjunto de convenciones (design patterns) que el IDE usa para inferir qué métodos corresponden a propiedades.

```
public void setNombrePropiedad(TipoPropiedad valor)
```

```
public TipoPropiedad getNombrePropiedad( )
```

Cuando el IDE carga un bean, usa el mecanismo denominado *reflection* para examinar todos los métodos, fijándose en aquellos que empiezan por **set** y **get**. El IDE añade las propiedades que encuentra a la hoja de propiedades para que el programador personalice el bean.

Propiedades simples

Una propiedad simple representa un único valor.

Ejemplo

```
//miembro de la clase que se usa para guardar el valor de la propiedad

private String nombre;

//métodos set y get de la propiedad denominada Nombre

public void setNombre(String nuevoNombre){
    nombre=nuevoNombre;
}
public String getNombre(){
    return nombre;
}
```

En el caso de que dicha propiedad sea booleana se escribe

```
//miembro de la clase que se usa para guardar el valor de la propiedad

private boolean conectado=false;

//métodos set y get de la propiedad denominada Conectado

public void setConectado(boolean nuevoValor){
    conectado=nuevoValor;
}

public boolean isConectado(){
    return conectado;
}
```

Propiedades indexadas

Una propiedad indexada representa un array de valores.

```
//miembro de la clase que se usa para guardar el valor de la propiedad

private int[] numeros={1,2,3,4};

//métodos set y get de la propiedad denominada Numeros, para el array completo

public void setNumeros(int[] nuevoValor){
    numeros=nuevoValor;
}

public int[] getNumeros(){
    return numeros;
}

//métodos get y set para un elemento de array

public void setNumeros(int indice, int nuevoValor){
    numeros[indice]=nuevoValor;
}

public int getNumeros(int indice){
    return numeros[indice];
}
```

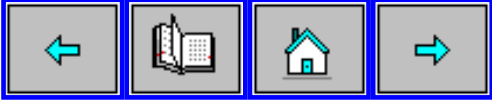

Propiedades ligadas (bound)

Los objetos de una clase que tiene una propiedad ligada notifican a otros objetos (listeners) interesados, cuando el valor de dicha propiedad cambia, permitiendo a estos objetos realizar alguna acción. Cuando la propiedad cambia, se crea un objeto (event) que contiene información acerca de la propiedad (su nombre, el valor previo y el nuevo valor), y lo pasa a los otros objetos (listeners) interesados en el cambio.

Propiedades restringidas (constrained)

Una propiedad restringida es similar a una propiedad ligada salvo que los objetos (listeners) a los que se les notifica el cambio del valor de la propiedad tienen la opción de vetar (veto) cualquier cambio en el valor de dicha propiedad. (Este tipo de propiedad no se estudiará en este capítulo).

Propiedades ligadas

[JavaBeans](#)

[Una clase con una propiedad ligada \(bound\)](#)
[La clase que define un suceso](#)
[El interface XXXListener](#)
[La fuente de los sucesos \(events\)](#)
[Los objetos \(listeners\) interesados](#)
[Vinculación entre la fuente de sucesos y los objetos \(listeners\) interesados](#)
[El código fuente](#)

Quando a un empleado (objeto de la clase *Asalariado*) le aumentan el sueldo, ha de notificar este hecho a otros: la familia, la Hacienda pública, etc. Vamos a estudiar con detalle los pasos necesarios para crear una clase *Asalariado* con una propiedad ligada (el sueldo) y otra clase *Hacienda* cuyos objetos (funcionarios) están interesados en el cambio en el valor de dicha propiedad.

Este ejemplo, nos permitirá profundizar aún más en el mecanismo que emplea Java 1.1 para [responder a las acciones del usuario](#) sobre los distintos controles y grupos de controles.

Una clase con una propiedad ligada (bound)

Creamos una clase denominada *Asalariado* con una propiedad ligada (bound) denominada *sueldo* de tipo **int**.

```
public class Asalariado{
    private int sueldo;

    public Asalariado() {
        sueldo=20;
    }
    public void setSueldo(int nuevoSueldo){
        sueldo=nuevoSueldo;
    }
}
```

```

    public int getSalario(){
        return sueldo;
    }
    //...
}

```

La clase *Asalariado* tiene un constructor por defecto, que asigna un valor inicial de 20 al miembro dato *sueldo*. Sueldo es una [propiedad](#) ya que tiene asociados dos métodos que empiezan por **set** y **get**.

Para notificar un cambio en dicha propiedad necesitamos llevar a cabo las siguientes tareas:

1. Crear una clase que defina un suceso (event) personalizado, denominada *XXXEvent*
2. Crear un interface denominado *XXXListener*, que declare los métodos que los objetos (listeners) a los que se le notifican el cambio en dicha propiedad, precisen implementar.
3. Crear un array (vector) que contenga la lista de objetos (listeners) interesados en el cambio en el valor de dicha propiedad.
4. Definir dos funciones denominadas *addXXXListener* y *removeXXXListener*, que añadan o eliminen objetos de dicha lista.

Vamos a estudiar detalladamente cada uno de los pasos:

La clase que define un suceso

Un suceso (event) es un objeto que indica que algo ha sucedido. Puede ser que el usuario haya movido el ratón, que un paquete de datos haya llegado a través de la red, etc. Cuando algo sucede, se ha de realizar alguna acción, por ejemplo, dibujar en la superficie del applet cuando se mueve el ratón, imprimir en la pantalla la información que ha llegado, etc.

La clase que define nuestro suceso (event) personalizado, que denominamos *SalarioEvent*, [deriva](#) de *EventObject*. Dicha clase tiene dos miembros dato, el sueldo que cobraba antes *anteSueldo*, y el sueldo que cobra ahora, *nuevoSueldo*. La clase base *EventObject* precisa conocer la fuente de los sucesos, que se le pasa en el primer parámetro del constructor luego, inicializa los dos miembros dato, que se pueden declarar **private** o **protected**, según convenga.

La clase define dos funciones miembro, *getNuevoSueldo* y *getAnteSueldo*, que permiten conocer los valores que guardan los dos miembros dato.

```
import java.util.*;

public class SalarioEvent extends EventObject {
    protected int anteSueldo, nuevoSueldo;
    public SalarioEvent(Object fuente, int anterior, int nuevo) {
        super(fuente);
        nuevoSueldo=nuevo;
        anteSueldo=anterior;
    }
    public int getNuevoSueldo(){ return nuevoSueldo;}
    public int getAnteSueldo(){ return anteSueldo;}
}
```

El interface XXXListener

Un [interface](#) es un grupo de métodos que implementan varias clases independientemente de su relación jerárquica, es decir, de que estén o no en una jerarquía.

La clase cuyos objetos (listeners) están interesados en el cambio en el valor de la propiedad ligada, ha de implementar un interface que se ha denominado *SalarioListener*. Dicho interface declara una única función *enteradoCambioSueldo* que ha de ser definida por la clase que implemente el interface.

```
import java.util.*;

public interface SalarioListener extends EventListener {
    public void enteradoCambioSueldo(EventObject e);
}
```

La fuente de los sucesos (events)

Un objeto que está interesado en recibir sucesos (events) se denomina *event listener*. Un objeto que produce los sucesos se llama *event source*, el cual mantiene una lista *salarioListeners* (objeto de [la clase Vector](#)) de objetos que están interesados en recibir sucesos y proporciona dos métodos para añadir *addSalarioListener* o eliminar *removeSalarioListener* dichos objetos de la lista.

```
public class Asalariado{
    private Vector salarioListeners=new Vector();

    public synchronized void addSalarioListener(SalarioListener listener){
        salarioListeners.addElement(listener);
    }
}
```

```

    }

    public synchronized void removeSalarioListener(SalarioListener listener){
        salarioListeners.removeElement(listener);
    }
    //...
}

```

Cada vez que se produce un cambio en el valor de la propiedad *Sueldo*, se ha de notificar dicho cambio a los objetos interesados que se guardan en el vector *salarioListeners*.

La función miembro o método que cambia la propiedad se denomina *setSueldo*. La tarea de dicha función como hemos visto anteriormente es la de actualizar el miembro dato *sueldo*, pero también tiene otras tareas como son las de crear un objeto de la clase *SalarioEvent* y notificar a los objetos interesados (listeners) de dicho cambio llamando a la función miembro *notificarCambio* y pasándole en su único argumento el objeto *event* creado.

Para crear un objeto *event* de la clase *SalarioEvent*, se precisa pasar al constructor tres datos: el objeto fuente de los sucesos, **this**, el sueldo que cobraba antes, *anteSueldo* y el sueldo que cobra ahora, *nuevoSueldo*.

```

public void setSueldo(int nuevoSueldo){
    int anteSueldo=sueldo;
    sueldo=nuevoSueldo;
    if(anteSueldo!=nuevoSueldo){
        SalarioEvent event=new SalarioEvent(this, anteSueldo, nuevoSueldo);
        notificarCambio(event);
    }
}

```

Se define la función *notificarCambio*, para notificar el cambio en la propiedad *Sueldo* a los objetos (listeners) que están interesados en cambio de dicha propiedad y que se guardan en el vector *salarioListeners*.. En dicha función, se crea una [copia](#) del vector *salarioListeners* y se guarda en la variable local *lista* de la clase *Vector*. La palabra clave [synchronized](#) evita que varios procesos ligeros o threads puedan acceder simultáneamente a la misma lista mientras se efectúa el proceso de copia.

```

Vector lista;
synchronized(this){
    lista=(Vector)salarioListeners.clone();
}

```

Finalmente, todos los objetos (listeners) interesados y que se guardan en el objeto *lista*, llaman a la función miembro *enteradoCambioSueldo*, ya que la clase que describe a dichos objetos, como veremos más adelante, implementa el interface *SalarioListener*. En el estudio de [la clase Vector](#) vimos como se accedía a cada uno de sus elementos.

```

for(int i=0; i<lista.size(); i++){
    SalarioListener listener=(SalarioListener)lista.elementAt(i);
    listener.enteradoCambioSueldo(event);
}

```

El código completo de la clase *Asalariado* es el siguiente

```
import java.beans.*;
import java.util.*;

public class Asalariado{
    private Vector salarioListeners=new Vector();
    private int sueldo;

    public Asalariado() {
        sueldo=20;
    }

    public void setSueldo(int nuevoSueldo){
        int anteSueldo=sueldo;
        sueldo=nuevoSueldo;
        if(anteSueldo!=nuevoSueldo){
            SalarioEvent event=new SalarioEvent(this, anteSueldo, nuevoSueldo);
            notificarCambio(event);
        }
    }

    public int getSalario(){
        return sueldo;
    }

    public synchronized void addSalarioListener(SalarioListener listener){
        salarioListeners.addElement(listener);
    }

    public synchronized void removeSalarioListener(SalarioListener listener){
        salarioListeners.removeElement(listener);
    }

    private void notificarCambio(SalarioEvent event){
        Vector lista;
        synchronized(this){
            lista=(Vector)salarioListeners.clone();
        }
        for(int i=0; i<lista.size(); i++){
            SalarioListener listener=(SalarioListener)lista.elementAt(i);
            listener.enteradoCambioSueldo(event);
        }
    }
}
```

Los objetos (listeners) interesados

La clase *Hacienda* que describe los objetos que están interesados en el cambio en el valor de la propiedad Sueldo, han de implementar el interface *SalarioListener* y definir la función *enteradoCambioSueldo*. Definimos una clase denominada *Hacienda* cuyos objetos (los funcionarios inspectores de hacienda) están interesados en el cambio de sueldo de los empleados.

```
public class Hacienda implements SalarioListener{

    public Hacienda() {
    }
    public void enteradoCambioSueldo(EventObject ev){
        if(ev instanceof SalarioEvent){
            SalarioEvent event=(SalarioEvent)ev;
            System.out.println("Hacienda: nuevo sueldo      "+event.getNuevoSueldo());
            System.out.println("Hacienda: sueldo anterior "+event.getAnteSueldo());
        }
    }
}
```

Como la función *enteradoCambioSueldo* recibe un objeto *ev* de la clase *SalarioEvent*, podemos extraer mediante las funciones miembro que se definen en dicha clase toda la información relativa al suceso: el sueldo que cobraba antes el empleado y el sueldo que cobra ahora. Esta información la obtenemos llamando a las funciones miembro *getNuevoSueldo* y *getAnteSueldo*. Con esta información el funcionario de Hacienda puede calcular las nuevas retenciones, impuestos, etc. En este caso, se limita, afortunadamente, a mostrar en la pantalla el sueldo anterior y el nuevo sueldo.

Vinculación entre la fuente de sucesos y los objetos (listeners) interesados

Para probar las clases *Asalariado* y *Hacienda* y comprobar como un objeto de la primera clase notifica el cambio en el valor de una de sus propiedades a un objeto de la segunda clase, creamos una aplicación.

En dicha aplicación, se crean dos objetos uno de cada una de las clases, llamando a su constructor por defecto o explícito según se requiera.

```
Hacienda funcionario1=new Hacienda();
Asalariado empleado=new Asalariado();
```

La vinculación entre el objeto fuente, *empleado*, y el objeto interesado en conocer el cambio en el valor de una de sus propiedades, *funcionario1* se realiza mediante la siguiente sentencia.

```
empleado.addSalarioListener(funcionario1);
```

El objeto *funcionario1* se añade a la lista (vector) de objetos interesados en conocer el nuevo sueldo del empleado.

Podemos poner más sentencias similares, para que más funcionarios de Hacienda sean notificados de dicho cambio. También podemos crear otra clase que se llame por ejemplo *Familia*, que implemente el interface *SalarioListener* y defina la función *enteradoCambioSueldo*. Creamos objetos de esta clase, la mujer, los hijos, etc, y los añadimos mediante *addSalarioListener* a la lista de personas (listeners) interesados en conocer la noticia.

Cuando escribimos la sentencia

```
empleado.setSueldo(50);
```

1. El objeto *empleado* llama a la función *setSueldo* que cambia la propiedad.
2. En el cuerpo de *setSueldo*, se comprueba que hay un cambio en el valor del miembro dato *sueldo*.
3. Se llama a la función miembro *notificarCambio* para dar a conocer a los objetos interesados que se guardan en el vector *salarioListeners* el cambio en el valor de dicha propiedad.
4. En el cuerpo de *notificarCambio*, los objetos (listeners) interesados llaman a la función *enteradoCambioSueldo*.
5. Los objetos interesados que pueden ser de la misma clase o de distinta clase, siempre que implementen el interface *SalarioListener*, realizan en el cuerpo de la función *enteradoCambioSueldo* las tareas, no siempre oportunas, con la información que se le proporciona a través del objeto *ev* de la clase *SalarioEvent*.

El código completo de la aplicación es el siguiente:

```
public class EjemploApp {

    public static void main(String[] args) {
        Hacienda funcionario1=new Hacienda();
        Asalariado empleado=new Asalariado();
        System.out.println("-----");
        empleado.addSalarioListener(funcionario1);
        empleado.setSueldo(50);
    }
}
```

El código fuente



[Asalariado.java](#), [SalarioEvent.java](#), [SalarioListener.java](#), [Hacienda.java](#), [EjemploApp.java](#)

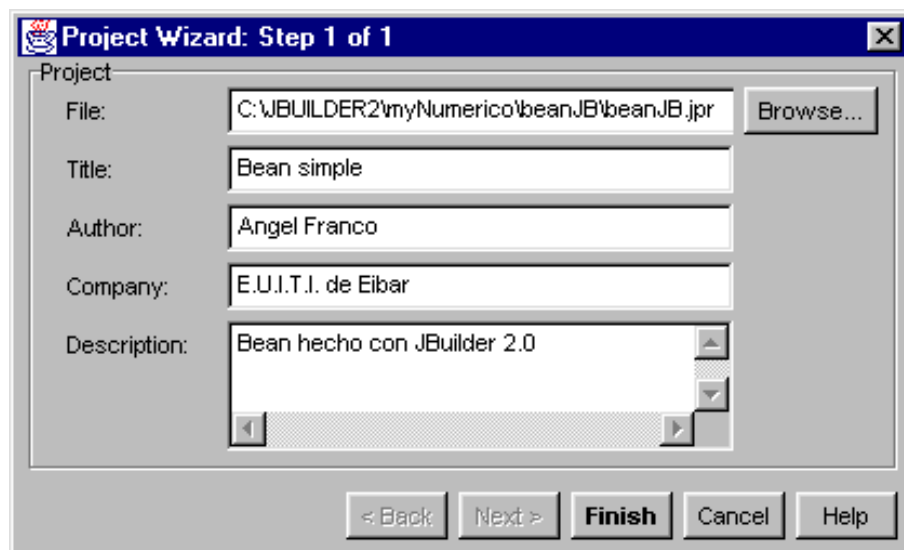
Creación de JavaBeans con JBuilder

[JavaBeans](#)[Un nuevo proyecto](#)[Añadir un bean al proyecto](#)[Definir una propiedad](#)[La clase cuyos objetos \(listeners\) están interesados en el cambio en el valor de la propiedad](#)[Vinculación entre la fuente de sucesos y los objetos \(listeners\) interesados](#)[El código fuente](#)

Una vez que hemos comprendido el funcionamiento básico de los JavaBeans y cómo interactúan entre sí. Vamos a aprovechar el Entorno Integrado de Desarrollo de JBuilder 2.0 para crear una aplicación similar a la estudiada en la página anterior pero sin apenas escribir código.

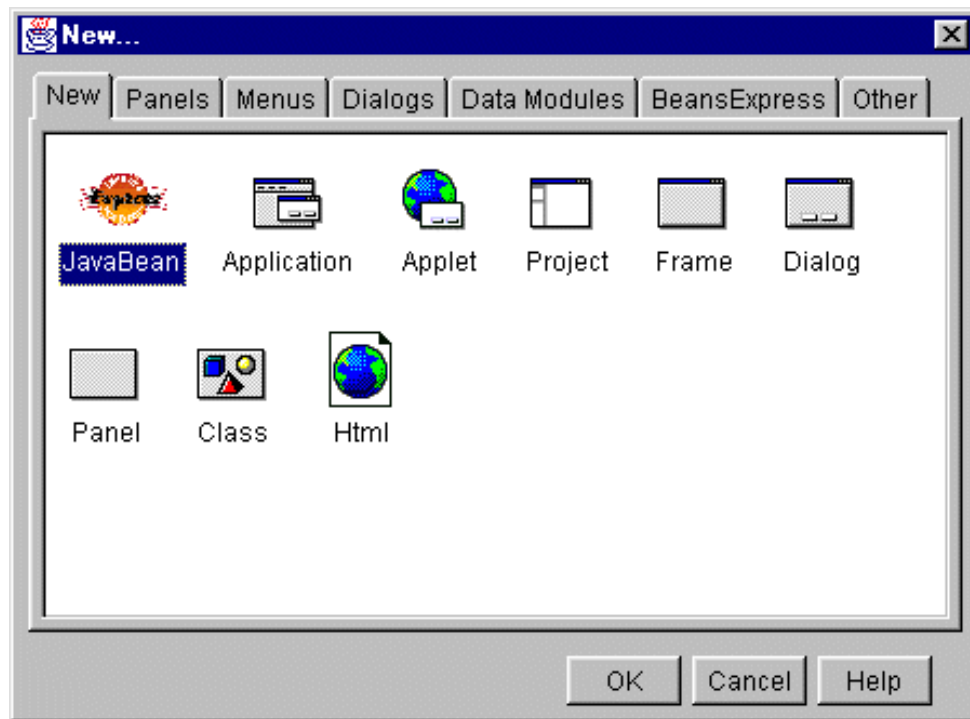
Un nuevo proyecto

Se selecciona **File/New project** y se escribe el nombre del proyecto, en este caso *beanJB* en el diálogo **Project Wizard**. Opcionalmente, se puede escribir la información relativa al título del bean, el autor, la empresa y una breve descripción del bean.

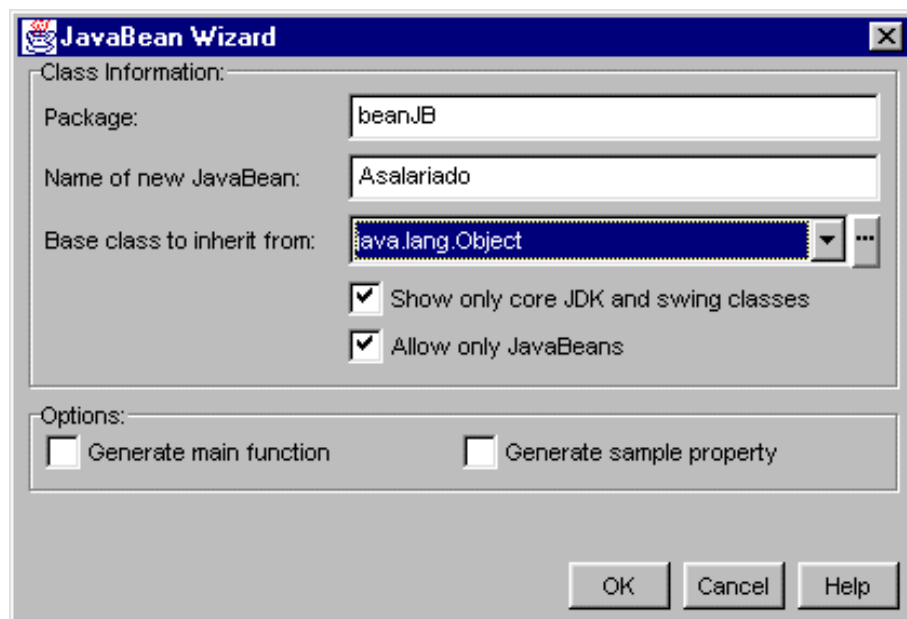


Añadir un bean al proyecto

Se selecciona **File/New** y en el diálogo **New** que aparece, se selecciona el icono **JavaBean** y se pulsa OK



A continuación aparece el diálogo **JavaBean Wizard** que nos pide el nombre del JavaBean, que lo denominamos *Asalariado* y a continuación, la clase de la cual deriva, en este caso, [la clase base Object](#) la raíz de todas las clases en Java.



Observamos el código generado. La clase *Asalariado* deriva implícitamente de *Object* y tiene un constructor por defecto (sin parámetros)

```
package beanJB;

import java.beans.*;
```

```
public class Asalariado {
    public Asalariado() {
    }
}
```

Definir una propiedad

Al seleccionar en la parte inferior del IDE la pestaña **Bean** (al lado de las pestañas Source y Design y Doc), aparece otro grupo de pestañas. Se pulsa con el puntero del ratón sobre la pestaña **Properties** y a continuación, se pulsa en el botón **Add Property**, apareciendo el diálogo **New Property**

New Property

Property data:

Property Name:

Type: ...

☒ Getter

☒ Setter

Binding:

BeanInfo data:

☒ Expose through BeanInfo

Display Name:

Short Description:

Editor:

OK Cancel Help Apply

En el primer control de edición, se introduce el nombre de la propiedad, *sueldo*. En el segundo, el tipo (*String* por defecto), borramos *String* y ponemos **int**. Las casillas indican que se generarán dos métodos para acceder a esta propiedad uno que empieza por **set** y el otro por **get**. Finalmente, se selecciona **bound** (ligada) como característica de la propiedad *sueldo*. La parte inferior del diálogo no la modificamos.

Se pulsa el botón OK y se genera nuevo código. Solamente, precisamos inicializar el miembro dato *sueldo* en el constructor. Por defecto, la variable *sueldo* toma el valor cero.

```

package beanJB;

import java.beans.*;

public class Asalariado {
    private int sueldo;
    private transient PropertyChangeSupport propertyChangeListeners =
        new PropertyChangeSupport(this);

    public Asalariado() {
        sueldo=20;
    }

    public int getSueldo() {
        return sueldo;
    }

    public void setSueldo(int newSueldo) {
        int oldSueldo = sueldo;
        sueldo = newSueldo;
        propertyChangeListeners.firePropertyChange("sueldo", new Integer(oldSueldo), new
Integer(newSueldo));
    }

    public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.removePropertyChangeListener(l);
    }

    public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.addPropertyChangeListener(l);
    }
}

```

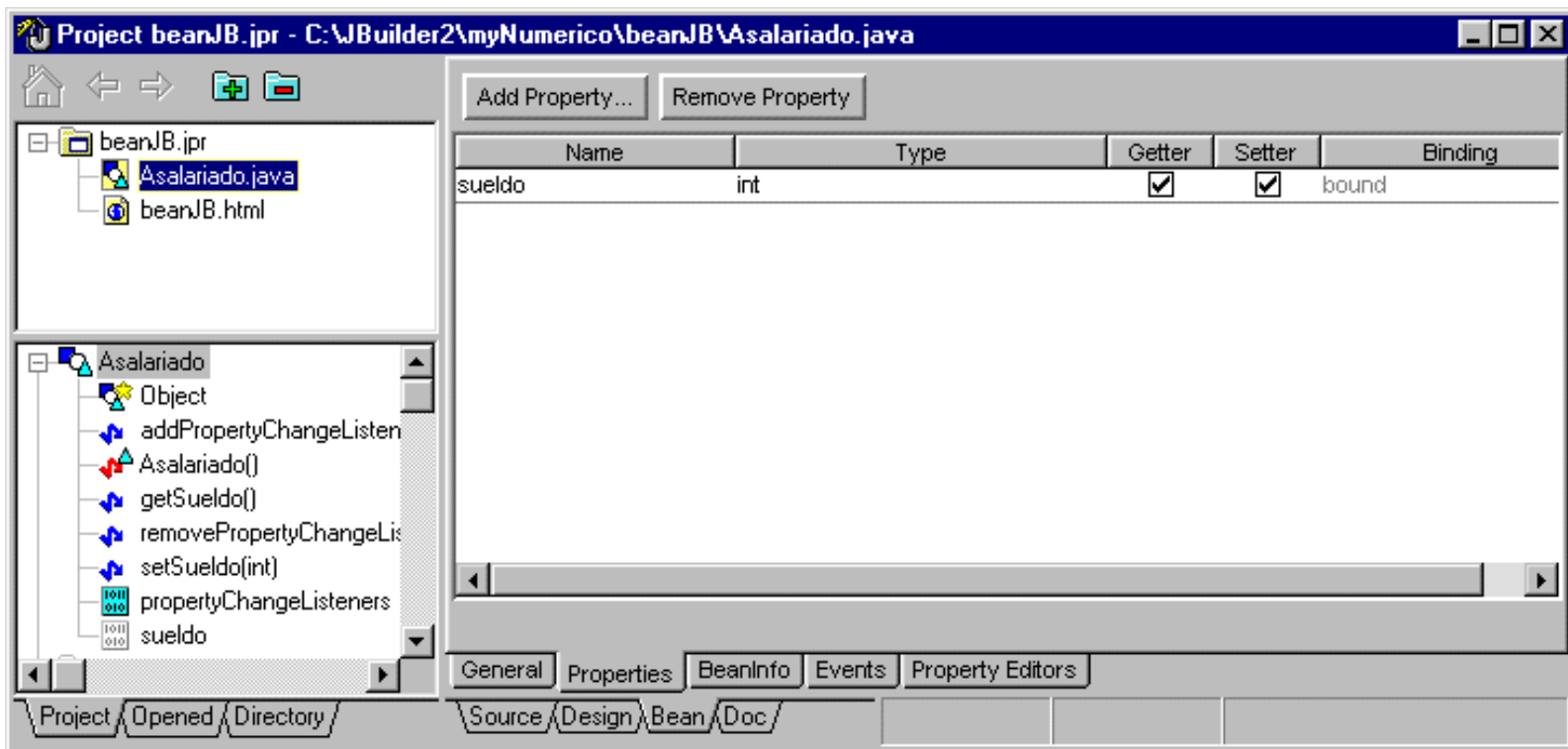
En la definición de la clase [Asalariado](#) en vez de un vector que guarda los objetos (listeners) interesados en el cambio de la propiedad ligada, aparece un objeto denominado *propertyChangeListeners* de la clase *PropertyChangeSupport*.

Se generan también las funciones que añaden *addPropertyChangeListener* o eliminan *removePropertyChangeListener* objetos de dicha lista.

Cuando se produce un cambio en la propiedad *sueldo*, se llama a la función *setSueldo*. En el cuerpo de dicha función, los objetos (listeners) interesados en ser notificados, llaman a la función *firePropertyChange*, pasándole la información relativa al suceso: el nombre de la propiedad, el valor previo de la propiedad (*oldSueldo*), el nuevo valor de la propiedad (*newSueldo*).

El nombre del [interface](#) es *PropertyChangeListener* y no precisa definición, ya que está dentro del paquete **java.beans.***

Una vez que hemos cerrado el diálogo **New Property** que define la propiedad, el contenido del panel **Properties** es el siguiente.



La clase cuyos objetos (listeners) están interesados en el cambio en el valor de la propiedad

Se añade un nuevo bean al proyecto denominado *Hacienda*, siguiendo los mismos pasos que para crear *Asalariado*.

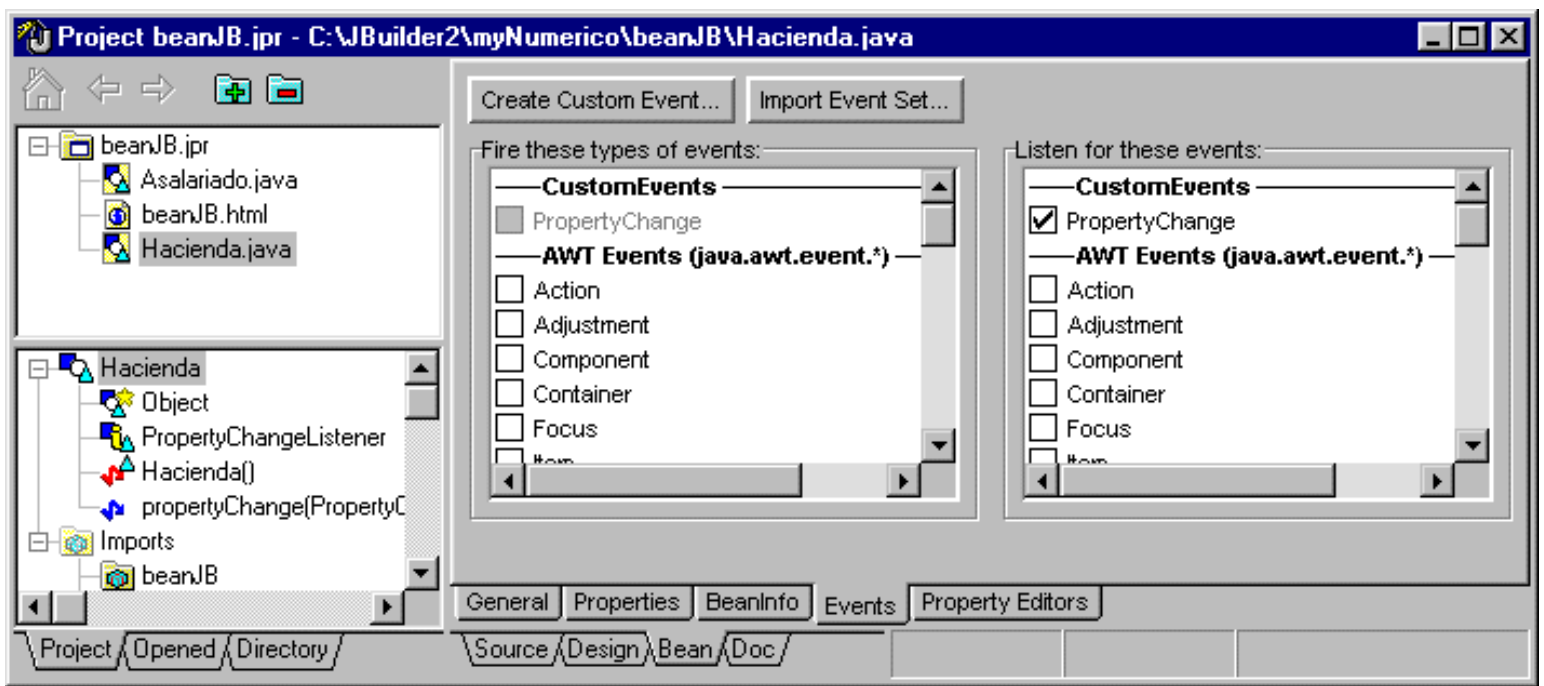
Se selecciona **File/New** y a continuación en el [diálogo New](#) se selecciona el icono **JavaBean**. En el diálogo titulado **JavaBean Wizard** se introduce el nombre del bean *Hacienda* y la clase de la cual deriva, la clase base *Object*.

Se genera el siguiente código

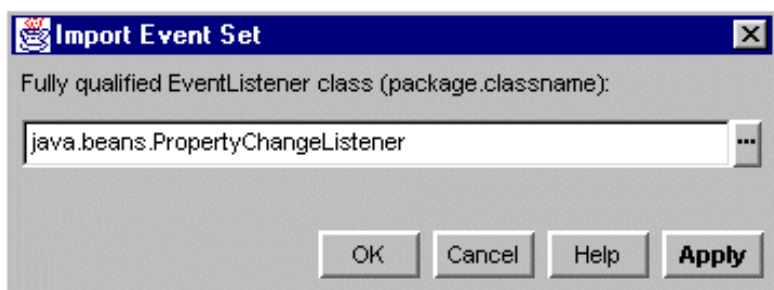
```
package beanJB;
public class Hacienda {

    public Hacienda() {
    }
}
```

Ahora seleccionamos la pestaña **Bean** y continuación la pestaña **Events**. Aparecen dos listas, la izquierda se titula **Fire these types of events**, y la derecha **Listen for these events**. La objetos (listeners) de la clase *Hacienda* están interesados en escuchar (listen) los sucesos (events) que emite (fire) los objetos de la clase *Asalariado*, luego iremos al panel de la derecha. El suceso es personalizado, **Custom event**, y resulta de un cambio en el valor de una propiedad **PropertyChange**. Por tanto, activamos esta casilla tal como se ve en la figura inferior



Si no aparecen los **CustomEvents** en la parte superior de los panels **Events** tal como se muestra en la figura, se pulsa el botón titulado **Import Event Set** y en el diálogo que aparece titulado **Import Event Set**, se introduce `java.beans.PropertyChangeListener`, tal como aparece en la figura inferior, y se pulsa el botón titulado OK para cerrar el diálogo.



A continuación, se pulsa la casilla del panel de la derecha **Listen for these events** titulada **PropertyChange**.

Al seleccionar la pestaña **Source** vemos que se ha generado el código siguiente.

```
package beanJB;
import java.beans.*;
public class Hacienda implements PropertyChangeListener {

    public Hacienda() {
    }

    public void propertyChange(PropertyChangeEvent e) {
    }
}
```

El programador solamente tiene que preocuparse por gestionar la información que le proporciona el suceso *e* de la clase *PropertyChangeEvent*, en el cuerpo de la función miembro *propertyChange*.

Como ya se ha mencionado, dicho suceso proporciona tres datos: el nombre de la propiedad, el valor previo y el nuevo valor, que se obtienen mediante las siguientes funciones miembro: *getPropertyName*, *getNewValue*, *getOldValue*. Nos limitaremos de

momento, a mostrar estos tres valores.

El código completo de la clase *Hacienda* es el siguiente

```
package beanJB;
import java.beans.*;

public class Hacienda implements PropertyChangeListener {

    public Hacienda() {

    }

    public void propertyChange(PropertyChangeEvent e) {
        System.out.println("Hacienda "+e.getPropertyName()+" nuevo:
"+e.getNewValue());
        System.out.println("Hacienda "+e.getPropertyName()+" anterior:
"+e.getOldValue());
    }
}
```

Vinculación entre la fuente de sucesos y los objetos (listeners) interesados

Para probar las clases *Asalariado* y *Hacienda* y comprobar como un objeto de la primera clase notifica el cambio en una de sus propiedades a un objeto de la segunda clase, creamos la aplicación [EjemploApp](#) similar a la estudiada en la página anterior.

En dicha aplicación, se crean dos objetos uno por cada una de las clases, llamando a su constructor por defecto o explícito según se requiera.

```
Hacienda funcionario1=new Hacienda();
Asalariado empleado=new Asalariado();
```

La vinculación entre el objeto fuente, *empleado*, y el objeto *funcionario1* interesado en conocer el cambio en el valor de su propiedad Sueldo, se realiza mediante la siguiente sentencia.

```
empleado.addPropertyChangeListener(funcionario1);
```

Cuando escribimos la sentencia

```
empleado.setSueldo(50);
```

1. El objeto *empleado* llama a la función *setSueldo* que cambia la propiedad.
2. En el cuerpo de *setSueldo*, cada uno de los objetos (listeners) interesados en el cambio en el valor de la propiedad ligada, llama a *firePropertyChange*, y le pasa la información relativa al suceso (event) generado: el nombre de la propiedad, el valor previo, y el nuevo valor.

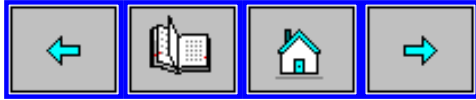
3. Se llama a la función miembro *propertyChange* miembro de la clase que implementa el interface *PropertyChangeListener* cuyos objetos (listeners) están interesados en el cambio en el valor de la propiedad .
4. En el cuerpo de la función *propertyChange*, se realizan las tareas relativas al tratamiento de la información que se le proporciona a través del suceso *e* de la clase *PropertyChangeEvent*.

El código fuente



[Asalariado.java](#), [Hacienda.java](#), [EjemploApp.java](#)

Un botón rudimentario como bean



[JavaBeans](#)

[Crear un proyecto](#)

[Añadir un bean al proyecto](#)

[Propiedades simples](#)

[Persistencia](#)

[Aspecto del botón](#)

[El tamaño del canvas](#)

[Relación entre el aspecto y las propiedades del bean](#)

[Respondiendo a las acciones del usuario](#)

[El botón emite un suceso del tipo *ActionEvent*.](#)

[El bean terminado](#)

[La clase *BeanInfo*](#)

[Deployment](#)

[El código fuente](#)

En esta página vamos a estudiar como se crea un JavaBean y cómo se pone en la barra de herramientas. El bean va a consistir en un control muy simple que simula un botón. Comprobaremos cómo funciona el bean arrastrándolo con el ratón desde la barra de herramientas y depositándolo en el applet. Veremos que al pulsar con el ratón sobre el botón se realiza cierta acción consistente, en el cambio del texto de un control etiqueta (label).

Crear un proyecto

Ya hemos estudiado en parte los pasos para crear un bean con el asistente **JavaBean Wizard**.

Creamos un nuevo proyecto seleccionando [File/New Project](#)

Añadir un bean al proyecto

[Añadimos al proyecto un JavaBean](#) seleccionado **File/New** y a continuación en el diálogo **New** seleccionamos el icono **JavaBean**

En el diálogo que aparece titulado **JavaBean Wizard** ponemos *Boton* en el campo **Name of new JavaBean**, el nombre del bean, y en el campo **Base class to inherit from** seleccionamos `java.awt.Panel`, aunque nuestra intención es que derive de `java.awt.Canvas`, pero esta opción no está habilitada.

El IDE genera el siguiente código

```
package bean1;

import java.awt.*;

public class Boton extends Panel {
    BorderLayout borderLayout1 = new BorderLayout();

    public Boton() {
        try {
            jbInit();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setLayout(borderLayout1);
    }
}
```

Modificamos el código para que la clase *Boton* derive de *Canvas*, quedando del siguiente modo.

```
import java.awt.*;

public class Boton extends Canvas {

    public Boton() {
        try {
            jbInit();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Un botón rudimentario como bean

```
private void jbInit() throws Exception {  
}  
}
```

Propiedades simples

El botón va a tener dos [propiedades simples](#) el color y el texto o título del botón.

Para [definir dichas propiedades](#) seleccionamos la pestaña **Bean** y a continuación la pestaña **Properties**, y pulsamos en el botón **addProperty**, apareciendo el diálogo titulado **New Property**

Para definir la propiedad *titulo* ponemos

| | |
|---------------|--------|
| Property name | titulo |
| Type | String |
| Binding | none |

En la tercera fila, **none** indica una propiedad simple

A continuación, pulsamos el botón **Apply**.

Para definir la propiedad *color* ponemos

| | |
|---------------|-------|
| Property name | color |
| Type | Color |
| Binding | none |

A continuación, pulsamos el botón **OK** para cerrar el diálogo

En el panel **Bean/Properties** queda reflejado las dos nuevas propiedades que ocupan las dos primeras filas.

En el código fuente, se definen dos nuevos miembros datos, y dos funciones miembro que empiezan por **set** y **get** para cada uno de dichos miembros. (Se ha retocado ligeramente el código para simplificarlo, evitando redundancias)

```
import java.awt.*;  
  
public class Boton extends Canvas{  
    private String titulo;  
    private Color color;  
  
    //...  
    public void setTitulo(String newTitulo) {  
        titulo = newTitulo;  
    }  
    public String getTitulo() {
```

Un botón rudimentario como bean

```
    return titulo;
}

public void setColor(Color newColor) {
    color = newColor;
}

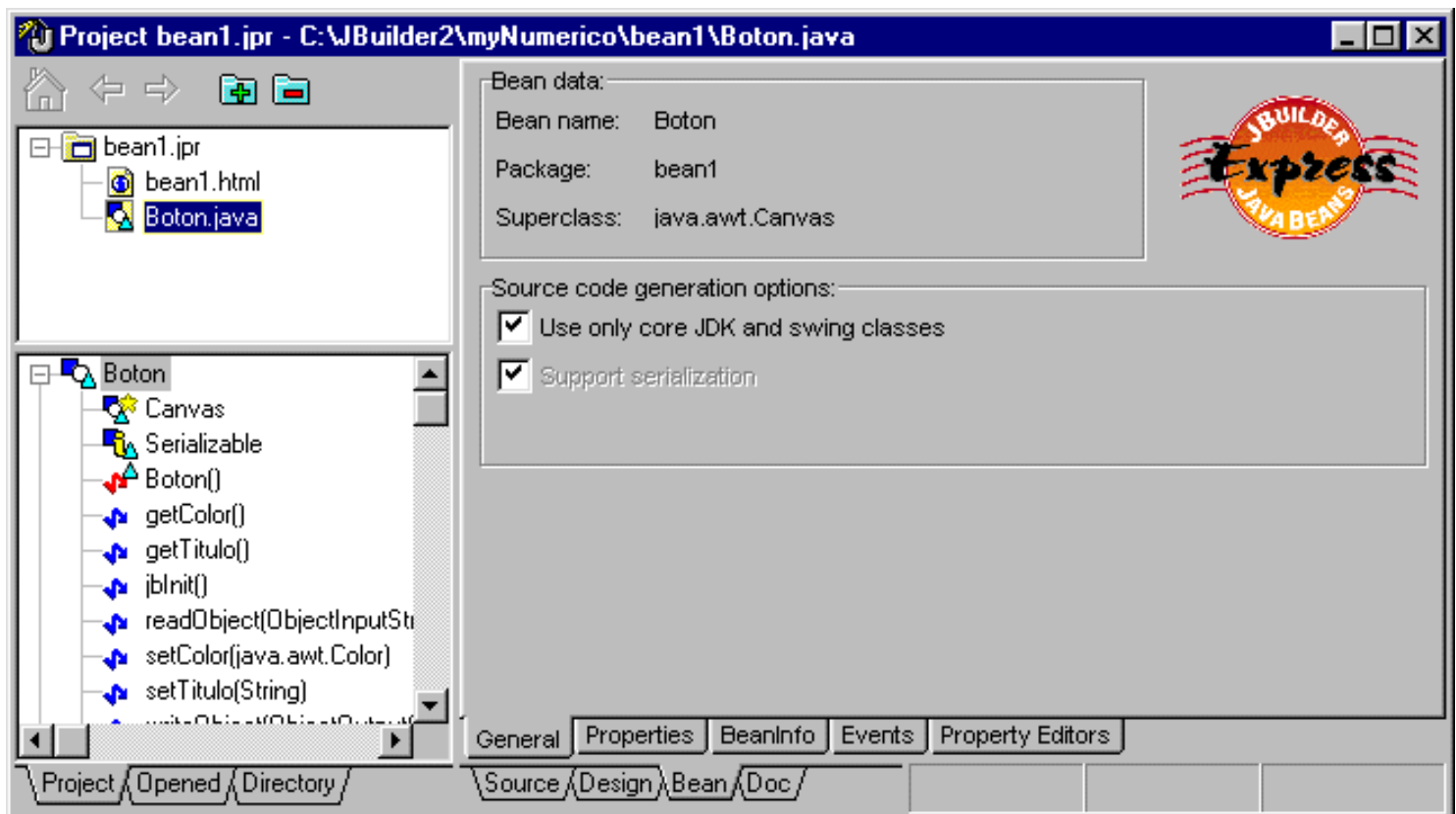
public Color getColor() {
    return color;
}
}
```

En *init* o *jbInit* establecemos el tamaño inicial del canvas, los valores iniciales de las propiedades *color* y *titulo*, así como de la fuente de texto empleada para mostrar el título centrado en el canvas.

```
private void jbInit() throws Exception {
    setSize(60,40);
    this.titulo="Bean";
    color=Color.yellow;
    setFont(new Font("Dialog", Font.BOLD, 12));
}
```

Persistencia

Vamos al panel **Bean/General**, y activamos la casilla, **Support serialization**. En la introducción a los JavaBeans hemos visto que esta es una de sus características.



[La persistencia](#) significa que podemos guardar el estado de un bean una vez que ha sido personalizado por el programador.

Para que el bean tenga esta característica ha de implementar el interface *Serializable*. Dicho estado se recupera cuando volvemos a correr el IDE con el programa que usa el bean.

```
import java.awt.*;

public class Boton extends Canvas implements java.io.Serializable {
    //...
}
```

El IDE añade dos funciones miembro *readObject* y *writeObject* que en principio, no son necesarias por lo que las eliminamos del código.

Aspecto del botón

Para dibujar el botón redefinimos la función *paint* de la clase base *Canvas*. El botón va a consistir en una región rectangular (el canvas) pintado del color que guarda la propiedad *color*, con un borde de forma rectangular y con el [título centrado](#).

```
public synchronized void paint(Graphics g) {
    int ancho=getSize().width;
    int alto=getSize().height;

    g.setColor(color);
    g.fillRect(1, 1, ancho-2, alto-2);
    g.draw3DRect(0, 0, ancho-1, alto-1, false);

    g.setColor(getForeground());
    g.setFont(getFont());

    g.drawRect(2, 2, ancho-4, alto-4);
    FontMetrics fm = g.getFontMetrics();
    g.drawString(titulo, (ancho-fm.stringWidth(titulo))/2,
        (alto+fm.getMaxAscent()-fm.getMaxDescent())/2);
}
```

El tamaño del canvas

Por defecto, [el control Canvas](#) no tiene tamaño, lo puede representar un problema cuando disponemos varios componentes en el applet. Por ejemplo, pueden ocurrir situaciones en las que el canvas no se vea. En este caso, es necesario redefinir las funciones miembro *getPreferredSize* y *getMinumunSize*, ambas funciones devuelven un objeto de la clase *Dimension*.

Determinamos el tamaño del canvas como la suma del [tamaño del texto](#) más un cierto margen, definido por las constantes MARGEN_X y MARGEN_Y.

```
public class Boton extends Canvas implements java.io.Serializable {
    private String titulo;
    private static final int MARGEN_X=12;
    private static final int MARGEN_Y=8;
```

```
//...
public Dimension getPreferredSize() {
    FontMetrics fm=getFontMetrics(getFont());
    return new Dimension(fm.stringWidth(titulo)+MARGEN_X,
        fm.getMaxAscent()+fm.getMaxDescent()+MARGEN_Y);
}

public Dimension getMinimumSize() {
    return getPreferredSize();
}
}
```

Relación entre el aspecto y las propiedades del bean

Cuando cambiamos la propiedad *color*, mediante la función miembro *setColor*, se ha de reflejar en el aspecto del botón. La forma de hacerlo es volver a dibujar el botón, llamando a la función *paint*.

```
public void setColor(Color newColor) {
    color = newColor;
    repaint();
}
```

Cuando se cambia la propiedad *titulo* mediante *setTitulo*, el tamaño del botón ha de cambiar para acomodar al nuevo texto. El código es algo más complicado. Cuando se cambia el *titulo* se calcula la nueva dimensión del canvas mediante *getPrefferedSize* y se cambia la dimensión del canvas mediante *setSize*. .

```
public void setTitulo(String newTitulo) {
    titulo = newTitulo;
    Dimension d = getPreferredSize();
    setSize(d.width, d.height);
    invalidate();
}
```

Respondiendo a las acciones del usuario

Un control botón se activa pulsando el botón izquierdo del ratón (*mousePressed*), el control parece hundirse y a la vez aparece un rectángulo dibujado a puntos alrededor del título. Cuando se deja de pulsar el botón izquierdo (*mouseReleased*) se realiza la acción: cerrar un diálogo, compilar un programa, guardar un archivo, etc.. Por tanto, hemos de definir las funciones respuesta a estas dos acciones del usuario sobre el botón simulado.

Pulsamos con el ratón sobre la pestaña **Design**, y vamos a la hoja de propiedades y sucesos del canvas, pulsamos sobre la pestaña **Events** y hacemos doble-clic en el editor situado a la derecha de *mousePressed*, y luego hacemos lo mismo con *mouseReleased*..

Se genera el código correspondiente a la [respuesta a las acciones del usuario](#) sobre dicho componente Se relaciona mediante *addMouseListener* la fuente de los sucesos, el canvas, **this**, con un objeto de una clase anónima que implementa el interface *MouseListener* y define las funciones *mousePressed* y *mouseReleased*.

```

private void jbInit() throws Exception {
//...
this.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        this_mousePressed(e);
    }
    public void mouseReleased(MouseEvent e) {
        this_mouseReleased(e);
    }
});

void this_mousePressed(MouseEvent e) {
//poner aquí el código de la función respuesta
}
void this_mouseReleased(MouseEvent e) {
//poner aquí el código de la función respuesta
}

```

El botón emite un suceso del tipo *ActionEvent*.

Cada vez que se pulsa sobre un botón se realiza una acción. El botón emite un suceso del tipo *ActionEvent*. Por tanto, hemos de programar nuestro bean para que emita sucesos de este tipo.

Seleccionamos la [pestaña Bean y a continuación Events](#), en el panel aparecen dos listas. Nos detenemos en la de la izquierda **Fire these types of events**, y activamos la casilla titulada **Action** debajo de **AWT events**. Al pulsar sobre la pestaña **Source** veremos que se ha generado nuevo código.

```

public class Boton extends Canvas implements java.io.Serializable{
    private transient Vector actionListeners;
//...
    public synchronized void removeActionListener(ActionListener l) {
        if (actionListeners != null && actionListeners.contains(l)) {
            Vector v = (Vector) actionListeners.clone();
            v.removeElement(l);
            actionListeners = v;
        }
    }

    public synchronized void addActionListener(ActionListener l) {
        Vector v = actionListeners == null ? new Vector(2) : (Vector)
actionListeners.clone();
        if (!v.contains(l)) {
            v.addElement(l);
            actionListeners = v;
        }
    }

    protected void fireActionPerformed(ActionEvent e) {
        if (actionListeners != null) {
            Vector listeners = actionListeners;

```

```

        int count = listeners.size();
        for (int i = 0; i < count; i++)
            ((ActionListener) listeners.elementAt(i)).actionPerformed(e);
    }
}

```

Se añade a la clase que describe el bean un miembro dato *actionListeners* de la clase *Vector* que tiene delante el modificador **transient**. Una clase que implementa el interface *Serializable* guardará en disco los valores de sus miembros dato, excepto aquellos que están marcados por la palabra clave **transient**.

Al explicar el significado de una propiedad ligada (bound), vimos cómo se [notifica a los objetos \(listeners\) interesados](#) el cambio en dicha propiedad. El código como podemos observar es similar.

Cada vez que se pulsa el botón izquierdo del ratón sobre el bean y a continuación se libera, se ha de notificar dicha acción a los objetos interesados que se guardan en el vector *actionListeners* y también, se ha de reflejar esta circunstancia en el aspecto del botón. En este ejemplo, estudiaremos solamente cómo se notifica dicha acción.

En la definición de la función respuesta *mousePressed* o bien *this_mousePressed*, que se llama cuando se pulsa sobre el botón izquierdo del ratón, la variable *bPulsado* del tipo **boolean** toma el valor **true**.

En la definición de la función respuesta *mouseReleased* o bien *this_mouseReleased*, que se llama cuando se libera el botón izquierdo del ratón, la variable *bPulsado* del tipo **boolean** toma el valor **false**. Según sea el valor de esta variable el aspecto del botón cambia de normal a pulsado (aparece como hundido y con un rectángulo punteado alrededor del título).

Cuando se deja de pulsar el botón izquierdo del ratón se emite un suceso del tipo *ActionEvent*. En el cuerpo de la función respuesta *this_mouseReleased* se llama a la función *fireActionPerformed*. En dicha función, todos los objetos (listeners) interesados y que se guardan en el vector *listeners*, llaman a la función miembro *actionPerformed*, siempre que la clase que describe a dichos objetos implemente el interface *ActionListener*.

```

void this_mousePressed(MouseEvent e) {
    bPulsado=true;
}

void this_mouseReleased(MouseEvent e) {
    if(bPulsado){
        bPulsado=false;
        ActionEvent ev=new ActionEvent(e.getSource(), e.getID(), titulo);
        fireActionPerformed(ev);
    }
}

```

El único problema que se presenta en la definición de la función respuesta *this_mouseReleased* es la conversión de un suceso (event) de la clase *MouseEvent* que proporciona la función respuesta a un suceso de la clase *ActionEvent*, que requiere la función *fireActionPerformed*.

Esta transformación se lleva a cabo en el cuerpo de la función respuesta *this_mouseReleased*. Extraemos del objeto *e* de la clase *MouseEvent*, la fuente de los sucesos mediante *getSource*, el identificador mediante *getID*, y se los pasamos en los dos primeros parámetros del constructor de *ActionEvent*. El tercer parámetro se denomina *command* y es el título del botón.

El bean terminado

El código completo del bean denominado *Boton* es el siguiente

```
package bean1;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Boton extends Canvas implements java.io.Serializable{
    private String titulo;
    private Color color;
    private static final int MARGEN_X=12;
    private static final int MARGEN_Y=8;
    private transient Vector actionListeners;
    boolean bPulsado=false;
    public Boton() {
        try {
            jbInit();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        setSize(60,40);
        this.titulo="Bean";
        color=Color.yellow;
        setFont(new Font("Dialog", Font.BOLD, 12));
        this.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                this_mousePressed(e);
            }
            public void mouseReleased(MouseEvent e) {
                this_mouseReleased(e);
            }
        });
    }

    public void setTitulo(String newTitulo) {
        titulo = newTitulo;
        Dimension d = getPreferredSize();
        setSize(d.width, d.height);
        invalidate();
    }

    public String getTitulo() {
        return titulo;
    }
}
```

```

    }

    public void setColor(Color newColor) {
        color = newColor;
        repaint();
    }

    public Color getColor() {
        return color;
    }

    public synchronized void paint(Graphics g) {
        int ancho=getSize().width;
        int alto=getSize().height;

        g.setColor(color);
        g.fillRect(1, 1, ancho-2, alto-2);
        g.draw3DRect(0, 0, ancho-1, alto-1, false);

        g.setColor(getForeground());
        g.setFont(getFont());

        g.drawRect(2, 2, ancho-4, alto-4);
        FontMetrics fm = g.getFontMetrics();
        g.drawString(titulo, (ancho-fm.stringWidth(titulo))/2, (alto+fm.getMaxAscent()-
fm.getMaxDescent())/2);
    }

    public Dimension getPreferredSize() {
        FontMetrics fm=getFontMetrics(getFont());
        return new Dimension(fm.stringWidth(titulo)+MARGEN_X,
fm.getMaxAscent()+fm.getMaxDescent()+MARGEN_Y);
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    void this_mousePressed(MouseEvent e) {
        bPulsado=true;
    }

    void this_mouseReleased(MouseEvent e) {
        if(bPulsado){
            bPulsado=false;
            ActionEvent ev=new ActionEvent(e.getSource(), e.getID(), titulo);
            fireActionPerformed(ev);
        }
    }

    public synchronized void removeActionListener(ActionListener l) {
        if (actionListeners != null && actionListeners.contains(l)) {
            Vector v = (Vector) actionListeners.clone();
            v.removeElement(l);
            actionListeners = v;
        }
    }
}

```

```

    public synchronized void addActionListener(ActionListener l) {
        Vector v = actionListeners == null ? new Vector(2) : (Vector)
actionListeners.clone();
        if (!v.contains(l)) {
            v.addElement(l);
            actionListeners = v;
        }
    }

    protected void fireActionPerformed(ActionEvent e) {
        if (actionListeners != null) {
            Vector listeners = actionListeners;
            int count = listeners.size();
            for (int i = 0; i < count; i++)
                ((ActionListener) listeners.elementAt(i)).actionPerformed(e);
        }
    }
}

```

La clase *BeanInfo*

Como se ha explicado en la [introducción a los JavaBeans](#), el IDE descubre las propiedades y los sucesos (events) de un bean a través del mecanismo denominado introspection. Otra forma de hacerlo, es a través de una clase que implemente el interface *BeanInfo*. Habitualmente las clases derivan de *SimpleBeanInfo* en vez de implementar todos los métodos del interface *BeanInfo*. Recuérdese la [diferencia entre *Listeners* y *Adapters*](#). Para que el IDE encuentre la correspondiente clase *BeanInfo* el nombre de la clase debe ser el mismo que el nombre del bean seguido por el string *BeanInfo*, por ejemplo *BotonBeanInfo*.

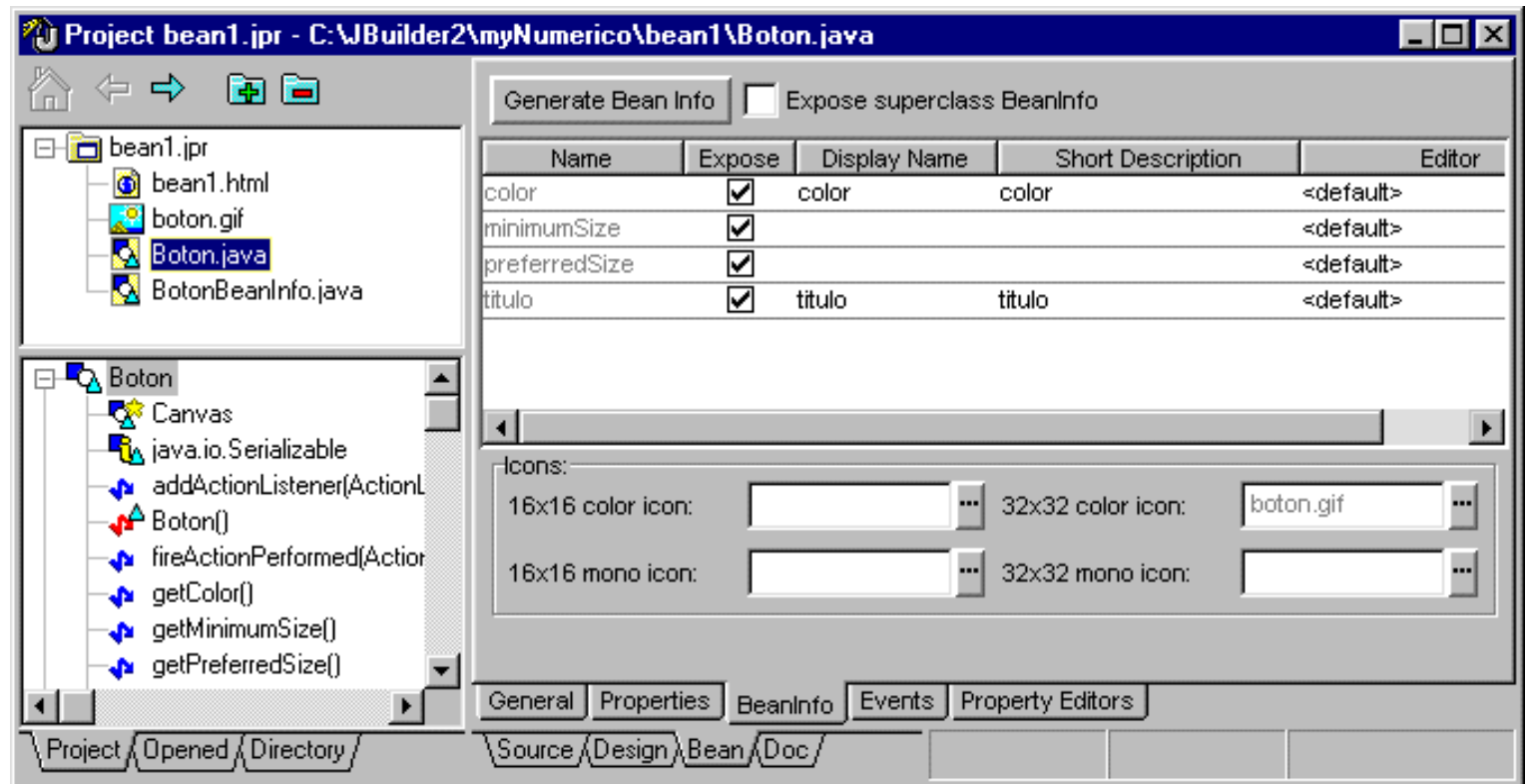
Una clase *BeanInfo* especifica la siguiente información:

- El icono que representa al bean
- Un objeto *BeanDescriptor* que contiene una referencia a una clase *Customizer*, que no estudiaremos en este capítulo.
- Una lista de propiedades del bean, con una breve descripción de estas propiedades. Esta descripción será utilizada por la hoja de propiedades del componente en el IDE en el momento de diseño. Se incluye el nombre de la propiedad, su valor, el tool tip (breve descripción), que son los datos que se introducen en el diálogo [New Property](#).
- Referencias al editor de propiedades, que no estudiaremos en este capítulo
- Una lista de métodos que define el bean, con una descripción de cada uno de ellos.

La clase que implementa el interface *BeanInfo* se usa antes que el mecanismo de la introspección. Añadiendo una clase *BeanInfo* a nuestro bean podemos incluso ocultar propiedades que no queremos que se muestren en la hoja de propiedades del componente en el IDE, que de otro modo si serían mostradas.

Una vez que el bean se ha compilado con éxito, creamos un icono representativo del mismo, para que se pueda identificar en la barra de herramientas. Dicho icono, lo ponemos en el subdirectorio del proyecto y lo añadimos al mismo pulsando en el botón + del panel de navegación, véase la figura inferior.

Seleccionado la pestaña **Bean** y a continuación **BeanInfo** aparece un panel, con información relativa al bean, y cuatro campos en la parte inferior en el que podemos poner el nombre de los archivos que guardan los iconos representativos del bean. Se pueden poner cuatro iconos en blanco y negro, y en color, en una tamaño de 16x16 o de 32x32. En nuestro caso hemos creado un único icono en color y con un tamaño de 32x32 pixels.



A continuación, pulsamos en el botón titulado **Generate Bean Info**, situado en la parte superior izquierda. Se crea una clase que se guarda en el archivo BotonBeanInfo.java que se añade automáticamente al proyecto.

De este modo, se concluye el proyecto que está formado, por el código fuente del bean, un archivo que contiene una clase con información relativa al bean, y el icono o los iconos representativos del bean.

Deployment

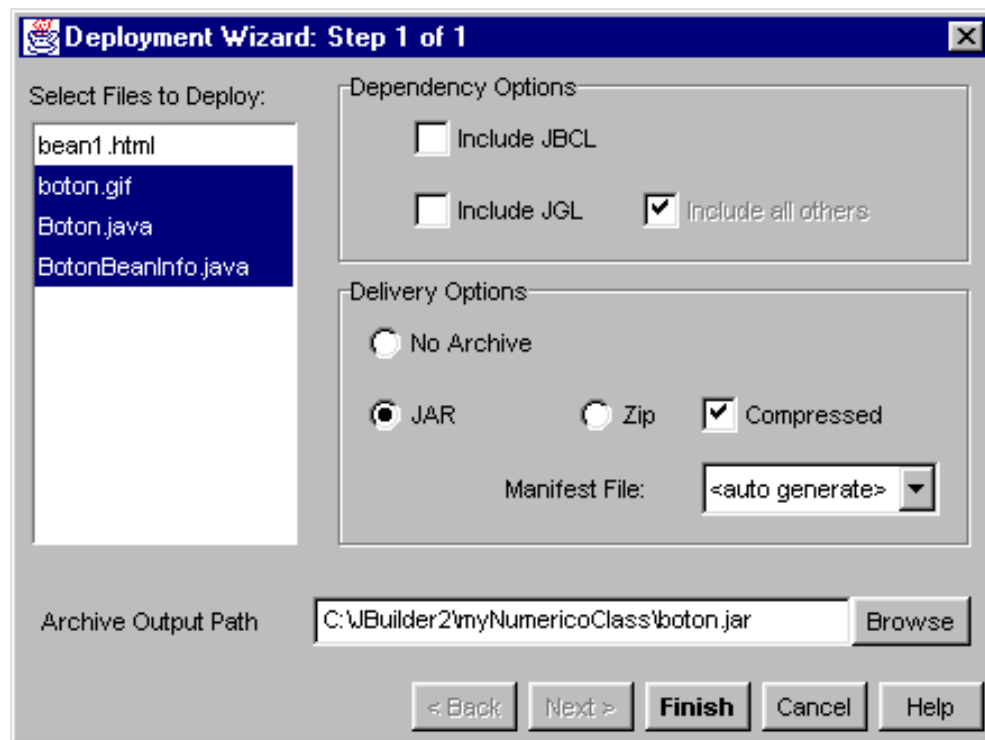
Si queremos distribuir el bean a otras personas, para que puedan usarlo, la mejor manera es empaquetarlo en un archivo JAR, similar a los archivos ZIP. De hecho se descomprimen con la misma herramienta, el programa WinZip. Un archivo JAR contiene habitualmente varias clases y archivos auxiliares.

Para crear un archivo JAR, JBuilder dispone de un asistente denominado **Deployment Wizard** al que se accede seleccionando en el menú **Wizard/Deployment Wizard**.

Seleccionamos los archivos que nos interesa comprimir, en este caso, excluimos el archivo .HTML En el campo **Archive Output Path** ponemos un nombre significativo al archivo JAR, y también podemos indicar el subdirectorio en el cual

Un botón rudimentario como bean

guardarlo. Finalmente, pulsamos en el botón **Finish**. El resto de las opciones las dejamos sin modificar.

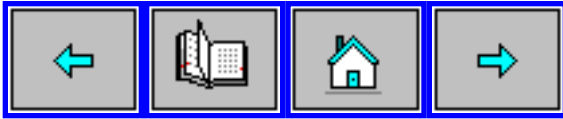


El código fuente



[boton.gif](#), [Boton.java](#), [BotonBeanInfo.java](#), [boton.jar](#)

Uso del bean



[JavaBeans](#)

[La paleta de componentes](#)

[El uso del bean](#)

[La respuesta a la acción del usuario sobre el botón](#)

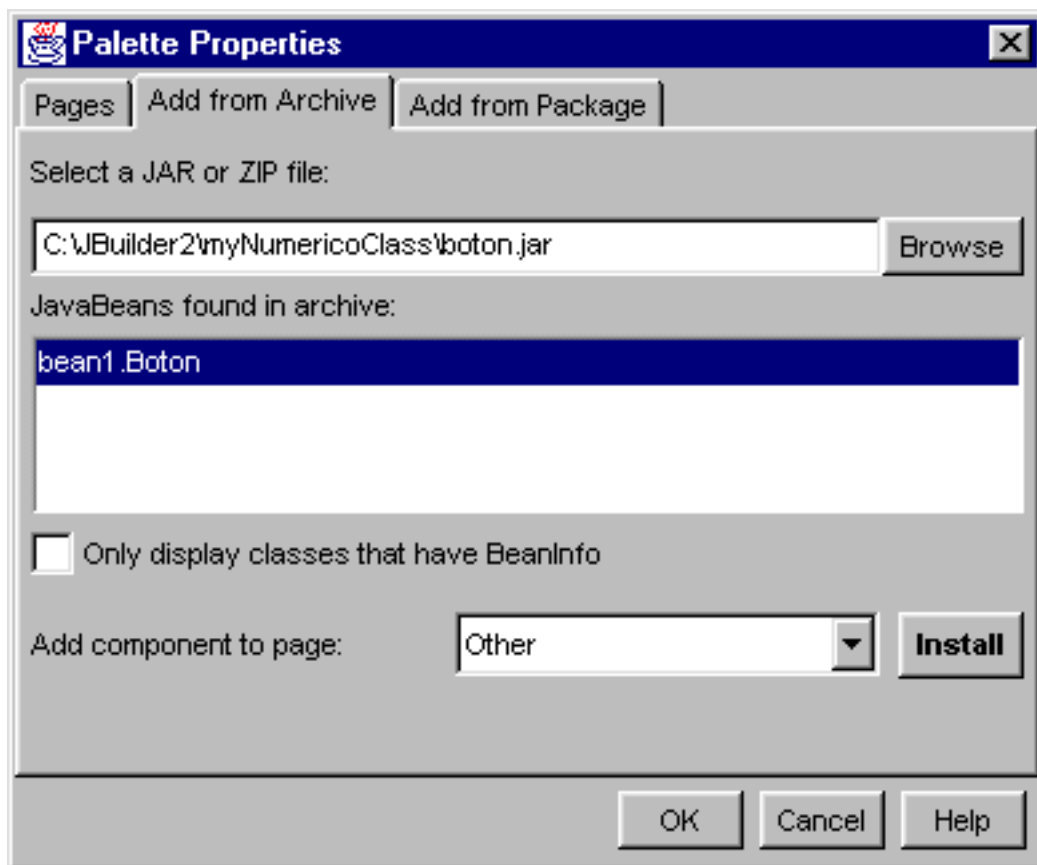
[El código fuente](#)

La paleta de componentes

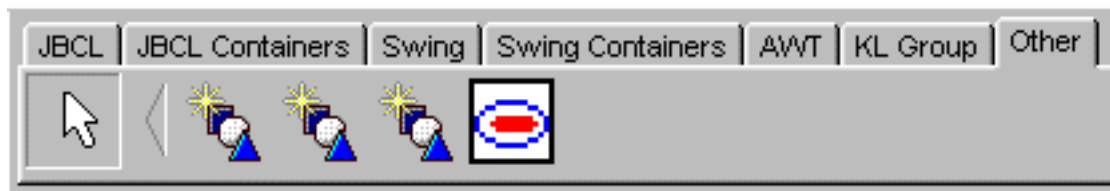
Para usar el bean lo primero que tenemos que hacer es situarlo en la paleta de componentes. La paleta de componentes tiene un conjunto de pestañas, que agrupan controles del mismo tipo o del mismo fabricante. Los controles Borland, los controles Swing para la versión Java 2.0, los controles AWT de la versión Java 1.1, paneles (containers), etc.

Situaremos el control que hemos diseñado en la hoja cuya pestaña se titula Other. Para ello, situamos el puntero del ratón en la barra de herramientas y pulsamos el botón derecho del ratón. Aparece un menú flotante con el nombre **Properties....** Lo seleccionamos y a continuación aparece el diálogo **Palette Properties**.

En la pestaña **Pages** elegimos **Other**, y a continuación pulsamos en la pestaña **Add from Archive**. Luego, pulsamos en el botón **Browse**, y vamos al subdirectorio donde hemos guardado el bean como [archivo.jar](#). Una vez cargado pulsamos el botón **Install**. Observaremos que se añade un icono a la hoja Other. Finalmente cerramos el diálogo pulsando en el botón **OK**.



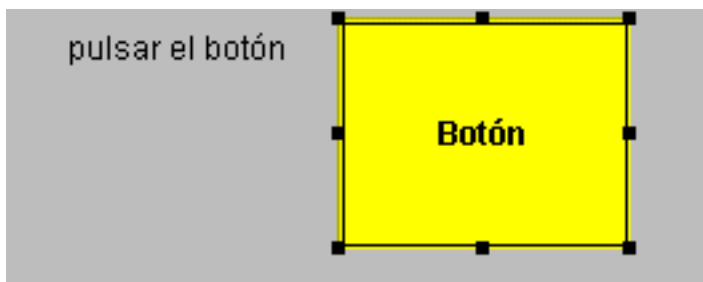
El aspecto de la paleta de componentes será el siguiente.



El nuevo componente aparece representado por un icono. Los otros tres componentes situados a la izquierda corresponden a otros beans a los que no se le ha asignado icono representativo.

El uso del bean

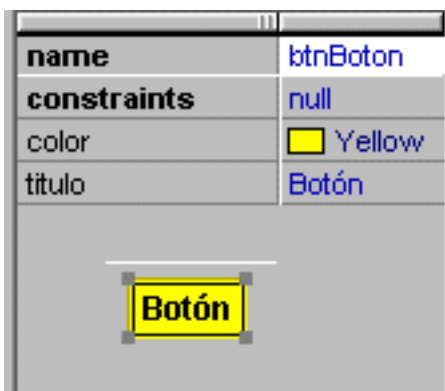
Creamos un applet para comprobar el funcionamiento de nuestro botón rudimentario. Una vez creado el esqueleto del applet con el asistente, se selecciona la pestaña diseño, **Design**. Cogemos con el ratón el componente *Boton* de la paleta *Other*, y lo depositamos en el applet. A continuación, cogemos el componente *Label* de la paleta *AWT* y lo depositamos en el applet.



Seleccionamos la etiqueta (Label) y cambiamos sus propiedades en la hoja de propiedades del componente. Cambiamos su nombre en el editor asociado a su propiedad **name**, y el texto que se muestra en el editor asociado de su propiedad **text**.

Seleccionamos el bean y aparecen sus propiedades en la hoja de propiedades del componente. En la figura inferior situada a la izquierda vemos la hoja de propiedades del componente, y debajo hemos superpuesto la imagen del bean en el momento del diseño.

Cambiamos el nombre del componente (el objeto o variable con la que se le conoce en el código fuente) en el editor **name** y le ponemos *btnBoton*. Luego, le podemos cambiar el color, pulsando en el editor de colores un pequeño botón cuyo título es ... y que da acceso a un diálogo que presenta los distintos colores. El cambio de color queda reflejado en el componente en el momento del diseño. Por último, podemos cambiar el título, en el editor **título** y el componente cambia de tamaño para acomodar al nuevo título.



| | |
|------------------|--|
| actionPerformed | |
| componentHidden | |
| componentMoved | |
| componentResized | |
| componentShown | |
| focusGained | |
| focusLost | |
| keyPressed | |
| keyReleased | |
| keyTyped | |
| mouseClicked | |
| mouseDragged | |
| mouseEntered | |
| mouseExited | |
| mouseMoved | |
| mousePressed | |
| mouseReleased | |

Cuando se cambia el título con el editor **título** de la hoja de propiedades del componente, se llama a la [función setTitulo](#), para que el bean cambie el título y modifique su tamaño. Del mismo modo, cuando se

cambia la propiedad color en el editor **color**, se llama a la función miembro [setColor](#) que pinta el botón con el nuevo color seleccionado.

El código fuente generado por el IDE es el siguiente

```
package bean1Test;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import bean1.*;

public class Bean1TestApplet extends Applet {
    bean1.Boton btnBoton = new bean1.Boton();
    Label label1 = new Label();

    public Bean1TestApplet() {
    }

    public void init() {
        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setSize(400,300);
        label1.setText("pulsar en el botón");
        btnBoton.setTitulo("Botón");
        this.add(label1, null);
        this.add(btnBoton, null);
    }
}
```

La respuesta a la acción del usuario sobre el botón

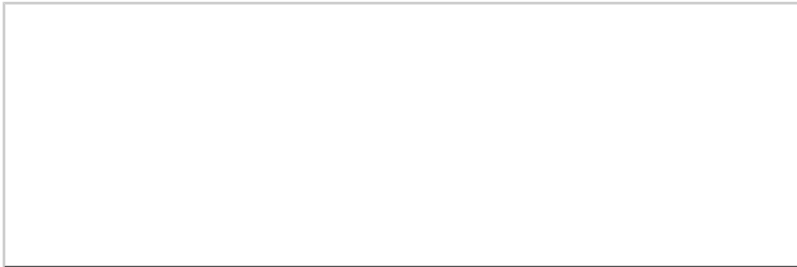
Seleccionamos nuestro botón rudimentario y vamos a la hoja de propiedades y sucesos, pulsamos en la pestaña **Events** y hacemos doble-clic en el editor correspondiente a **actionPerformed** tal como se ve a la derecha en la figura anterior. El IDE genera el código que [define la función respuesta](#) a la acción del usuario sobre el botón.

```
private void jbInit() throws Exception {  
//...  
    btnBoton.addActionListener(new java.awt.event.ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            btnBoton_actionPerformed(e);  
        }  
    });  
}  
  
void btnBoton_actionPerformed(ActionEvent e) {  
//poner aquí el código de la función respuesta  
}
```

Ahora, solamente nos queda definir la tarea que deseamos que se realice cuando se pulsa sobre el botón, es decir, escribir el código de la función respuesta.

En este caso, la tarea es muy simple, la etiqueta (label) cambia el texto, cuando se pulsa el botón.

```
void btnBoton_actionPerformed(ActionEvent e) {  
    labell.setText("botón pulsado");  
}
```



El código fuente completo de este ejemplo es el siguiente

```

package bean1Test;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import bean1.*;

public class Bean1TestApplet extends Applet {
    boolean isStandalone = false;
    bean1.Boton btnBoton = new bean1.Boton();
    Label label1 = new Label();
    FlowLayout flowLayout1 = new FlowLayout();

    public Bean1TestApplet() {
    }
    public void init() {
        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setSize(400,300);
        label1.setText("pulsar en el botón");
        this.setLayout(flowLayout1);
        btnBoton.setTitulo("Botón");
        btnBoton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btnBoton_actionPerformed(e);
            }
        });
        this.add(label1, null);
        this.add(btnBoton, null);
    }

    void btnBoton_actionPerformed(ActionEvent e) {
        label1.setText("botón pulsado");
    }
}

```

El código fuente



[Bean1TestApplet.java](#)

Un bean hecho con controles AWT



[JavaBeans](#)

[Diseño del bean](#)

[Propiedades y sucesos del bean](#)

[Relación entre el aspecto y las propiedades del bean](#)

[Preparación del bean](#)

[El uso del bean](#)

[El código fuente](#)

Una vez que hemos estudiado las características esenciales de los beans, vamos a crear un bean que tenga alguna utilidad para el programador de applets y aplicaciones.

La barra de desplazamiento (*Scrollbar*) es un componente muy útil para introducir datos, no es tan flexible como un control de edición, pero tiene la ventaja de ser accionado exclusivamente por el ratón. Su limitación está en que solamente podemos seleccionar datos enteros entre un mínimo y un máximo, y en que no nos muestra la cantidad a la que equivale la posición del dedo.

El control de edición es más flexible, ya que podemos introducir cualquier dato. Tiene la desventaja de la utilización del teclado, y que hemos de verificar los datos que se introducen. Por ejemplo, si se requiere un dato numérico, hemos de verificar que no se introducen otros caracteres, una coma en vez de un punto como separador de la parte entera y la decimal, que el dato introducido está dentro de un determinado intervalo, etc.

En el [Ejercicio 2](#) se planteaba al lector crear un applet con un control de edición y una barra de desplazamiento horizontal. Si el lector ha resuelto con éxito este ejercicio, podrá reutilizar la mayor parte del código para crear el bean. La única diferencia reside en que en el ejercicio hemos puesto los controles sobre un applet (un panel especializado), mientras que en el JavaBean los situaremos sobre un panel (objeto de la clase *Panel*)

La combinación de un control de edición y una barra de desplazamiento, es muy útil para la programación de aplicaciones en el ámbito científico. Vamos a crear un bean que tenga estos dos componentes situados sobre un panel. Además, haremos que la posición del dedo en la barra de desplazamiento no se limite a números enteros sino a números reales en general.

Diseño del bean

Creamos un nuevo proyecto seleccionando [File/New Project](#)

[Añadimos al proyecto un JavaBean](#) seleccionado **File/New** y a continuación en el diálogo **New** el icono **JavaBean**

En el diálogo que aparece titulado **JavaBean Wizard** ponemos *BarTexto* en el campo **Name of new JavaBean**, el nombre del bean, y en el campo **Base class to inherit from** seleccionamos `java.awt.Panel`.

Observamos que se genera una clase denominada *BarTexto* que deriva de la clase *Panel*.

Diseño

Seleccionamos la pestaña **Design** y situamos sobre el panel un control de edición a la izquierda y una barra de desplazamiento horizontal a la derecha. Podemos introducir otros nombres para los controles o dejar los que pone por defecto el IDE. No es necesario efectuar otros cambios significativos en las hojas de propiedades de los componentes.

```
textField1.setText("0.0");
textField1.setColumns(3);
scrollbar1.setValue(50);
scrollbar1.setOrientation(0);
```

[BorderLayout](#) es el gestor de diseño que elegimos para el panel *BarTexto*, de modo que el control de edición queda a la izquierda (WEST) y la barra de desplazamiento en el centro (CENTER).

```
this.setLayout(borderLayout1);
this.add(textField1, BorderLayout.WEST);
this.add(scrollbar1, BorderLayout.CENTER);
```

Tamaño del bean

Como ya vimos al estudiar el control botón rudimentario en la página anterior, es necesario redefinir las funciones *getPreferredSize* y *getMinimumSize*. En este caso, se ha tomado como unidad las dimensiones del carácter **0** de la fuente de texto por defecto. El ancho se ha tomado igual a 20 veces la anchura de un carácter y el alto, igual al doble de la altura de un carácter. La [clase FontMetrics](#) nos proporciona la información relativa a la fuente de texto actualmente seleccionada.

Estas dimensiones son las que tendrá el bean cuando se coge de la barra de herramientas y se deposita en el applet, luego, podemos cambiarlas actuando con el ratón en el momento de diseño. Las dimensiones finales de este componente dependerán de las dimensiones del panel que lo contiene, de los otros componentes presentes y del gestor de diseño (layout) empleado. Las dimensiones del componente no podrán ser inferiores al valor mínimo dado por la función miembro *getMinimumSize*.

```
public Dimension getPreferredSize() {
    FontMetrics fm=getFontMetrics(getFont());
    return new Dimension(20*fm.stringWidth("0"), 2*fm.getHeight());
}
public Dimension getMinimumSize() {
    return getPreferredSize();
}
```

Cada programador puede definir estas funciones como mejor le convenga.

Interacción entre los controles

Examinamos en este apartado cómo interaccionan los dos controles. Cómo la posición del dedo en la barra de desplazamiento se muestra en el control de edición, y cómo el dato que se introduce en el control de edición se refleja en la posición del dedo en la barra de desplazamiento.

- **Respuesta a las acciones sobre la barra de desplazamiento**

En primer lugar, queremos que el valor al que equivale la posición del dedo en la barra de desplazamiento se muestre en el control de edición.

En el capítulo dedicado al estudio de la respuesta a las acciones del usuario sobre los controles examinamos el [control barra de desplazamiento](#).

Para responder a las acciones de mover el dedo en la barra de desplazamiento, actuar con el ratón sobre las flechas situadas en los extremos, o en entre los extremos de la barra y el dedo, vamos a elegir la alternativa más cómoda, la que nos proporciona el IDE de JBuilder. En el modo diseño, seleccionamos el control *scrollbar1* y en la hoja de propiedades y sucesos pulsamos en la pestaña **Events**, haciendo doble-clic en el editor de *adjustmentValueChanged*. En el código que se genera se asocia el control *scrollbar1* (la fuente de sucesos) con un objeto de una [clase anónima](#) que implementa el interface *AdjustmentListener*, mediante la función miembro *addAdjustmentListener*.

```
scrollbar1.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
    public void adjustmentValueChanged(AdjustmentEvent e) {
        scrollbar1_adjustmentValueChanged(e);
    }
});
```

En la función respuesta *adjustmentValueChanged* o bien en *scrollbar1_adjustmentValueChanged*, leemos la posición del dedo suministrada por el suceso *e* de la clase *AdjustmentEvent* mediante *getValue*. [Convertimos el valor numérico a texto](#) y lo mostramos en el control de edición *textField1* mediante *setText*.

```
void scrollbar1_adjustmentValueChanged(AdjustmentEvent e) {
    int pos=e.getValue();
    textField1.setText(String.valueOf(pos));
}
```

- **Respuesta a las acciones sobre el control de edición**

Al introducir un dato en el control de edición y pulsar la tecla RETORNO o ENTER queremos que dicho dato se refleje en la posición del dedo en la barra de desplazamiento.

Cuando se pulsa RETORNO en un control de edición se genera un suceso del tipo *ActionEvent*, como en el control botón. Para responder a esta acción, nos situamos en el modo diseño, pestaña **Design**, seleccionamos el control *textField1*, y vamos a su hoja de propiedades y sucesos. Pulsamos en la pestaña **Events**, hacemos doble-clic en el editor asociado a **actionPerformed**. El IDE genera el siguiente código.

```
textField1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        textField1_actionPerformed(e);
    }
});
```

```
});
```

En el cuerpo de la función respuesta *actionPerformed* o bien *textField1_actionPerformed*, leemos el dato introducido en el control de edición mediante *getText*, [convertimos el texto en número](#), y modificamos la posición del dedo de la barra de desplazamiento *scrollbar1* mediante *setValue*.

```
void textField1_actionPerformed(ActionEvent e) {
    int pos=Integer.parseInt(textField1.getText());
    scrollbar1.setValue(pos);
}
```

Filtrado de los caracteres

En el control de edición introducimos un dato numérico, de modo que no son necesarios los caracteres no numéricos. En el estudio del [control de edición](#) vimos como podía crearse un filtro de estas características.

Cada vez que que se teclea un carácter en un control de edición se genera un suceso de la clase *KeyEvent*. Para responder a estas acciones, seleccionamos el control de edición *textField1*, y vamos a su hoja de propiedades y sucesos. Pulsamos en la pestaña **Events** y hacemos doble-clic en el editor asociado a **keyTyped**. El IDE genera el siguiente código.

```
textField1.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyTyped(KeyEvent e) {
        textField1_keyTyped(e);
    }
});
```

En la función respuesta *keyTyped* o bien en *textField1_keyTyped*, filtramos los caracteres

```
void textField1_keyTyped(KeyEvent e) {
    char c=e.getKeyChar();
    if((c>='0' && c<='9') || (c=='.')) || (c==KeyEvent.VK_DELETE)
        || (c==KeyEvent.VK_BACK_SPACE)) {
        return;
    }
    e.consume();
}
```

Solamente, aparecerán en el control de edición los caracteres numéricos del 0 al 9, el carácter punto, separador de la parte entera y decimal, y los caracteres correspondientes a las teclas de edición suprimir (*KeyEvent.VK_DELETE*) y retroceso (*KeyEvent.VK_BACK_SPACE*). El resto de los caracteres son consumidos *consume* y no llegan al control de edición.

Propiedades y sucesos del bean

Una vez concluído el diseño, la siguiente tarea a realizar en el bean, es la de determinar sus propiedades simples o ligadas y los sucesos (events) que emite o a los que ha de responder.

Persistencia

El primer paso, el más sencillo, es el de proporcionar al bean la característica de la persistencia. Seleccionamos la pestaña **Bean** y a continuación **General** y activamos la casilla [Support serialization](#).

El IDE genera el código para que la clase *BarTexto* implemente el interface *Serializable*

```
public class BarTexto extends Panel implements java.io.Serializable{
//...
}
```

Propiedades

El bean va a tener dos [propiedades simples](#) el valor máximo y el valor mínimo, y una [propiedad ligada](#), el valor indicador (al que equivale la posición del dedo en la barra de desplazamiento)

Para [definir dichas propiedades](#) seleccionamos la pestaña **Bean** y a continuación la pestaña **Properties**, y pulsamos en el botón **addProperty**, apareciendo el diálogo titulado **New Property**

Para definir la propiedad simple *maximo* ponemos

| | |
|---------------|--------|
| Property name | maximo |
| Type | double |
| Binding | none |

A continuación, pulsamos el botón **Apply**.

Para definir la propiedad *minimo* ponemos

| | |
|---------------|--------|
| Property name | minimo |
| Type | double |
| Binding | none |

A continuación, pulsamos el botón **Apply**

Para definir la propiedad ligada *indicador* ponemos

| | |
|---------------|-----------|
| Property name | indicador |
| Type | double |
| Binding | bound |

A continuación pulsamos el botón **OK** para cerrar el diálogo

El IDE genera nuevo código,

```
public class BarTexto extends Panel implements java.io.Serializable{
```

```

    private double maximo=1.0;
    private double minimo=0.0;
    private double indicador=0.0;
    private transient PropertyChangeSupport propertyChangeListeners = new
PropertyChangeSupport(this);
//...

    public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.removePropertyChangeListener(l);
    }
    public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.addPropertyChangeListener(l);
    }

    public void setMaximo(double newMaximo) {
        maximo = newMaximo;
    }
    public double getMaximo() {
        return maximo;
    }

    public void setMinimo(double newMinimo) {
        minimo = newMinimo;
    }
    public double getMinimo() {
        return minimo;
    }

    public void setIndicador(double newIndicador) {
        double oldIndicador = indicador;
        indicador = newIndicador;
        propertyChangeListeners.firePropertyChange("indicador",
            new Double(oldIndicador), new Double(newIndicador));
    }
    public double getIndicador() {
        return indicador;
    }
}

```

Sucesos (events)

El bean emite (fire) un suceso de la clase *PropertyChangeEvent* cuando cambia su [propiedad ligada](#) *indicador*, pero no responde (listen) a ningún suceso.

```

    public void setIndicador(double newIndicador) {
        double oldIndicador = indicador;
        indicador = newIndicador;
        propertyChangeListeners.firePropertyChange("indicador",
            new Double(oldIndicador), new Double(newIndicador));
    }

```

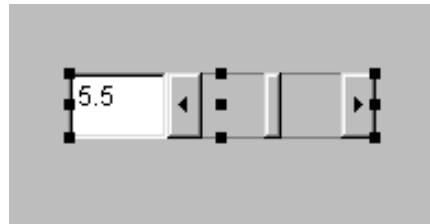
Los objetos (listeners) interesados en el cambio de esta propiedad ligada, como veremos más adelante, han de implementar el interface *PropertyChangeListener* y definir la función *propertyChange*.

Relación entre el aspecto y las propiedades del bean

Ahora queremos que cuando cambiamos el valor de las propiedades en la hoja de propiedades del componente, se refleje este cambio en el aspecto del bean. En concreto, cuando se cambia el valor de la propiedad *indicador*, este cambio debe reflejarse en la posición del dedo en la barra de desplazamiento.

En la figura inferior tenemos la hoja de propiedades a la izquierda y el aspecto del bean sobre un applet en el momento de diseño a la derecha.

| | |
|--------------------|------------------|
| name | barTexto1 |
| constraints | 184, 121, -1, -1 |
| indicador | 5.5 |
| maximo | 12.0 |
| minimo | 0.0 |



Conversión de unidades

La barra de desplazamiento tiene por defecto un valor mínimo que vale cero y un valor máximo que vale 100, las posiciones del dedo en la barra de desplazamiento son números enteros comprendidos entre 0 y 100.

Hemos de transformar el valor de la propiedad *indicador* (un número real) comprendido entre los valores de las propiedades *minimo* y *maximo* en un valor entero comprendido entre 0 y 100 que señale la posición del dedo en la barra de desplazamiento. La función *valorEntero* realiza esta transformación

```
private int valorEntero(double x){
    return (int)((100*(x-minimo))/(maximo-minimo));
}
```

A su vez, hemos de transformar la posición del dedo, un valor entero comprendido entre 0 y 100, en el valor de la propiedad *indicador* comprendido entre los valores *minimo* y *maximo*. Esta tarea la realiza la función *valorDouble*

```
private double valorDouble(int x){
    return (minimo+x*(maximo-minimo)/100);
}
```

Cambio en el valor de la propiedad indicador

- **Actualizar el bean cuando se modifican las propiedades**

Cuando se cambia el valor de la propiedad *indicador* en la hoja de propiedades o durante la ejecución del programa, se llama a la función *setIndicador* para reflejar dicho cambio. En el cuerpo de dicha función, se deberá actualizar la posición del dedo en la barra de desplazamiento y el texto en el control de edición. Por razón de seguridad, se comprobará que el nuevo valor de la propiedad *indicador* está comprendido entre los valores *minimo* y *maximo*.

```

public void setIndicador(double newIndicador) {
    if(newIndicador<minimo) newIndicador=minimo;
    if(newIndicador>maximo) newIndicador=maximo;
//...
    textField1.setText(String.valueOf(indicador));
    scrollbar1.setValue(valorEntero(indicador));
}

```

- **Actualizar la propiedad ligada cuando se introduce un dato en el control de edición**

Cuando se introduce un determinado valor numérico en el control de edición, debe quedar reflejado en el la posición del dedo en la barra de desplazamiento y también, debe actualizarse la propiedad *indicador*. Ya hemos comentado en el apartado [Interacción entre controles](#) cómo cambia la posición del dedo en la barra de desplazamiento cuando se introduce un número entero en el control de edición *textField1* y se pulsa RETORNO. Ahora, manejamos números reales, por lo que el código de la función repuesta a la acción de pulsar RETORNO en el control de edición precisa de algunos cambios.

```

void textField1_actionPerformed(ActionEvent e) {
    double pos=Double.valueOf(textField1.getText()).doubleValue();
    if(pos<minimo) pos=minimo;
    if(pos>maximo) pos=maximo;
    scrollbar1.setValue(valorEntero(pos));
    setIndicador(pos);
}

```

En el código de la función respuesta *textField1_actionPerformed*, se obtiene el texto del control de edición *textField1*, se [convierte el texto en un número del tipo *double*](#). Se verifica que dicho número está en el intervalo comprendido entre los valores de las propiedades *minimo* y *maximo*. Por último, se actualiza la posición del dedo en la barra de desplazamiento, realizando una conversión de unidades, y se actualiza la propiedad *indicador* llamado a *setIndicador*.

- **Actualizar la propiedad ligada cuando se mueve el dedo en la barra de desplazamiento**

Cuando se mueve con el ratón el dedo en la barra de desplazamiento, vimos en el apartado [Interacción entre controles](#) cómo cambia el texto en el control de edición. Ahora, trabajamos con números reales y además hemos de actualizar la propiedad ligada *indicador*.

```

void scrollbar1_adjustmentValueChanged(AdjustmentEvent e) {
    int pos=e.getValue();
    textField1.setText(String.valueOf(valorDouble(pos)));
    setIndicador(valorDouble(pos));
}

```

Se lee la posición del dedo en la barra de desplazamiento mediante *getValue*, se convierte el valor entero devuelto en un número real comprendido entre los valores de las propiedades *minimo* y *maximo*, se [convierte dicho número real en texto](#), y se muestra en el control de edición *textField1*, mediante *setText*. Por último, se actualiza la propiedad *indicador* llamado a *setIndicador*.

El código completo del bean *BarTexto* es el siguiente

```

package bean3;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class BarTexto extends Panel implements java.io.Serializable{
    private double maximo=1.0;
    private double minimo=0.0;
    private double indicador=0.0;
    TextField textField1 = new TextField();
    Scrollbar scrollbar1 = new Scrollbar();
    BorderLayout borderLayout1 = new BorderLayout();
    private transient PropertyChangeSupport propertyChangeListeners = new
PropertyChangeSupport(this);

    public BarTexto() {
        try {
            jbInit();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setSize(new Dimension(100, 20));
        textField1.setText("0.0");
        textField1.setColumns(3);
        textField1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                textField1_actionPerformed(e);
            }
        });
        textField1.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyTyped(KeyEvent e) {
                textField1_keyTyped(e);
            }
        });
        scrollbar1.setValue(50);
        scrollbar1.setOrientation(0);
        scrollbar1.addAdjustmentListener(new java.awt.event.AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                scrollbar1_adjustmentValueChanged(e);
            }
        });
        this.setLayout(borderLayout1);
        this.add(textField1, BorderLayout.WEST);
        this.add(scrollbar1, BorderLayout.CENTER);
    }

    void scrollbar1_adjustmentValueChanged(AdjustmentEvent e) {
        int pos=e.getValue();
    }

```

```

        textField1.setText(String.valueOf(valorDouble(pos)));
        setIndicador(valorDouble(pos));
    }

    void textField1_keyTyped(KeyEvent e) {
        char c=e.getKeyChar();
        if((c>='0' &&
c<='9') || (c=='.')) || (c==KeyEvent.VK_DELETE) || (c==KeyEvent.VK_BACK_SPACE)){
            return;
        }
        e.consume();
    }

    void textField1_actionPerformed(ActionEvent e) {
        double pos=Double.valueOf(textField1.getText()).doubleValue();
        if(pos<minimo) pos=minimo;
        if(pos>maximo) pos=maximo;
        scrollbar1.setValue(valorEntero(pos));
        setIndicador(pos);
    }

    private int valorEntero(double x){
        return (int)((100*(x-minimo))/(maximo-minimo));
    }

    private double valorDouble(int x){
        return (minimo+x*(maximo-minimo)/100);
    }

    public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.removePropertyChangeListener(l);
    }

    public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
        propertyChangeListeners.addPropertyChangeListener(l);
    }

    public Dimension getPreferredSize() {
        FontMetrics fm=getFontMetrics(getFont());
        return new Dimension(20*fm.stringWidth("0"), 2*fm.getHeight());
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    public void setMaximo(double newMaximo) {
        maximo = newMaximo;
    }

    public double getMaximo() {
        return maximo;
    }

    public void setMinimo(double newMinimo) {
        minimo = newMinimo;
    }

```

```

    public double getMinimo() {
        return minimo;
    }

    public void setIndicador(double newIndicador) {
        if(newIndicador<minimo) newIndicador=minimo;
        if(newIndicador>maximo) newIndicador=maximo;
        double  oldIndicador = indicador;
        indicador = newIndicador;
        textField1.setText(String.valueOf(indicador));
        scrollbar1.setValue(valorEntero(indicador));
        propertyChangeListeners.firePropertyChange("indicador", new
Double(oldIndicador), new Double(newIndicador));
    }
    public double getIndicador() {
        return indicador;
    }
}

```

Preparación del bean

Una vez completado el código fuente, se compila, y si tiene éxito se prepara para su utilización en otros programas.

BeanInfo

El primer paso es crear uno o más iconos representativos del bean. En este caso, se ha creado un icono en color de 32x32 pixels que lo guardamos en el archivo BarTexto.gif

El segundopaso, consiste en crear una clase que implemente el [interface *BeanInfo*](#). Seleccionado las pestañas **Bean** y a continuación, **BeanInfo** apareciendo un panel con información relativa al bean. En la parte inferior derecha, pulsamos sobre el botón ... del campo **32x32 color icon** y cargamos el icono que se guarda en el archivo BarTexto.gif.

Finalmente, pulsamos en el botón titulado **Generate Bean Info** y observamos como se añade un nuevo archivo al proyecto.

Deployment

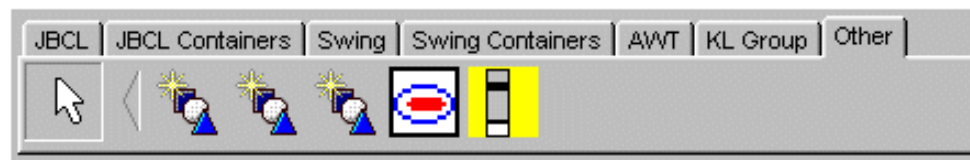
Como ya se ha explicado, la mejor forma de guardar los applets y los beans es en un [archivo .JAR](#)

Volvemos a compilar el archivo BarTexto.java y seleccionamos en el menú del IDE **Wizard/Deployment Wizard**. En el campo **Archive Output Path** del diálogo que aparece, ponemos un nombre significativo al archivo JAR, y también podemos indicar el subdirectorio en el cual guardarlo. Finalmente, pulsamos en el botón **Finish**

Insertar el bean en la barra de herramientas

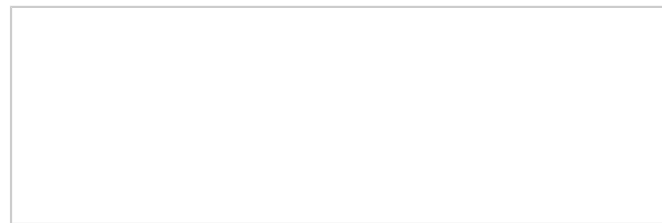
La última etapa en la creación de un bean es su [inserción en la barra de herramientas](#). Ya se ha explicado en la página anterior el proceso de instalación de un bean en la barra de herramientas, que tiene dos partes

- Cargar el bean y situarlo en la página correspondiente (normalmente Other)
- Instalar el bean pulsando el botón **Install** en el diálogo **Palette Properties**



El uso del bean

Para usar el bean *BarTexto* creamos el siguiente applet



En el momento del diseño de applet podemos cambiar las propiedades del bean *BarTexto*: *máximo*, *mínimo* e *indicador* según se muestra en la figura inferior. En los editores de la hoja de propiedades del bean hemos introducido nuevos valores para sus propiedades.

| | |
|--------------------|---------------|
| name | barTexto1 |
| constraints | 184,121,-1,-1 |
| indicador | 5.5 |
| maximo | 12.0 |
| minimo | 0.0 |

Los valores que se introducen en el control de edición o la posición del dedo en la barra de desplazamiento, van a representar los radios de una circunferencia.

En la parte inferior del applet situamos un panel con dos controles una etiqueta *label1* que contiene el texto "Area del círculo", y un control de edición *textArea* en el que se muestra el área del círculo cuyo radio hemos seleccionado en el control *barTexto1* de la clase *BarTexto*.

```
barTexto1.setMaximo(15.0);
barTexto1.setIndicador(6.0);
barTexto1.setMinimo(2.0);
```



```
label1.setText("Area de un círculo");
textArea.setColumns(10);
textArea.setText("0.0");
```

Para conectar el bean *barTexto1* con el control de edición *textArea*, hemos de definir la función respuesta al cambio de la propiedad ligada *indicador*. En el momento del diseño, pestaña **Design**, seleccionamos el bean y en la hoja de propiedades y sucesos pulsamos en la [pestaña Events](#), haciendo doble-clic en el editor de *propertyChange*, tal como se ve en la figura.

| | |
|----------------|------------------|
| mouseDragged | |
| mouseEntered | |
| mouseExited | |
| mouseMoved | |
| mousePressed | |
| mouseReleased | |
| propertyChange | 1_propertyChange |

El IDE genera el siguiente código.

```
barTexto1.addPropertyChangeListener(new java.beans.PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        barTexto1_propertyChange(e);
    }
});
```

En el cuerpo de la función respuesta *propertyChange*, o bien *barTexto1_propertyChange* definimos la tarea a realizar cuando cambia el valor de la propiedad ligada *indicador* del control *barTexto1*.

En este caso, será calcular el área del círculo. y mostrarlo en el control de edición *textArea*.

```
void barTexto1_propertyChange(PropertyChangeEvent e) {
    double radio=((Double)e.getNewValue()).doubleValue();
    double area=Math.PI*radio*radio;
    textArea.setText(String.valueOf(area));
}
```

El objeto *e* de la clase *PropertyChangeEvent*, contiene la información relativa al suceso (cambio en el valor de la propiedad ligada *indicador*). La función *getNewValue*, devuelve el nuevo valor de la propiedad ligada *indicador*. No devuelve un valor **double**, sino un objeto de la clase base *Object* que ha de ser promocionado a un objeto de la clase envolvente *Double*. Recuérdese la forma en la que se emite (fire) el suceso correspondiente al cambio en el valor de la propiedad ligada *indicador*.

```
propertyChangeListeners.firePropertyChange("indicador",
    new Double(oldIndicador), new Double(newIndicador));
```

Para convertir un objeto de la clase envolvente *Double* en un dato del tipo predefinido **double** empleamos la función *doubleValue*. Una vez que tenemos el radio, se calcula el área del círculo y se guarda en la variable local *area*. Finalmente, se convierte el [valor numérico a texto](#), para mostrarlo en el control de edición *textArea*.

El código completo de este applet es el siguiente

```

package bean3Test;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import bean3.*;
import java.beans.*;

public class Bean3TestApplet extends Applet {
    bean3.BarTextol barTextol = new bean3.BarTextol();
    Panel Panell = new Panel();
    Label labell = new Label();
    TextField textArea = new TextField();
    FlowLayout flowLayout1 = new FlowLayout();
    BorderLayout borderLayout1 = new BorderLayout();

    public Bean3TestApplet() {
    }

    public void init() {
        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setSize(new Dimension(400, 75));
        this.setLayout(borderLayout1);
        barTextol.setMaximo(15.0);
        barTextol.setIndicador(6.0);
        barTextol.setMinimo(2.0);
        barTextol.addPropertyChangeListener(new java.beans.PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent e) {
                barTextol_propertyChange(e);
            }
        });
        Panell.setLayout(flowLayout1);
        labell.setText("Area de un círculo");
        textArea.setColumns(10);
        textArea.setText("0.0");
        this.add(Panell, BorderLayout.CENTER);
        Panell.add(labell, null);
        Panell.add(textArea, null);
        this.add(barTextol, BorderLayout.NORTH);
//valor inicial
        double radio=barTextol.getIndicador();
        double area=Math.PI*radio*radio;
        textArea.setText(String.valueOf(area));
    }

    void barTextol_propertyChange(PropertyChangeEvent e) {

```

```
double radio=((Double)e.getNewValue()).doubleValue();  
double area=Math.PI*radio*radio;  
textArea.setText(String.valueOf(area));  
}  
}
```

El código fuente

 [BarTexto.gif](#), [BarTexto.java](#), [BarTextoBeanInfo.java](#), [BarTexto.jar](#)

 [Bean3TestApplet.java](#)