

Programación Java

Autor: CLEFormación S.L

Localidad y año de impresión: Madrid, 2014

Copyright: CLEFormación

Oracle Designer, Oracle Reports, PL/SQL, SQL*Plus, Oracle Enterprise Manager son marcas registradas por Oracle Corporation

Windows, Visual Basic son marcas registradas por Microsoft Corporation.

Introducción a Java

Características del lenguaje.....	1-2
Estructura de un programa Java.	1-4
Compilar y ejecutar con JDK.....	1-7
Variable de Entorno PATH	1-7
Variable de Entorno CLASSPATH	1-8
Compilación	1-8
Compilación y ejecución IDE software libre(Eclipse).....	1-10

Tipos de datos, palabras reservadas y estructuras de control de flujo

Elementos del lenguaje.....	2-2
Variables	2-2
Tipos de datos básicos.	2-2
Literales.	2-3
Tipos referencia a objeto (String)	2-6
Conversiones de tipos.	2-7
Operadores	2-9
Sentencias Condicionales	2-12
Sentencias if-else	2-12
Sentencia switch	2-13
Sentencias de Bucle.....	2-14
Bucle while.....	2-14
Bucle do-while	2-14
Bucle for.....	2-15
Control de flujo	2-16

Estructuras de Almacenamiento

Declaración de Arrays.....	3-2
Inicialización de Arrays	3-2
Arrays Multidimensionales.....	3-3

Clases

Clases	4-2
Conceptos Generales	4-2
Una clase Empleado.....	4-2
Setter and Getter	4-3
Construcción e inicialización de objetos.....	4-3
El operador "this"	4-5
Sobrecarga de métodos	4-6
Variables y métodos estáticos.....	4-6
El Garbage Collector	4-7
Modificadores de métodos y atributos.....	4-7
Trabajar con clases	4-8

Manejo de excepciones y creación de excepciones de usuario

Introducción.....	5-2
Tipos de excepciones	5-3
Mecanismo de las excepciones	5-4
Sentencias try-catch	5-4
Sentencia finally.....	5-4
La pila de llamadas.....	5-5
La regla de declarar o capturar	5-6
Creación de excepciones de usuario	5-7

Características avanzadas de clases

Herencia.....	6-2
Sobreescritura de métodos	6-4
Polimorfismo	6-5
La clase Obj	6-9
El método toString()	6-9
Clases Abstractas	6-10
Interfaces.....	6-11
Ámbito de visibilidad	6-14

Cadenas y Clases envoltantes de tipos básicos de datos

Cadenas. La Clase String	7-2
Métodos de la Clase String	7-4
Clases envoltantes de Tipos Básicos de Datos	7-7
Métodos de las clases envoltantes	7-7

Documentación

Paquetes.....	8-2
Crear paquetes de clases.....	8-2
El paquete java.lang (El Paquete de Lenguaje Java)	8-4
El paquete java.util (El Paquete de utilidades)	8-5
Documentar proyectos con javadoc	8-7
Comentarios para documentación.....	8-7
Una clase comentada.....	8-8
Convenciones de nombres.....	8-8
Un ejemplo: class Empleado	8-8
El API de java	8-12

Colecciones

Introducción.....	9-2
Interfaces.....	9-3
Collection	9-3
List	9-4
Set.....	9-5
Map.....	9-5
SortedMap	9-6
Implementaciones	9-7
ArrayList.....	9-7
LinkedList.....	9-8
HashSet	9-9
HashMap	9-11
TreeMap	9-11
Ejemplo de uso	9-13

Construcción básica de interfaces gráficos de usuario

Introducción	10-2
desarrollo con swing.....	10-3
Contenedores alto nivel	10-4
Contenedores intermedios	10-7
componentes ligeros.....	10-11
Gestión de eventos	10-26
Creación de eventos.....	10-26

Entrada y Salida estándar

Introducción a los Elementos Básicos del Api	11-1
Flujos de datos	11-1
Clases básicas flujos de bytes	11-2
InputStream	11-2
OutputStream	11-2
Clases Básicas Flujos de Caracteres.....	11-3
Reader	11-3
Writer	11-3
Flujos predefinidos	11-5
Leer Entradas en la Consola	11-5
Escribir Entradas en la Consola	11-6
Tipos de Nodos Disponibles	11-7
Leer y escribir archivos en flujos de bytes.....	11-7
Piped.....	11-8
Tipos de Filtros Disponibles	11-12
Utilización de filtros en entrada y salida	11-12
La Clase File.....	11-15
Archivos de acceso aleatorio.....	11-17

Entrada/Salida con NIO2

Introducción.....	12-1
Path	12-2
Rutas Relativas.....	12-2
Rutas Absolutas.....	12-2
Metada y Propiedades de Ficheros	12-5
Atributos básicos de un fichero	12-5
Modificar propiedades fecha de creación del fichero	12-6
Permisos de los ficheros a través de Posix.....	12-6
Leer, escribir y crear ficheros	12-7
Escribir en ficheros	12-7
Leer ficheros.....	12-7
Creación de ficheros.....	12-8
Ficheros Acceso Aleatorio	12-9

Programación Concurrente (Threads)

Introducción a los Threads.....	13-1
Los thread.....	13-1
Creación de un Threads.....	13-3
Extendiendo de la clase Thread	13-3
Implementación del interface Runnable	13-4
Control de threads.....	13-6
Parada de un Thread con sleep	13-6
Prioridades entre hilos (yield)	13-6
Enlazar threads (join).....	13-8
Sincronización de Threads.....	13-10
Utilización de synchronized	13-11
control de sincronización wait y notify	13-13

Programación en Red TCP/IP

Introducción	14-1
Dirección y puerto	14-1
Programación del protocolo TCP	14-3
modelo de comunicación	14-3
Creación de un servidor TCP	14-3
creación de un cliente TCP	14-5
Servidor multithread basado en TCP	14-7

Introducción a Java

TABLA DE CONTENIDOS

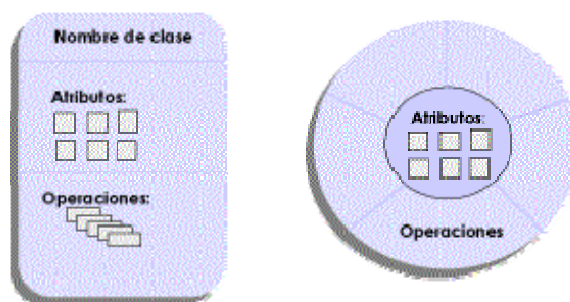
Características del lenguaje.....	2
Estructura de un programa Java.	4
Compilar y ejecutar con JDK	7
Variable de Entorno PATH.....	7
Variable de Entorno CLASSPATH	8
Compilación	8
Compilación y ejecución IDE software libre(Eclipse).....	10

Características del lenguaje

Java es un lenguaje de programación de **uso general**, es decir está preparado para realizar programas en cualquier dispositivo y plataforma: programas de consola, programas con interface gráfico, programas de internet, programas con dispositivos móviles, de mensajería, de sistemas...

Java es un lenguaje de programación totalmente **orientado a objetos**, por lo que la base del lenguaje lo constituyen las clases. De hecho, todo código escrito en Java debe estar dentro de una clase.

Una clase es un conjunto de datos y código que describe el comportamiento de los objetos de esa clase. Se puede decir que la clase es el molde a partir del que se pueden crear(o instanciar) los objetos.



Clase y Objeto

Los objetos poseen características, que son implementadas en la clase mediante datos (llamados datos miembro), estos datos no son accesibles directamente desde el exterior(encapsulamiento), para que las características del objeto puedan ser establecidas y modificadas por el programa que utiliza el objeto, la clase proporciona una serie de procedimientos (métodos) que dan acceso a los datos, de forma que estén protegidos de accesos indebidos o indeseados.

Pero una de las características más importantes de Java es su **Portabilidad**, es decir, la capacidad de ejecutar el mismo programa en distintas plataformas y en distintos sistemas operativos, característica sobre todo vital en los desarrollos en internet, donde conviven en la misma aplicación distintas plataformas y sistemas.

El compilador de Java, en lugar de generar software ejecutable (absolutamente dependiente del sistema donde se genera, como en otros lenguajes de programación), crea un código de Bytes, denominado "**Bytecodes**", que es interpretado por el entorno de ejecución de java(JRE, Java Runtime Environment), instalado en cada sistema operativo de la máquina en donde se ejecuta este software. A este entorno se le denomina, genéricamente, la "**Máquina Virtual de Java**" (JVM).

Como la mayoría de los fabricantes y sistemas operativos han apostado por Java, existe una máquina virtual para prácticamente la mayoría de sistemas.

Todo programa Java se puede ejecutar en cualquier máquina gracias a que en cada equipo se puede instalar la MÁQUINA VIRTUAL DE JAVA.

Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código de bytes, sin importar en modo alguno la máquina en que ha sido generado.

Para programar en JAVA existen diferentes entornos de programación.

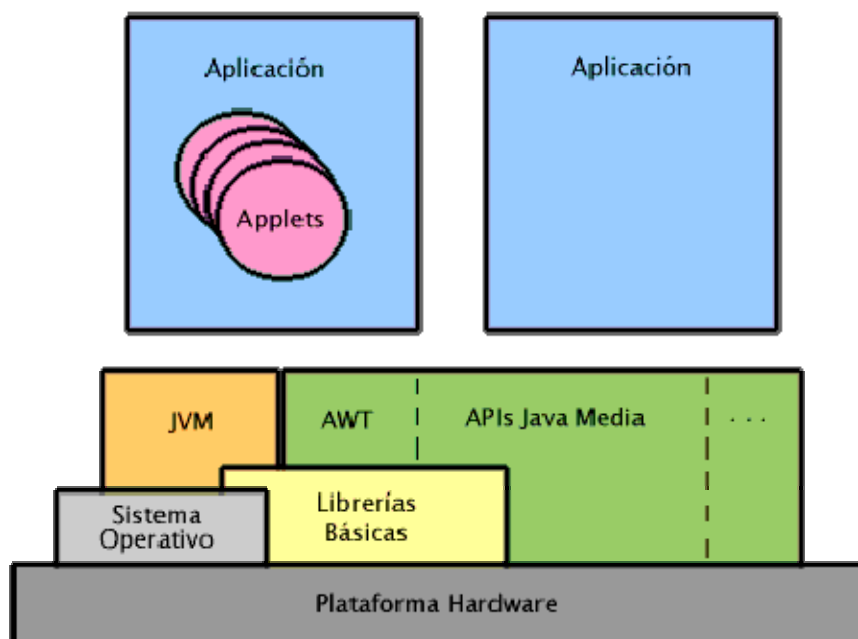
Java tiene su propio entorno de desarrollo, denominado JDK(Java Development Kit), para compilar, ejecutar, documentar, etc los programas.

Además los fabricantes de software han creado sus propios entornos de desarrollo de programas, que permite crear el código fuente de los programas, compilarlos, ejecutarlos, etc. Podemos distinguir dos tipos de entornos de desarrollo:

- Libres, como Eclipse, que será el utilizado en este curso, concretamente la versión denominada kepler, y Netbeans.
- De pago, como JDeveloper, JBuilder, y otras muchas.

La evolución de Java ha sido muy rápida. Pero como estamos viendo Java no es solo un lenguaje de programación, además está dotado de :

- Un entorno de ejecución, el JRE, y existe uno por cada sistema operativo.
- Un entorno de desarrollo, el JDK, con programas como:
 - **Javac** que compila las clases definidas
 - **Java**, que ejecuta las clases de una aplicación.
 - **Javadoc**, para documentar las clases
- Un conjunto de **librerías** organizadas en distintos directorios, que poseen la definición de clases desarrolladas por el fabricante para poder realizar programas java con muy diferentes entornos, plataformas y aplicativos: consola, red, sql, interface gráfico de usuario. Al conjunto de librerías que contienen clases de uso general se le denominan JSE, es decir, Java Standard Edition(java.lang,java.util, java.sql, java.awt...).



Estructura de un programa Java.

Todo programa en Java debe tener al menos una clase. Es común escribir cada clase en un archivo fuente. Las extensiones del fuente y ejecutable son:

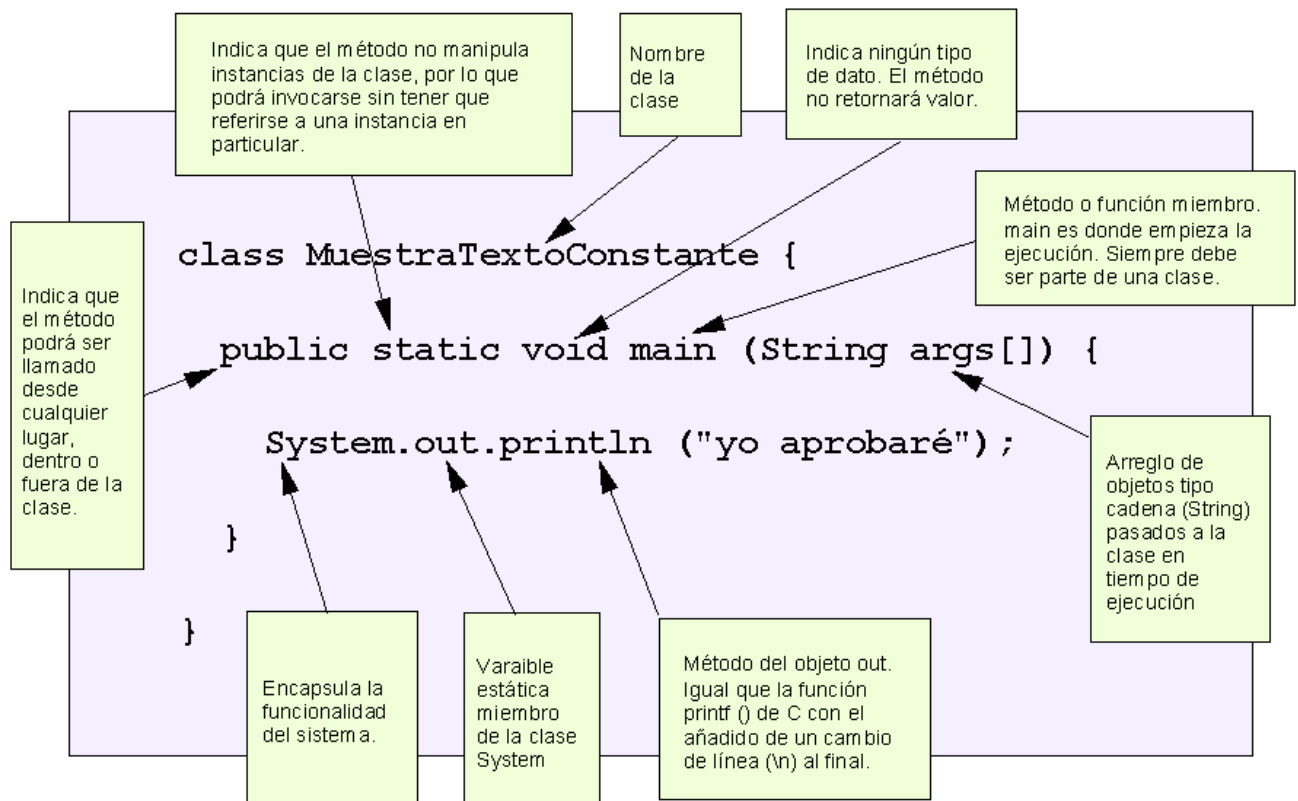
- los archivos fuente tiene extensión **.java**. y el nombre es el de la clase.
- El **bytecode** tras la compilación se llamará igual que la clase con extensión **.class**

El programa más básico que se puede escribir en Java es aquel que presenta la definición de una clase que posee un método llamado **main** que lanza un mensaje a la consola con un literal. El código fuente será el siguiente:

```
public class PrimerPrograma
{
    public static void main (String [ ] args )
    {
        System.out.println ("Bienvenido al mundo de JAVA");
    }
}
```

Toda aplicación Java, que se ejecute con la Consola del ordenador, debe contar con una clase que implemente un método llamado **main**. La Máquina Virtual ejecutará este método para iniciar el programa, representa por tanto el punto de inicio de la aplicación.

En la siguiente figura viene qué es cada uno de las partes de este pequeño código, te darás cuenta que no entiendes la mayoría de los componentes, es muy pronto. A lo largo del curso irás entendiendo todos estos términos.



EL PROGRAMA HOLAMUNDO

El programa HolaMundo muestra un saludo por pantalla(console) y su código se muestra a continuación:

```

1 public class HolaMundo
2 {
3     public static void main(String[] args){
4         System.out.println("Hola Mundo" + " bienvenido a Java");
5     }
6 }

```

LÍNEA 1

Declara la clase HolaMundo. El nombre de clase especificado se utilizará para generar el correspondiente fichero de clase, en este caso, HolaMundo.class. El contenido de la clase se delimita mediante las llaves { (línea2) y } (línea6).

LÍNEA3

Esta línea contiene el punto de entrada para la ejecución de un programa Java. Si se analiza ésta línea por partes:

- **Public:** El método main es de acceso público para todos, incluido el intérprete de Java. Más adelante veremos con detalle los mecanismos de visibilidad en Java.
- **Static:** Indica al compilador que el método main puede utilizarse dentro del contexto de la clase HolaMundo sin necesidad que exista ningún objeto.
- **Void:** Indica que el método no devuelve ningún valor.
- **String[] args :** Parámetro del método main. Normalmente invocaremos al programa sin pasarle argumentos, pero en caso contrario en él se guardan los argumentos escritos en línea de comandos al ejecutar el programa.

LÍNEA 4

Contiene el código del método main. Aquí podremos definir variables, ejecutar sentencias, etc.. En este caso usamos el mandato necesario para sacar un literal alfanumérico por la pantalla:

```
System.out.println("Hola mundo" + " bienvenido a Java");
```

De momento tienes que creerte que con la sentencia **System.out.println()** saldrá por la pantalla del ordenador, al terminar el programa, los literales y/o contenido de las variables que concatenas(símbolo +) como argumentos. Más adelante explicaremos que es cada cosa.

LÍNEAS 5 Y 6

Estas llaves marcan la finalización del método main y de la clase HolaMundo respectivamente.

Compilar y ejecutar con JDK

El JDK, versión 7, lo obtendremos a partir de la siguiente dirección:

- <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Para la configuración del JDK, una vez realizada su instalación, es necesario establecer en el sistema operativo donde se va ejecutar la variable de entorno PATH, para decir dónde están los programas binarios(ejecutables)

- javac y java,

y poder invocarlos desde cualquier directorio y la variable de entorno CLASSPATH, para informar a la JVM en qué directorio se encuentran las clases que utiliza mi programa, tanto las proporcionadas por el JDK, como las que yo incorpore a mi proyecto.

Variable de Entorno PATH

Esta variable apunta al directorio donde se encuentran los ejecutables del JDK como puede ser el compilador, interprete java, depurador, etc.

Este directorio es el directorio /bin que se encuentra en el directorio de instalación del jdk.

PATH=.;C:\WINDOWS;C:\WINDOWS\

↑ COMMAND;C:\;C:\DOS;C:\Archivos de Programas\Java\jdk1.7.0_51\bin

ESTABLECIMIENTO DE LA VARIABLE PATH EN WINDOWS

DESDE LA CONSOLA

- Teclear `set path=%path% C:\Archivos de Programas\Java\jdk1.7.0_51\bin;`
- Y se añade la dirección al path existente.

WINDOWS 8

- Arrastre el cursor a la esquina inferior derecha de la pantalla
- Haga clic en el icono de búsqueda y escriba: Panel de control
- Haga clic en -> Panel de control -> Sistema -> Opciones avanzadas
- Haga clic en Variables de entorno, en Variables del sistema, busque PATH y haga clic en él.
- En las ventanas Editar, modifique PATH agregando la ubicación de la clase al valor dePATH. Si no dispone del elemento PATH, puede optar por agregar una nueva variable y agregar PATH como el nombre y la ubicación de la clase como valor.
- Cierre la ventana.
- Vuelva a abrir la ventana del indicador de comandos y ejecute el código de java.

WINDOWS 7

- Seleccione Equipo en el menú Inicio
- Seleccione Propiedades del sistema en el menú contextual
- Haga clic en Configuración avanzada del sistema > ficha Opciones avanzadas
- Haga clic en Variables de entorno, en Variables del sistema, busque PATH y haga clic en él.

- En las ventanas Editar, modifique PATH agregando la ubicación de la clase al valor dePATH. Si no dispone del elemento PATH, puede optar por agregar una nueva variable y agregar PATH como el nombre y la ubicación de la clase como valor.
- Vuelva a abrir la ventana del indicador de comandos y ejecute el código de java.

WINDOWS XP

- Inicio -> Panel de control -> Sistema -> Opciones avanzadas
- Haga clic en Variables de entorno, en Variables del sistema, busque PATH y haga clic en él.
- En las ventanas Editar, modifique PATH agregando la ubicación de la clase al valor dePATH. Si no dispone del elemento PATH, puede optar por agregar una nueva variable y agregar PATH como el nombre y la ubicación de la clase como valor.
- Cierre la ventana.
- Vuelva a abrir la ventana del indicador de comandos y ejecute el código de java.

WINDOWS VISTA

- Haga clic con el botón derecho en el icono Mi PC
- Seleccione Propiedades en el menú contextual
- Haga clic en el separador Opciones avanzadas (enlace Configuración avanzada del sistema en Vista)
- En las ventanas Editar, modifique PATH agregando la ubicación de la clase al valor dePATH. Si no dispone del elemento PATH, puede optar por agregar una nueva variable y agregar PATH como el nombre y la ubicación de la clase como valor.
- Vuelva a abrir la ventana del indicador de comandos y ejecute el código de java.

Variable de Entorno CLASSPATH

Este variable de entorno es de uso obligatorio en entornos jdk1.x.x y apunta al directorio donde se localizan las clases java.

En este ejemplo se muestra la definicion de la variable de entorno classpath típica con un entorno jdk 1.x.x :

- El directorio donde están las clases que yo genere
- El directorio donde se encuentran las clases propias del JDK
 - El fichero src.zip . C:\Archivos de Programa\java\jd **jdk1.7.0_51**
 - y el fichero tools.jar C:\Archivos de Programa\java\ **jdk1.7.0_51**\lib

Compilación

Una vez creado el fichero HolaMundo.java con el código fuente del ejemplo, se tiene que utilizar la herramienta del JDK **javac** para realizar la compilación:

```
javac dir\HolaMundo.java
```

Si la compilación ha sido correcta, en el mismo directorio que el fichero fuente se encuentra HolaMundo.class.

Para poder ejecutar el programa deberemos situarnos en el directorio padre de cap1 y desde allí ejecutar:

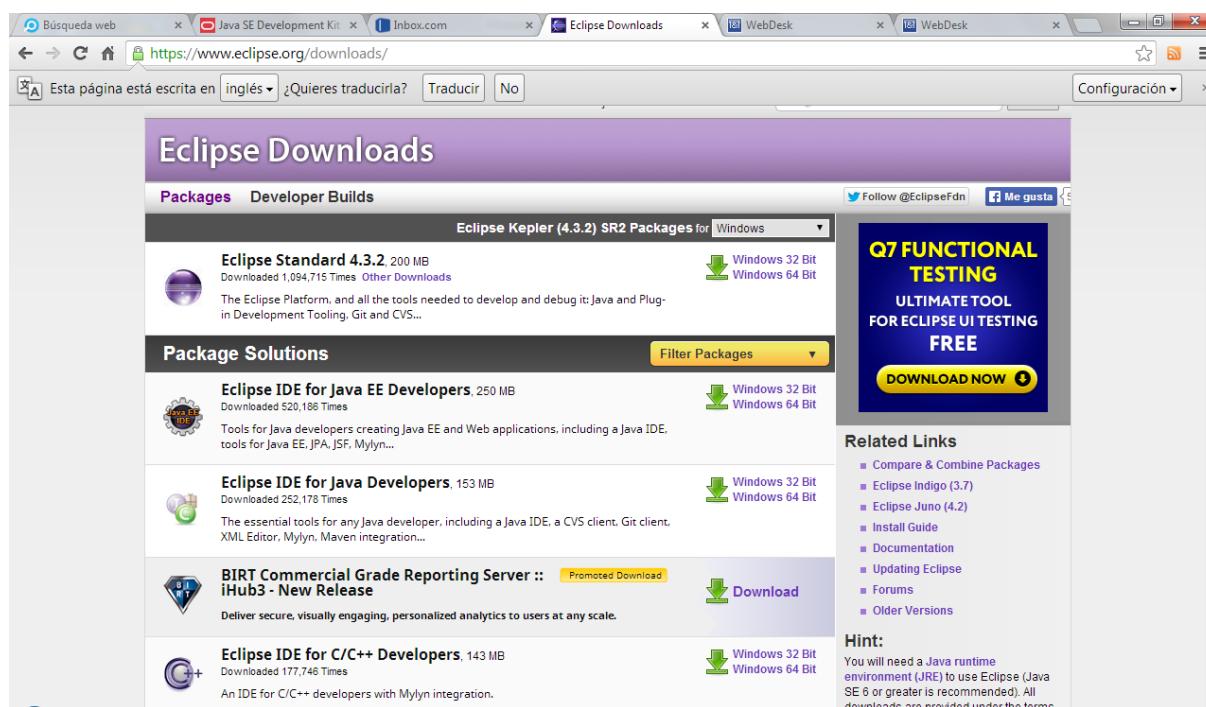
```
Java HolaMundo
```

Compilación y ejecución IDE software libre(Eclipse)

Pero lo normal será emplear un IDE, o entorno de desarrollo para crear y ejecutar nuestros programas. En este curso emplearemos eclipse, que es software libre, y lo podemos instalar fácilmente en nuestros ordenadores.

Lo instalaremos desde la página web:

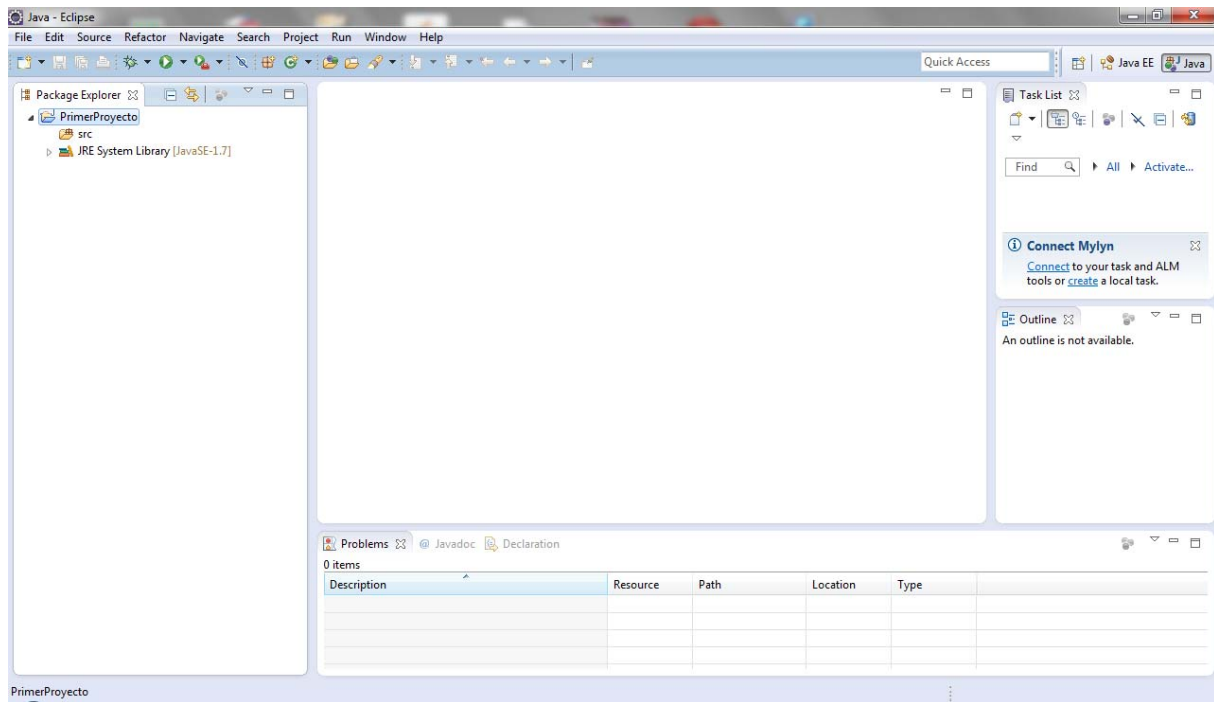
- <https://www.eclipse.org/downloads/>



Elegiremos la opción Eclipse IDE java EE Developers, de 32 o 64 Bit, dependiendo de la máquina que poseas.

Una vez instalado, dispondremos de la plataforma de desarrollo "eclipse", software totalmente libre, que nos servirá a lo largo del curso para escribir y ejecutar nuestras aplicaciones Java.

Cuando arrancamos eclipse escogemos el directorio de trabajo donde vamos a trabajar. Creamos un proyecto(en este caso llamado PrimerProyecto), le asignamos (ya veremos cómo) la versión de JRE con la que vamos a trabajar(en nuestro caso la 1.7), y eclipse nos proporciona un sistema de directorios para dejar las clases fuente(.java) y los bytecodes (.class) tras compilar cada clase.



Tipos de datos, palabras reservadas y estructuras de control de flujo

TABLA DE CONTENIDOS

Elementos del lenguaje.	2
Variables	2
Tipos de datos básicos.	2
Literales.	3
Números Enteros	4
Números Reales con Coma Flotante	5
Valores de tipo Carácter	6
Tipos referencia a objeto (String)	6
Conversiones de tipos.	7
Conversiones automáticas	7
Conversiones explícitas. (CASTING)	7
Promoción de tipos implícita.	8
Operadores	9
Sentencias Condicionales	12
Sentencias if-else	12
Sentencia switch	13
Sentencias de Bucle	14
Bucle while	14
Bucle do-while	14
Bucle for	15
Control de flujo	16

Elementos del lenguaje.

Como todos los lenguajes de programación, Java cuenta con una serie de elementos y normas para construir los programas: VARIABLES, OPERADORES, ESTRUCTURAS DE CONTROL, etc.

Variables

Una variable es una unidad básica de almacenamiento, en la que se puede almacenar un tipo definido por Java o un objeto de una clase.

En Java las variables deben ser declaradas antes de utilizarlas:

```
tipo          nombre_variable [= valor] ;
```

ejemplos:

```
int  A;
int  B= 8;
int  K, C, V=2, J=7;
boolean F = true;
char carácter = '@';
double D = 3.25;
String nombre;
String apellido = "Pérez";
```

valor puede ser una constante o una expresión. Las variables locales deben ser siempre inicializadas explícitamente.

Una variable solo puede ser utilizada dentro del bloque en el que ha sido declarada.

Una variable no puede tener el mismo nombre que otra de ámbito exterior.

Los tipos de datos pueden ser:

- Tipos básicos de datos o variables primitivas, para almacenar números enteros, números reales, booleanos y caracteres.
- Tipos de referencia, que almacenan la dirección o referencia que apunta a un objeto concreto de una clase. En este apartado veremos referencias a cadenas de caracteres, cuya clase se denomina String, aunque su estudio exhaustivo se realizará en la Unidad 7.
- Arrays, es decir el almacenamiento de n elementos de un solo tipo consecutivo. Su estudio se deja para la Unidad 3.

Tipos de datos básicos.

Se pueden dividir en cuatro grupos:

- Números Enteros:
 - byte (8 bits), del -128 al +127
 - short (16 bits), de -32K a +32K
 - int (32 bits) y de -2GB a +2GB
 - long (64 bits). De -9EB a 9EB

- Números en Coma flotante:
 - float (32 bits) y
 - double (64 bits).
- Caracteres: char (16 bit), representa caracteres unicode.
- Lógicos: Boolean (1 bit), admiten los valores true y false. No son compatibles con los numéricos.

TIPO DATO	BIT	VALOR	STANDARD
boolean	8	true o false	
Char	16	'\u0000' to '\uFFFF'	(ISO Unicode character set)
byte	8	-128 a +127	
short	16	-32.768 a +32.767	
int	32	-2.147.483.684 a +2.147.483.647	
long	64	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807	
float	32	-3,40292347E+38 a +3,40292347E+38	(IEEE 754 floating point)
double	64	-1,79769313486231570E+308 a +1,79769313486231570E+308	(IEEE 754 floating point)

Literales.

Un literal es un valor constante. Java sigue las siguientes normas respecto a los literales:

- Un literal entero es considerado como int, aunque, no existe ningún problema si este se asigna a un byte o un short si está dentro de su rango.
- Los literales en coma flotante son considerados como double, si queremos que sea considerado como float hay que añadir la letra f al final de la constante, pues no se puede asignar directamente el literal double a una variable tipo float (aunque esté dentro del rango):
 - float f = 3.14f;
 - double d = 3.14 ;
- Los literales booleanos son true y false, no pueden convertirse en ninguna representación numérica.

- Los literales tipo carácter se representan entre comilla simple(' '). Y los caracteres especiales están representados por unas secuencias, llamadas secuencias de escape. Son:

- `\n` salto a nueva línea,
- `\t` tab tabulador
- `\b` backspace borra el carácter anterior,
- `\f` formfeed
- `\\` es el carácter backslash
- `'` comilla
- `"` comillas
- `\0` nulo
- `\ddd` es un char que emplea el valor octal (d son cifras octales, por ejemplo `\329`, tienen que ser de tres o menos cifras y menores de `\377`), se puede dar también la representación hexadecimal, siempre de cuatro cifras. Aquí ponemos en orden los caracteres citados antes con su número hexadecimal: `\u000A`, `\u0009`, `\u0008`, `\u000D`, `\u000C`, `\u005C`, `\u0027` y `\u0022`.

Las cadenas se escriben entre comillas, como por ejemplo "Pietro" y todas las secuencias de escape válidas se pueden introducir en una larga cadena, creando cosas interesantes como, por ejemplo:

```
System.out.println("\tNombre:\tTomás\n\tApellido:\tSánchez");
// dará este resultado:
// Nombre: Tomás
// Apellido: Sánchez
```

Números Enteros

En Java existen varios tipos de números enteros distinguiéndose por el tamaño ocupado en memoria y por tanto el rango de valores que pueden contener.

Ejemplos:

```
int    var_a;
short  var_b=20;
long   var_c, var_d;
byte   var_e;
var_a = 015;           //Valor en octal
var_c = var_b * var_a;
var_d = 15L;
var_e = 0xDFA1;        //Valor en hexadecimal
```

Notar que al declarar un número de tipo long se le añade el sufijo L no importando si este está en mayúscula o minúscula.

Números Reales con Coma Flotante

Una constante es real o de punto flotante si:

- Incluye un punto decimal.
- Tiene una parte exponencial (letra E o e).
- Va seguida del molde 'double' (letra D o d).
- Va seguida del molde 'float' (letra F o f).

Obsérvese su uso:

```
5.0  
7E10  
35F  
2.6D
```

El rango de los tipos reales se observa en la siguiente tabla:

Tipo de dato	Longitud	Inicialización
float	32	0.0F
double	64	0.0D

El formato de un número real en Java sigue la norma IEEE 754.

Los moldes D (o d) y F (o f) permiten indicar si un número real es de tipo 'double' o 'float'. Como ya se ha dicho, si no se indica nada es de tipo doble.

Valores Booleanos

Toman valores de tipo booleano.

Ejemplos:

```
boolean    existe = false;
boolean    crear_otro;
crear_otro = true;
```

Valores de tipo Carácter

Toman valores de tipo carácter (char). Su tamaño en memoria es de 16 Bits número de bits con los que se representan los caracteres unicode. Los valores que van de 0x0000h a 0x00FFh representan los caracteres ASCII.

Ejemplos:

```
char        caracter1 = 'a';
char        carácter2;
carácter2 = 'P';
```

Tipos referencia a objeto (String)

Para referenciar a cadenas de caracteres, necesitaríamos un array de char, pero como esta formación es compleja, y en muchísimas ocasiones nos es necesario utilizar variables de este tipo, Java proporciona una clase, para referenciar a objetos que contengan una cadena de caracteres, la Clase String.

Como su uso es muy frecuente, Java nos proporciona (solo para esta clase) una forma abreviada para la creación y asignación de un valor a la variable creada:

```
String nombre;
nombre = "pepe";
String apellido = "López";
```

Aunque la manera más correcta sería:

```
String nombre;
nombre = new String("pepe");
String apellido = new String("López");
```

Cuando veamos la Unidad de las Clases, verás que el operador new es el que se usa para instanciar un objeto de cualquier clase.

Aunque ahora te parezca que las dos formas de crear objetos de la clase String tiene el mismo efecto, comprobarás más adelante que esto no es así.

Por supuesto la clase String tiene un conjunto de métodos para trabajar con el contenido asignado al objeto, y se estudiará en la Unidad 7.

Conversiones de tipos.

Cuando los tipos de datos son compatibles y el tipo de destino es más grande que el tipo de origen, Java realiza una conversión automática.

Conversiones automáticas.

Java realizará de forma automática las siguientes conversiones:

- byte a cualquier numérico,
- short a todos los numéricos excepto byte,
- int a todos los numéricos excepto byte y short,
- long a float y double,
- float a double,
- char a todos los numéricos excepto byte y short.

Conversiones explícitas. (CASTING).

Cualquier conversión entre tipos numéricos y char es posible mediante la realización de un estrechamiento(casting), esto se realiza utilizando el formato:

```
tipo    var_tipo_destino = (tipo_destino)  var_tipo_origen;
```

Ejemplos:

```
public class Programal {  
  
    public static void main(String[] args) {  
        byte b=127;  
        short s = 32000;  
        int i = 12000;  
        long l = 12000;  
        float f = 12;  
        double d = 12.5;  
        char c = 'a';  
        boolean bul = true;  
        s=i; //error no puede convertir de entero a short  
        f=d; // error no puede convertir de double a float  
        i=l; // error no puede convertir de long a entero  
        s=(short)i; //correcto convertimos el entero a short  
        f=(float)d; //correcto convertimos el double a float  
        i=(int)l; // correcto convertimos el long a entero  
  
    }  
}
```

El estrechamiento o casting puede provocar en algunos casos una pérdida de datos.

Promoción de tipos implícita.

Además de la conversión de tipos, Java define las siguientes reglas de promoción de tipos en operaciones aritméticas:

- Los tipos `byte` y `short` se promocionan a `int`.
- Si algún operando es `long`, la expresión completa se promociona a `long`.
- Si algún operando es `float`, la expresión se promociona a `float`.
- Si algún operando es `double`, la expresión se promociona a `double`.

Operadores

- Operadores Aritméticos

DESCRIPCION	SÍMBOLO	EXPRESION DE EJEMPLO	RESULTADO
Multiplicación	*	2*4	8
División	/	8/2	4
Resto división entera	%	5%2	1
Suma	+	2+2	4
Resta	-	7-3	4
Incremento	++	2++	3
Decremento	--	5--	4
Asignación más incremento	+=, -=, *=, /=, %=	5*=4 (equivale a 5= 5*4)	20
		5+=4 (equivale a 5=5+4)	9

Los operadores ++ y -- pueden aparecer delante o después de la variable sobre la que se aplican. Si se colocan delante, la variable se incrementará o decrementará antes de ser utilizada en la expresión.

```
int i1=2, j=3;
System.out.println("postincremento : " + i1++);
System.out.println("preincremento : " + ++j);

//Resultados
postincremento : 2 --- primero sale el contenido de la variable y luego se
suma uno a i1.
preincremento : 4 primero se suma a j y luego sale la variable.
```


- Operadores Lógicos

DESCRIPCION	SÍMBOLO	EXPRESION DE EJEMPLO	RESULTADO EJEMPLO
Igualdad	==	2==2	true
Desigualdad	!=	2!=2	false
Menor de	<	5<2	false
Mayor de	>	5>2	true
Menor igual de	<=	2<=2	true
Mayor igual de	>=	1>=2	false

- Operadores de Relación

DESCRIPCION	SÍMBOLO	EXPRESION DE EJEMPLO	RESULTADO EJEMPLO
Negación	!	!(2==2)	false
Y lógico	&&	(2==2) &&(2>=0)	True
O lógico		(2==2) (2!=2)	true

- Separadores y comentarios

SEPARADOR	DESCRIPCION
()	Contiene listas de parámetros, tanto en la definición de un método como en la llamada al mismo. También para realizar casting de tipos de datos.
{}	Para definir bloques de código, definir ámbitos
[]	Para declarar arrays, y referenciar valores dentro de los mismos
;	Separar sentencias
,	Separar identificadores consecutivos en la definición de variables y listas de parámetros. También para encadenar sentencias en una sentencia for.
.	Separa un nombre de método de una variable de referencia. También separa nombres en la definición de un paquete.
//	Comentario de una línea

/* */	Comentario de un bloque de líneas
exp1 ? exp2 : exp3	? : equivale a una operación de control if ... else
instanceof	Comprueba si una variable objeto contiene una instancia de la clase especificada
new	Para instanciar objetos. También para inicializar un array

Sentencias Condicionales

Las sentencias condicionales permiten la ejecución de partes del código de forma selectiva mediante expresiones condicionales.

Para todas las sentencias de control, cuando las declaraciones asociadas a las sentencias y clausulas, son más de una siempre se encierran entre doble llave : {}.

Si solo hay una declaración no es necesario.

Sentencias if-else

En las sentencias if-else se indica una expresión que siempre se evalúa como boolean. No son válidas expresiones que devuelvan enteros como en C/C++.

La sintaxis básica es la siguiente:

```
if(condicion_booleana_1){
    ....
}else if(condicion_booleana_2){
    ....
}else{
    ....
}
```

También se pueden anidar sentencias if-else:

```
if(condicion_booleana_1){
    ....
}else if(condicion_booleana_2){
    ....
    if(condicion_booleana_3){
        ....
    }else{
        ....
    }
}else{
    ....
}
```

Obsérvese el siguiente ejemplo:

```
public class PruebaIf{
    public static void main(String[] args){
        int opcion = 1;

        if(opcion == 0){
            System.out.println("Opcion 0");
        }else if(opcion == 1){
            System.out.println("Opcion 1");
        }else if(opcion == 2){
            System.out.println("Opcion 2");
        }else{
            System.out.println("Opcion Incorrecta");
        }
    }
}
```

Sentencia switch

La sentencia switch representa una alternativa múltiple,

La sintaxis es:

```
switch(expresion ){
    case expr_2: ....
                    break;
    case expr_3: ....
                    break;
    case expr_4: ....
                    break;
    default:      ....
                    break;
}
```

‘expresion’ tiene que ser compatible con el tipo int, y desde la versión 7 de Java también se puede hacer switch de una cadena de caracteres. Así, solo serán válidas expresiones de tipo byte, short, int , char y String.

El nivel de default permite tratar esas opciones no contempladas. Si la sentencia break no está presente, se ejecutan las distintas opciones en cascada, sin comprobar el valor del siguiente ‘case’.

Obsérvese el ejemplo:

```
public class PruebaSwitch{
    public static void main(String[] args){
        int opcion = 1;

        switch(opcion){
            case 0: System.out.println("Opcion 0");
                    break;
            case 1: System.out.println("Opcion 1");
                    break;
            case 2: System.out.println("Opcion 2");
                    break;
            default: System.out.println("Opcion Incorrecta");
        }
    }
}
```

La cláusula **break**; tiene la misión de convertir la sentencia switch en una alternativa múltiple, es decir aquella en la que a la primer valor encontrado de entre todas las ramas, se ejecuten las declaraciones asociadas, y el switch termine.

Si no existe la palabra break, se ejecutan todas las sentencias en secuencia a partir del primer valor encontrado, aunque estén asociadas a otras ramas del switch.

Sentencias de Bucle

Las sentencias de bucle permiten la ejecución repetida de bloques de código.

Bucle while

La sintaxis de 'while' ejecuta un bucle de 0 a n veces. Su estructura es:

```
while(expr_booleana_no_salida){  
    ....  
}
```

Es decir primero pregunta si se cumple la condición, y luego ejecuta las sentencia que hay dentro (mientras se cumple la condición).

Ejemplo:

```
int i = -1;  
while(++i<10){  
    System.out.println("i: " + i);  
}  
Resultado:  
i: 0  
i: 1  
i: 2  
i: 3  
i: 4  
i: 5  
i: 6  
i: 7  
i: 8  
i: 9
```

Bucle do-while

A diferencia de 'for' y 'while', 'do-while' asegura al menos una ejecución del bloque de código, ya que comprueba la condición al final del mismo. Su sintaxis es:

```
do{  
    ....  
}while(expr_booleana_no_salida);
```

Ejemplo:

```
int i = 0;  
do{  
    System.out.println("i: " + i);  
}while(++i<10);  
El resultado es el mismo:  
i: 0  
i: 1  
i: 2  
i: 3  
i: 4  
i: 5  
i: 6  
i: 7  
i: 8  
i: 9
```

Bucle for

La sintaxis de 'for' es:

```
for(expr_inicial;expr_booleana_no_salida;expr_incremento){  
    ....  
}
```

Ejemplo:

```
for(int i=0; i<10; i++){  
    System.out.println("i: " + i);  
}  
El resultado es el mismo:  
i: 0  
i: 1  
i: 2  
i: 3  
i: 4  
i: 5  
i: 6  
i: 7  
i: 8  
i: 9
```

En este caso la variable i es local al bucle for.

Control de flujo

Existen dos sentencias que permiten el control de flujo en los bucles:

- break [etiqueta];
- continue [etiqueta];

La sentencia break se usa para salir de bucles, switch o bloques de código. La sentencia continue permite saltar a la siguiente iteración de un bucle.

```
package cap3;

public class PruebaControl{
    public static void main(String[] args){
        boolean valor = true;

        bucle_1: while(valor){
            System.out.println("En el bucle while...");
            for(int i=0; i<5; i++){
                System.out.println("En el bucle for...");
                break bucle_1;
            }
        }
        bucle_2: for(int i=0; i<2; i++){
            System.out.println("i: " + i);
            for(int j=0; j<10; j++){
                System.out.println("j: " + j);
                continue bucle_2;
            }
        }
    }
}

La salida del ejemplo es:
En el bucle while...
En el bucle for...
i: 0
j: 0
i: 1
j: 0
```

Estructuras de almacenamiento

TABLA DE CONTENIDOS

Declaración de Arrays	2
Inicialización de Arrays	2
Arrays Multidimensionales	3

Declaración de Arrays

Los arrays permiten agrupar varios datos u objetos del mismo tipo o clase bajo un mismo nombre. Por tanto hablaremos de arrays de enteros, array de boolean, array de double, array de String , etc...

En Java, un array es un objeto incluso cuando es un array de tipos primitivos. Al ser un objeto, su declaración implica solo una reserva de memoria para la referencia. Nunca se especifica el tamaño del array en su declaración. Existen dos formas distintas de declarar arrays:

```
int i[]; // Estilo C/C++
int[] i; // Estilo alternativo
```

Con el estilo alternativo, Java permite mantener agrupado el tipo y nombre de la variable.

Hay una diferencia entre estas dos definiciones:

- en el primer caso los corchetes afectan sólo a la variable.
- En el segundo al tipo.
 - Por ejemplo, si tenemos “int a[], b;” a es un array de enteros y b es una variable int.
 - Sin embargo, si tenemos “int[] a, b;” a y b son array de enteros.

Inicialización de Arrays

Para crear arrays se utiliza la palabra reservada ‘new’, (como en la creación de cualquier otro objeto). En este momento se especifica el tamaño del array y se realiza la reserva de memoria.

```
int[] i = new int[10];
Empleado2[] e = new Empleado2[10];
String [] nombres = new String[10];
```

El primer array permite almacenar 10 enteros. Cada una de las posiciones del array se inicializa a los valores por defecto, es decir, 0. En los otros casos, el array de Empleado2 y de String, se trata de objetos. Se realiza una reserva de memoria para mantener diez referencias, pero no se crean los objetos. Cada uno de estos objetos tendrán que crearse de forma manual y hasta entonces, las referencias tendrán un valor nulo.

Todos los arrays se indexan desde 0 hasta longitud -1. Cualquier acceso fuera de rango generará una excepción.

Java permite una manera de inicializar arrays en el momento de su declaración:

```
int[] i = {1,2,3,4,5,6,7,8,9,10};  
String[] s = {"Fernando","Carmen","Luis","Elena"};
```

Vemos un ejemplo para la carga en un array de 5 elementos de los números comprendidos entre el 0 y el 5. Y sacamos por pantalla el cuadrado de cada número almacenado.

```
public class Unidimensional {  
  
    public static void main(String[] args) {  
        int [] tabla1;  
        tabla1 = new int[5];  
        for (int i=0;i<5;i++)  
            tabla1[i] = i*i;  
        for (int i = 0;i<5;i++)  
            System.out.println("el cuadrado de " + i +  
                               "es: " + tabla1[i]);  
    }  
}  
el cuadrado de 0 es: 0  
el cuadrado de 1 es: 1  
el cuadrado de 2 es: 4  
el cuadrado de 3 es: 9  
el cuadrado de 4 es: 16
```

Fíjate en la los parámetros del método main. Ahora entiendes que es String[] args.

Arrays Multidimensionales

Java permite la creación de arrays de arrays(y arrays de arrays de arrays, etc..). Véase el siguiente fragmento de código:

```
int[][] dos = new int [2][2];  
dos[0][0] = 1;  
dos[0][1] = 2;  
dos[1][0] = 3;  
dos[1][1] = 4;
```

También existe la posibilidad de crear arrays multidimensionales irregulares:

```
package cap4;  
  
public class Arrays{  
  
    public static void main(String [] args){  
  
        int[][] v = new int[3][]; //el primer índice tiene 3 elementos  
        v[0] = new int[3]; //el elemento 1 tiene tres subelementos  
        v[0][0] = 11;  
        v[0][1] = 12;  
        v[0][2] = 13;  
  
        v[1] = new int[4]; // el segundo elemento tiene 4 subelementos  
        v[1][0] = 21;  
        v[1][1] = 22;
```

```

v[1][2] = 23;
v[1][3] = 24;
v[2] = new int[5]; // el tercer elemento tiene 5 subelementos
v[2][0] = 31;
v[2][1] = 32;
v[2][2] = 33;
v[2][3] = 34;
v[2][4] = 35;

/* tambien válido así
int v[][] = { {11,12,13}, {21,22,23,24}, {31,32,33,34,35}};
*/
// Lectura de una array bidimensional con "dos for" anidados
for(int i=0;i<v.length;i++){
    for(int j=0;j<v[i].length;j++){
        System.out.print(v[i][j] + " ");
    }
    System.out.println();
}
}
}
Resulataados
11 12 13
21 22 23 24
31 32 33 34 35

```

En el ejemplo se usa la propiedad `length` de los arrays, que permite conocer la longitud de un array.

En el siguiente ejemplo se imprimen en pantalla los argumentos recibidos como parámetro en el método `main`:

```

package cap4;

public class ArgumentosMain{
    public static void main(String[] args){
        if(args!=null && args.length>=2){
            System.out.println("Los argumentos son: "+args[0]+" y "+args[1]);
        }else{
            System.out.println("No hay suficientes parámetros");
            System.out.println("Ej: java cap4.ArgumentosMain Pepe 5");
        }
    }
}

```

4

Clases

TABLA DE CONTENIDOS

Clases	2
Conceptos Generales	2
Una clase Empleado.....	2
Setter and Getter	3
Construcción e inicialización de objetos.....	3
Constructores	4
El operador "this"	5
Sobrecarga de métodos.....	6
Variables y métodos estáticos	6
El Garbage Collector.....	7
Modificadores de métodos y atributos.....	7
Trabajar con clases.....	8

Clases

Conceptos Generales

Un objeto es una colección de datos y métodos que trabajan con los datos. Al tipo de datos de un objeto se le llama clase. A un objeto también se le conoce como una instancia de su clase. En Java, un objeto se crea usando el operador 'new', que invoca a un constructor de la clase para inicializar el objeto.

Una clase Empleado

```
public class Empleado{
    private int dias=30;
    private String nombre;
    private double sueldo;
    //Setter & Getter
    public String getNombre(){
        return nombre;
    }
    public double getSueldo(){
        return sueldo;
    }
    public int getDiasVacaciones(){
        return dias;
    }
    public void setNombre(String n){
        nombre = n;
    }
    public void setSueldo(double s){
        sueldo = s;
    }
    public void printInfo(){
        System.out.println("Nombre: " + nombre + "\nDias: " + dias +
            "\nSueldo: " + sueldo);
    }
}

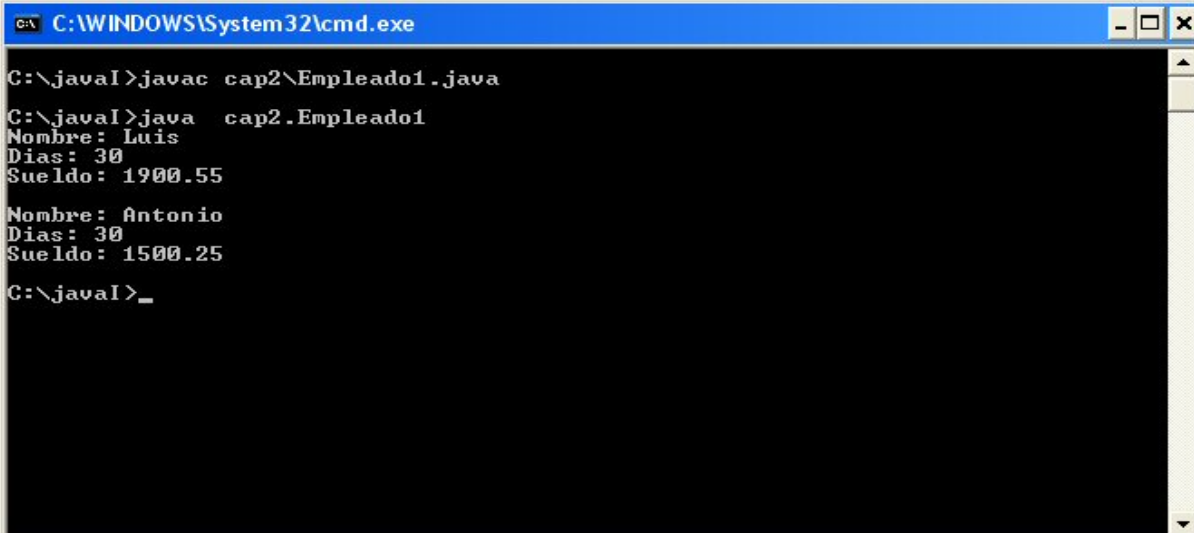
public static void main(String[] args){
    Empleado e1 = new Empleado();
    e1.setNombre("Luis");
    e1.setSueldo(1900.55D);

    Empleado e2 = new Empleado();
    e2.setNombre("Antonio");
    e2.setSueldo(1500.25D);

    e1.printInfo();
    System.out.println();
    e2.printInfo();
}
```

En el ejemplo se observa una clase que define un prototipo de empleado. Todas sus variables se declaran 'private' con lo cual no será posible acceder a ellas desde el exterior de la clase. En cambio, los métodos formarán parte de la interfaz visible del objeto puesto que son declarados 'public' y por tanto visibles desde el exterior de la clase.

En el método principal, 'main', se crean e inicializan varios objetos a partir de la clase Empleado. En la salida del programa se comprueba que e1 y e2 son objetos distintos:



```
C:\WINDOWS\System32\cmd.exe

C:\javaI>javac cap2\Empleado1.java

C:\javaI>java cap2.Empleado1
Nombre: Luis
Dias: 30
Suelo: 1900.55
Nombre: Antonio
Dias: 30
Suelo: 1500.25
C:\javaI>_
```

Setter and Getter

Como los atributos de una Clase son privados, los objetos que instanciamos en las aplicaciones, no tienen acceso a sus atributos, por eso son necesarios:

- un método por cada atributo para ver, es decir que nos devuelva, el contenido de la variable (los métodos getAtributo),
- y un método por cada atributo para asignar contenido a la variable (los métodos setAtributo)

Construcción e inicialización de objetos

La construcción e inicialización de objetos en Java se realiza con el operador 'new'. Con esta llamada:

```
Empleado e1 = new Empleado();
```

- Se reserva memoria para el nuevo objeto y se inicializan sus variables a los valores por omisión.
- Se realiza cualquier inicialización explícita.
- Se ejecuta un constructor.

La primera fase es totalmente invisible y su objetivo es evitar que las variables se creen con valores erróneos.

La inicialización explícita consiste en asignar valores a las variables en el momento de su declaración. Esto ya se ha utilizado en el primer ejemplo:

```
public class Empleado1{
    private int dias=30;
    private String nombre;
    private double sueldo;
```

En concreto, la variable 'dias' toma por valor 30 en el momento de la construcción del objeto, pero esto no es lo más normal. En las clases se definen variables, pero no se les asigna valor.

Para dar valor a las variables debemos emplear los métodos setAtributo, como en el ejemplo:

```
Empleado e1 = new Empleado();
e1.setNombre("Luis");
e1.setSueldo(1900.55D);
```

El método Empleado(), es el llamado constructor por defecto, es decir un método llamado igual que la clase (por eso va en mayúsculas), y que proporciona Java a toda clase que no tiene definido explícitamente ningún constructor, para poder instanciar un objeto de esa clase.

Constructores

Los constructores permiten realizar una inicialización más compleja porque:

- Permiten tratamiento de errores.
- Pueden recibir parámetros en su llamada.
- Se pueden utilizar bucles y/o sentencias condicionales para automatizar cualquier inicialización.

Los constructores se distinguen del resto de métodos por:

- Tienen el mismo nombre que la clase.
- No definen tipo de retorno.

Una vez que la clase decide crear un constructor con parámetros, el constructor por defecto lo deja de incorporar Java. Por tanto si queremos definir un constructor sin parámetros lo tendremos que añadir a la lista de métodos de la clase.

```
public class Empleado2{
    private int dias=30;
    private String nombre;
    private double sueldo;

    private static int numeroEmpleados = 0;

    public Empleado2(String n, double s) {
        nombre = n;
        sueldo = s;
        numeroEmpleados++;
    }
    public String getNombre(){
        return nombre;
    }
    public double getSueldo(){
```

```
        return sueldo;
    }
    public int getDiasVacaciones(){
        return dias;
    }
    public void setNombre(String n){
        nombre = n;
    }
    public void setSueldo(double s){
        sueldo = s;
    }

    public void printInfo(){
        System.out.println("Nombre: " + nombre + "\nDias: " + dias +
            "\nSueldo: " + sueldo);
    }
    public static int getNumeroEmpleados() {
        return numeroEmpleados;
    }
    public static void main(String[] args){
        Empleado2 e1 = new Empleado2("Luis",1900.55D);

        Empleado2 e2 = new Empleado2("Antonio",1500.25D);

        Empleado2 e3 = new Empleado2("Miguel",1200.15D);

        Empleado2 e4 = new Empleado2();
        // error, no existe un constructor sin parámetros
        e1.printInfo();
        System.out.println();
        e2.printInfo();
        System.out.println();
        e3.printInfo();
        System.out.println("\nNumero de Empleados: " +
            e2.getNumeroEmpleados());
    }
}
```

En el ejemplo Empleado2, se añade un método constructor que permite la creación e inicialización del objeto en un solo paso. Se observa también una nueva variable y un nuevo método con el modificador 'static'.

El empleado e4, da error porque no existe un constructor sin parámetros.

El operador "this"

Quiero que te fijes en la definición de este constructor:

```
public Empleado2(String n, double s) {
    nombre = n;
    sueldo = s;
    numeroEmpleados++;
}
```

Los nombres de los parámetro que recibe el constructor son **n** y **s**.

- n para recibir el nombre
- s para recibir el salario

Pero y si para aclarar el constructor decidiera llamar a los parámetros de entrada igual que a las variables de la clase. La definición quedaría:

```
public Empleado2(String nombre, double sueldo) {  
    nombre = nombre; // error  
    sueldo = sueldo; // error  
    numeroEmpleados++;  
}
```

Java no sabe distinguir si nombre se refiere al parámetro de entrada o a su propia variable. Para distinguir se usa el operador `this`, para hacer referencia al objeto de la clase que haya invocado al constructor. Por tanto quedaría:

```
public Empleado2(String nombre, double sueldo) {  
    this.nombre = nombre;  
    this.sueldo = sueldo;  
    numeroEmpleados++;  
}
```

Ten en cuenta que cuando uses una herramienta de desarrollo, hay montones de ayudas para construir las clases, estas ayudas te crean los constructores, y usan los mismos nombres para los parámetros de entrada que para las variables de la clase. De forma que el uso de `this` está a la orden del día.

Sobrecarga de métodos

Esta característica permite tener en una misma clase varios métodos con el mismo nombre pero con distintos argumentos. Aunque se permite que el tipo de retorno sea distinto, no es suficiente, la lista de argumentos tiene que serlo también para poder diferenciarlos. En el siguiente ejemplo se muestra su uso:

```
package cap5;  
  
public class PruebaSobrecarga{  
    public void calcula(int i){  
        //Realiza los calculos necesarios...  
        System.out.println("Calculando con enteros..");  
    }  
    public void calcula(double d){  
        //Realiza los calculos necesarios...  
        System.out.println("Calculando con dobles..");  
    }  
    public static void main(String[] args){  
        calcula(5);  
        calcula(10.0);  
    }  
}
```

La máquina virtual de Java ejecuta uno u otro dependiendo del tipo de parámetros que reciba el método en el momento de la llamada a dicho método.

Los constructores también se sobrecargan, mismo nombre de método, distinto número y/o tipo de parámetros.

Variables y métodos estáticos

Las variables y métodos estáticos, no son propios de un objeto sino comunes a toda la clase.

Las variables 'static' no pertenecen a ninguna instancia si no a la clase. Cualquier cambio realizado sobre ella implicará un cambio del valor de la variable en el resto de los objetos de esa clase.

Esto es así, porque para las variables propias de cada objeto, se realiza una reserva de memoria. En cambio, para las variables estáticas, solo se realiza una reserva en el momento que la clase es referenciada.

Los métodos estáticos, son de uso general y no tienen acceso a la parte propia de un objeto, solamente a su parte estática. Permiten ser llamados mediante el nombre de la clase, no hace falta un puntero.

- Si la variable o el método estático es privado, sólo puede ser referenciado por los métodos de la clase.
- Si son públicos, pueden ser referenciados desde fuera de la clase, simplemente poniendo
 - Clase.variable , por ejemplo System.out : out es una variable static y pública de la Clase System, por eso la podemos referenciar sin crear una variable de la clase System.
 - o Clase.método, por ejemplo el método main de las clases de consola. La máquina virtual de java no tiene que crear un objeto de la Clase a ejecutar, simplemente busca el método main que es estático y público para ejecutarlo.

El Garbage Collector

También conocido como 'Recolector de Basura', es el encargado de la liberación de recursos. Este proceso de liberación de memoria se realiza automáticamente durante la ejecución de un programa Java, de forma paralela y transparente, mediante un hilo que comprueba en los tiempos muertos de la máquina virtual los objetos que han sido derreferenciados para descargarlos.

Esto quiere decir que en nuestras aplicaciones nosotros nos preocupamos de

- crear las referencias a objetos, es decir las variables,
- instanciar el objeto (new Clase(p1,p2)).
- Usar el objeto a través de sus métodos.

De eliminarlo se encarga la máquina virtual de Java.

Sólo en ciertas ocasiones, que ya comentaremos, será necesario que nosotros le eliminemos explícitamente.

Modificadores de métodos y atributos

La siguiente tabla muestra los modificadores empleados:

MODIFICADOR	DESCRIPCION
public	Un método o atributo definido como público puede ser ejecutado o referenciado desde cualquier clase. Normalmente los métodos son públicos y los datos son privados
private	Un método o atributo definido como privado solo es accesible desde la clase que lo ha definido
protected	Un método o atributo definido como protegido es accesible desde la clase que lo define y desde todas las clases derivadas de ella(subclases)
Sin especificar	Un método o atributo que no tiene modificador de acceso solo es accesible desde las clases del mismo paquete.

Por otra parte están los modificadores que definen la naturaleza de un método o atributo. Lo reflejamos en la siguiente tabla:

MODIFICADOR	DESCRIPCION
static	Un método o atributo definido como estático puede ser ejecutado sin crear un objeto que lo ha definido.
final	Un método o atributo definido como final no admite sobrecargas ni sobrescrituras. Un atributo static y final es una constante , y en la nomenclatura se pone la variable en mayúsculas. Normalmente, para que pueda ser referenciada será, además, pública.
synchronized	Un método o atributo definido como sincronizado bloquea el acceso a un dato u objeto hasta que termine de ejecutarse el método, para que ningún otro método pueda alterar este dato u objeto.

Trabajar con clases

Aunque en los ejemplos anteriores hemos definido la clase Empleado con el método main incluida en la clase, vamos a ver ahora que la manera más profesional de definir la aplicaciones es :

1. Crear cada clase en fichero .java con sus métodos propios, constructores, setter and getter, y el resto de métodos de su responsabilidad.
2. Compilar cada clase para ver que no tienen errores.

3. Crear uno o varias clases con el método main, de prueba para comprobar el correcto uso de las clases.
4. Crear escenarios de prueba, con el método main, para probar las distintas partes de la aplicación.

Por tanto en la definición de la clase Empleado del primer punto de la unidad, quedaría con un fichero Empleado.java, y otro fichero con la clase TestEmpleado.java. Ambos se compilan por separado, pero solo se ejecuta el fichero TestEmpleado, que es el que contiene el método main estático.

```
public class Empleado{
    private int dias=30;
    private String nombre;
    private double sueldo;
    public Empleado(){
    }
    public Empleado(String pNombre, double pSueldo){
        nombre=pNombre;
        sueldo = pSueldo;
    }

    public String getNombre(){
        return nombre;
    }
    public double getSueldo(){
        return sueldo;
    }
    public int getDiasVacaciones(){
        return dias;
    }
    public void setNombre(String n){
        nombre = n;
    }
    public void setSueldo(double s){
        sueldo = s;
    }
    public void printInfo(){
        System.out.println("Nombre: " + nombre + "\nDias: " + dias +
            "\nSueldo: " + sueldo);
    }
}
```

```
public class TestEmpleado{
    public static void main(String[] args){
        Empleado e1 = new Empleado();
        e1.setNombre("Luis");
        e1.setSueldo(1900.55D);

        Empleado e2 = new Empleado();
        e2.setNombre("Antonio");
        e2.setSueldo(1500.25D);
        Empleado e3 = new Empleado("Andrés",20000);

        e1.printInfo();
        System.out.println();
        e2.printInfo();
        System.out.println();
        e3.printInfo();
    }
}
```



```
}  
}  
}
```

Manejo de excepciones y creación de excepciones de usuario

TABLA DE CONTENIDOS

Introducción.....	2
Tipos de excepciones	3
Mecanismo de las excepciones	4
Sentencias try-catch.....	4
Sentencia finally	4
La pila de llamadas	5
La regla de declarar o capturar	6
Creación de excepciones de usuario	7

Introducción

Una excepción es un evento que ocurre durante la ejecución de un programa que detiene el flujo normal de la secuencia de instrucciones de ese programa, por tanto es una situación anormal que surge durante la ejecución de una situación de nuestra aplicación.

En realidad lo que ocurre es cuando se está ejecutando una instrucción en un método de una clase, y se produce un error, ocurre la siguiente secuencia de acciones:

1. Se crea un objeto de la clase del error correspondiente.
2. Si el error no se controla, este objeto se propaga al método de la clase que llamó a este método.
3. Si este segundo método lo trata, el objeto muere ahí. Si no se trata se propaga al siguiente método. Y así sucesivamente.
4. Si el objeto error lo propaga el método main, llega a la máquina virtual de java (JMM). Esta siempre para la ejecución de la clase que contiene el main, y por tanto la aplicación se termina anormalmente, y por la consola del cliente le muestra la lista de clases que han intervenido, desde el main hasta el método de la clase que produjo el error.

Java, define la clase Exception para tratar aquellos errores que los programas puedan encontrar. En lugar de finalizar el programa, se puede escribir código para tratar el error y continuar la ejecución normal del programa.

También define la clase Error y la clase Exception, para aquellas situaciones graves que no se pueden tratar para recuperar la ejecución normal del programa.

El tratamiento de errores se sitúa, en un bloque separado, al final del código de ejecución normal. Así el código resultante es más claro y legible. En el momento que se produce un error, la máquina virtual Java cambia automáticamente el flujo de control a la parte de código que trata el error. En el siguiente fragmento se observa la distribución de código:

```
try{
    ...
    código_normal
} catch(Exception e){
    ...
    código_tratamiento_error
    ...
}
```

Tipos de excepciones

En el siguiente árbol se puede observar la jerarquía de clases de la superclase Throwable, para excepciones:

- Exception
 - RuntimeException
 - NullPointerException
 - ArrayIndexOutOfBoundsException
 -
 - IOException
 - FileNotFoundException
 - EOFException
 -
- Error

Las RuntimeException son excepciones que nunca deberían generarse pues son debidas a fallos en la implementación del programa. Por ejemplo,

- ArrayIndexOutOfBoundsException cuando se accede a un índice de una matriz que no existe,
- o NullPointerException cuando una variable de tipo referencia tiene valor null y se intenta acceder al objeto apuntado por ella.

Las IOException así como todas las excepciones que no extiendan RuntimeException siempre tienen que controlarse ya que pueden surgir ante cualquier eventualidad.

Mecanismo de las excepciones

Sentencias try-catch

Las sentencias susceptibles de poder generar algún tipo de excepción que queramos controlar deben situarse dentro del bloque try. Al finalizar este bloque, se sitúa la lista de bloques catch, uno para cada tipo de excepción que quiera capturarse o bien una clase padre que las capture todas.

```
try{
    //codigo que puede lanzar la excepción
}catch(TipoExcepcion1 ex){
    //Tratamiento excepción 1
}catch(TipoExcepcion2 ex){
    //Tratamiento excepcion 2
}
```

Los bloques catch tienen que seguir un orden lógico, es decir, los más generales tienen que ir al final. Obsérvese el siguiente ejemplo:

```
try{
    //codigo que puede lanzar la excepción
}catch(RuntimeException ex){
    //Tratamiento excepción 1
}catch(ArrayIndexOutOfBoundsException ex){
    //Tratamiento excepcion 2
}
```

Este código no compila porque `ArrayIndexOutOfBoundsException` es hija de `RunTimeException` y por lo tanto, ha sido capturada en el primer bloque catch.

Sentencia finally

La sentencia finally define un bloque de código que se ejecuta siempre, independientemente de si se produce una excepción o de si se captura.

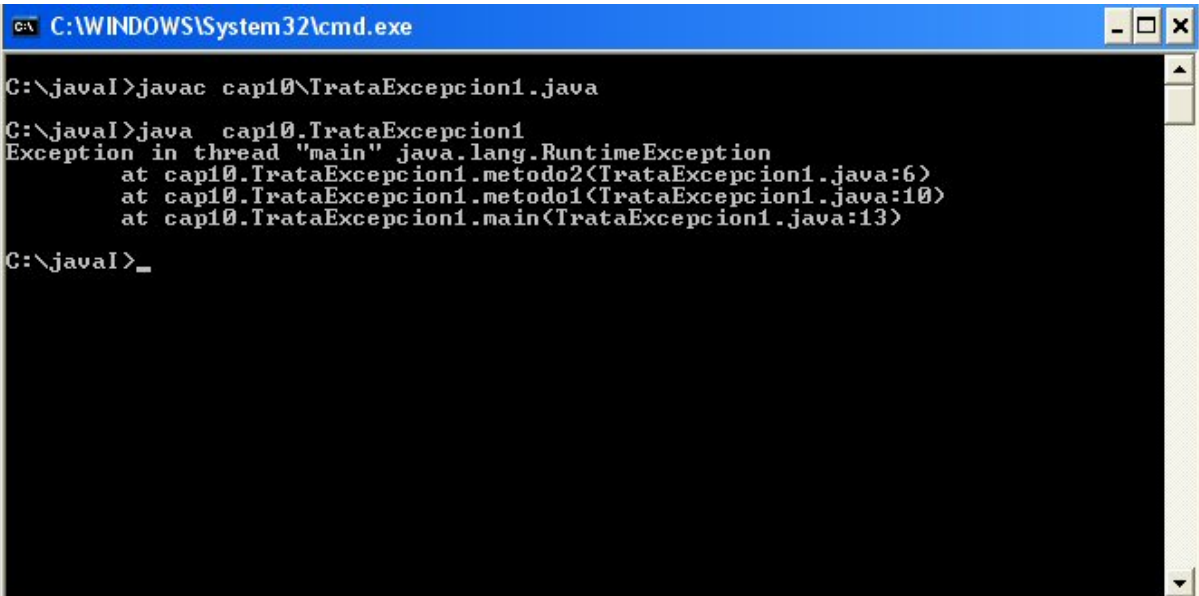
```
try{
    //codigo que puede lanzar la excepción
}catch(TipoExcepcion1 ex){
    //Tratamiento excepción 1
}catch(TipoExcepcion2 ex){
    //Tratamiento excepcion 2
}finally{
    //Se ejecuta siempre
}
```

La pila de llamadas

Como se ha visto, cuando una sentencia de un método encuentra un problema que le impide seguir con su ejecución, crea un objeto de tipo `Exception` y lo lanza. Si el método está preparado para capturar esa excepción, la captura y la trata. Si no es así, el método relanza la excepción al método llamante, y así sucesivamente hasta llegar al método `main()`. Obsérvese el ejemplo:

```
public class TrataExcepcion1{  
    public static void metodo2(){  
        throw new RuntimeException();  
    }  
    public static void metodo1(){  
        metodo2();  
    }  
    public static void main(String[] args){  
        metodo1();  
    }  
}
```

Con la sentencia 'throw' se puede lanzar un objeto `Exception`. Al producirse la excepción en el `metodo2()`, donde no existe bloque try-catch, se relanza al método llamante, es decir, `metodo1()` que tampoco tiene bloque try-catch. Este relanza la excepción al `main()` donde tampoco se captura la excepción y finaliza el programa. Esta es la salida del programa, donde se observa la pila de llamadas:



```
C:\WINDOWS\System32\cmd.exe  
C:\javaI>javac cap10\TrataExcepcion1.java  
C:\javaI>java cap10.TrataExcepcion1  
Exception in thread "main" java.lang.RuntimeException  
    at cap10.TrataExcepcion1.metodo2<TrataExcepcion1.java:6>  
    at cap10.TrataExcepcion1.metodo1<TrataExcepcion1.java:10>  
    at cap10.TrataExcepcion1.main<TrataExcepcion1.java:13>  
C:\javaI>_
```


La regla de declarar o capturar

En los métodos que puedan generar excepciones que no extiendan RuntimeException, se debe tomar una opción sobre la excepción:

- Declarar → La excepción será relanzada al método llamante. Esto se indica con la cláusula 'throws' en la declaración del método. Puede indicar varias excepciones separándolas por comas.
- Capturar → La excepción será capturada en un bloque try-catch.

En el siguiente ejemplo se observa como la excepción se genera en el metodo2() y se declara. También se declara en el método1() y en última instancia, se captura en el método main():

```
import java.io.*;

public class TrataExcepcion2{

    public static void metodo2() throws IOException{
        throw new IOException();
    }

    public static void metodo1() throws IOException{
        metodo2();
    }

    public static void main(String[] args){
        try{
            metodo1();
        }catch(IOException e){
            System.err.println("Excepcion de entrada/salida");
        }
    }
}
```

Creación de excepciones de usuario

El programador puede crear y levantar sus propios objetos de excepción. Esto se consigue creando una clase excepción propia que herede de Exception. Aunque la herencia se ve en profundidad en la siguiente unidad, baste decir ahora que cuando una clase hereda de otra, puede acceder a todos los métodos públicos de la superclase.

1. Crear la clase MiException que hereda de Exception.
 - Definimos dos constructores, el constructor por defecto, y un constructor que recibe una cadena de caracteres para que el programador pueda levantar un objeto de error con el mensaje que considere oportuno. La instrucción `super(mensaje)` indica llamar al constructor de Exception para que se termine de formar el objeto(ya lo entenderás en la siguiente Unidad.)

```
public class miException extend Exception {
    public MiException(){}
    public MiException(String mensaje) {
        super(mensaje)
    }
}
```

2. Levantar y transferir mi excepción desde un método de una clase, analizamos las líneas del código.

Línea 3-4: si se cumple cierta condición, levanto el objeto de mi error con el mensaje " el numero no puede ser cero ".

Línea 2: en la definición del método se decide transmitir el objeto error levantado en la línea 4. Para ello ponemos `throws miException`.

```
1  public Class MiClase {
2      public void procMetodo (int i) throws MiException {
3          if (i==0)
4              throw new MiException("el numero no puede ser cero");
```

3. En el método main que llama al método `procMetodo(0)` de la clase `MiClase`, se decide capturar el error, y en el tratamiento sacar por consola el mensaje que nos mandan desde el método que generó el error.

```
public class TestMiException {
    public static void main(String[] args) {
        MiClase obj = new MiClase();
        try {
            obj.procMetodo(0);
        }catch(miException e)
        {
            System.out.println("mensaje : " + e.getMessage());
        }
    }
}

.....
```

4. En el método main que llama al método procMetodo(7) de la clase MiClase, se decide no capturar el error. Por tanto el error se transmite a la máquina virtual, y está para el proceso informando de la secuencia de métodos de las clases correspondientes implicadas..

```
public class TestMiException {  
    public static void main(Strng[] args) {  
        MiClase obj = new MiClase();  
  
        objMiClase.procMetodo(0);  
        .....  
    }  
}
```

Resultado:

```
Exception in thread "main" java.lang.Error: Unresolved compilation  
problem:  
    Unhandled exception type MiException  
  
    at TestMiException.main(TestMiException.java:7)
```

Como hemos comentado con anterioridad la forma profesional es capturar los errores mediante los bloque try catch.

Características avanzadas de clases

TABLA DE CONTENIDOS

Herencia	2
Sobreescritura de métodos	4
Polimorfismo	5
Casting de objetos	6
La clase Obj	9
El método toString()	9
Clases Abstractas	10
Interfaces	11
Ámbito de visibilidad	14

Herencia

En programación, es muy común crear un modelo o clase genérico y después versiones más concretas. En los siguientes ejemplos vemos las clases Manager y Técnico:

```
public class Tecnico{
    private int dias=30;
    private String nombre;
    private double sueldo;
    private String especialidad = "Ninguna";

    private static int numeroEmpleados = 0;

    public Tecnico(String n, double s) {
        nombre = n;
        sueldo = s;
        numeroEmpleados++;
    }

    public String getNombre(){
        return nombre;
    }

    public double getSueldo(){
        return sueldo;
    }

    public int getDiasVacaciones(){
        return dias;
    }

    public void setNombre(String n){
        nombre = n;
    }

    public void setSueldo(double s){
        sueldo = s;
    }

    public void printInfo(){
        System.out.println("Nombre: " + nombre +
            "\nDias: " + dias + "\nSueldo: " + sueldo);
    }

    public static int getNumeroEmpleados() {
        return numeroEmpleados;
    }

    public void setEspecialidad(String especialidad){
        this.especialidad = especialidad;
    }

    public String getEspecialidad(){
        return especialidad;
    }
}
```

```
package cap5;

public class Manager{
```

```
private int dias=30;
private String nombre;
private double sueldo;

private static int numeroEmpleados = 0;

public Manager(String n, double s) {
    nombre = n;
    sueldo = s;
    numeroEmpleados++;
}

public String getNombre(){
    return nombre;
}

public double getSueldo(){
    return sueldo;
}

public int getDiasVacaciones(){
    return dias;
}

public void setNombre(String n){
    nombre = n;
}

public void setSueldo(double s){
    sueldo = s;
}

public static int getNumeroEmpleados() {
    return numeroEmpleados;
}

public double calcBonus(int extra){
    return sueldo * extra / 100;
}
}
```

En la clase Manager y Tecnico se observa que su contenido es idéntico a Empleado2 pero con un método añadido. Al ser clases distintas, cualquier cambio realizado en los métodos comunes deberá ser aplicado a las tres clases. Para evitar estas duplicaciones de datos y de trabajo, existe el concepto de subclase o herencia, es decir, crear una nueva clase a partir de una que ya existe pero sin tener que reescribir todo el código.

En Java, la herencia se implementa con la palabra reservada 'extends'. De esta manera se puede heredar toda la funcionalidad de una clase. En los siguientes ejemplos se pueden observar las clases Manager y Tecnico utilizando los mecanismos de herencia:


```
public class Tecnico extends Empleado2{

    private String especialidad = "Ninguna";

    public Tecnico(String nombre, double sueldo, String especialidad){
        super(nombre,sueldo);
        this.especialidad = especialidad;
    }
    public void printInfo(){
        super.printInfo();
        System.out.println("\nEspecialidad: " + especialidad);
    }
    public void setEspecialidad(String especialidad){
        this.especialidad = especialidad;
    }
    public String getEspecialidad(){
        return especialidad;
    }
}
```

```
public class Manager extends cap2.Empleado2{

    public Manager(String nombre, double sueldo){
        super(nombre, sueldo);
    }
    public double calcBonus(int extra, int tiempo){
        return tiempo * extra / 100;
    }
}
```

Al heredar ambas clases de la clase Empleado2, cualquier cambio realizado en esta clase se verá reflejado en las otras dos. Los constructores no se heredan y las clases Tecnico y Manager usan sus propios constructores. Desde ellos se accede al constructor de la clase padre mediante la palabra reservada 'super'.

Java solo permite la herencia simple, es decir, solo se puede heredar de una clase en un mismo nivel. Así, la clase Manager tiene la siguiente estructura jerárquica:

Object ← Empleado2 ← Manager

Esto es así porque en Java, toda clase (excepto Object) es hija de Object. Cuando una clase no hereda de otra clase mediante 'extends', automáticamente esa clase heredará de Object.

Sobreescritura de métodos

Al heredar de una clase, algunas veces se quieren añadir nuevas características, otras se quieren modificar características ya existentes. En éste último caso, podemos definir el

método en la nueva clase, con el mismo nombre, mismo tipo de retorno y mismos argumentos que en la clase padre. Obsérvese en el ejemplo como el método de la clase padre ha sido sobrescrito:

```
public class ClasePadre{  
    protected int numero;  
  
    public void metodo(){  
        System.out.println("metodo padre");  
    }  
}
```

```
public class ClaseHijo extends ClasePadre{  
  
    public void metodo(){  
        System.out.println("metodo hijo");  
    }  
}
```

Polimorfismo

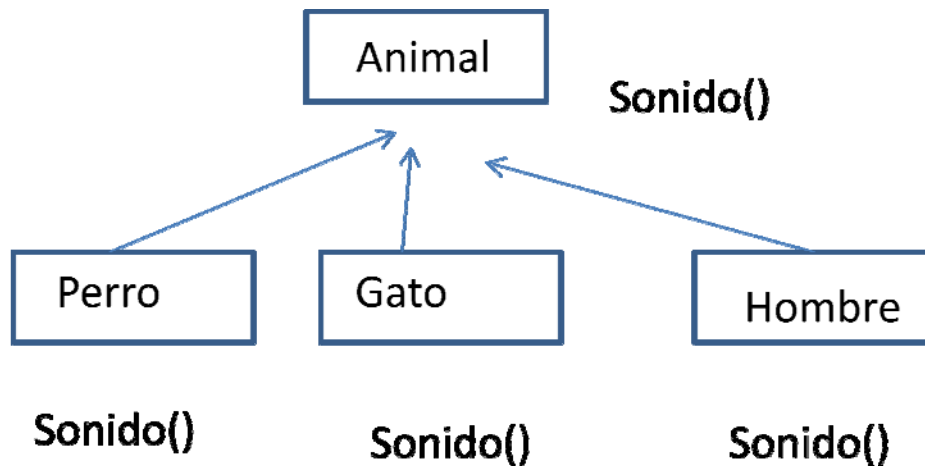
Java, al igual que muchos lenguajes orientados a objetos, permite apuntar a un objeto con una variable del tipo de la clase padre. Así, la siguiente expresión es totalmente correcta:

```
Empleado2 e = new Manager();
```

Este objeto, pese a ser un Manager y tener todas las características de éste, solamente se podrá acceder a la parte propia de Empleado2, porque para el compilador solo es un Empleado2. Cualquier acceso a la parte propia de Manager producirá un error de compilación.

En el caso que se quisiera acceder a la parte de Manager, se tendría que realizar una conversión o casting.

Un ejemplo:



Perro gato y hombre heredan de Animal, que tiene un método sonido(), que implementan sobrescribiéndole sus subclases.

Veamos el siguiente código:

```
Animal a1 = new Perro();
Animal a2 = new Gato();
Animal a3 = new Hombre();
System.out.println("que dice el animal: " + a2.sonido());
//Resultado .. el animal dice miau
```

En este caso a2 es una referencia a Animal, que apunta a un objeto de tipo gato, por tanto al aplicar el método a2.sonido(), se ejecuta el método de gato. Con este mecanismo desde una referencia a la superclase solo podemos ejecutar métodos polimórficos de la subclase.

En este ejemplo desde a2, sólo tenemos acceso a métodos de animal.

Si un Gato tuviera un método no definido en la superclase (por ejemplo cuantasVidas()), desde a2 no tendríamos visibilidad. Por tanto desde a2 para ejecutar un método de Gato, deberíamos de hacer un casting:

```
((Gato)a1).cuantasVidas();
```

Casting de objetos

En los casos que se tiene un objeto apuntado por una variable del tipo del padre y se sepa que es un objeto de una subclase, se puede realizar una conversión para poder acceder a su parte propia. Es importante antes de realizar una conversión comprobar el tipo del objeto. Esto se hace con el operador instanceof. Obsérvese el siguiente fragmento de código:

```
public void doAlgo(Empleado2 e){

    if(e instanceof Manager){
        Manager m = (Manager)e;
        //Ahora ya se puede acceder a la parte propia de Manager
        e.calcBonus(3000,4);
    }else{
        // No se trata de un Manager y no se puede hacer esa conversión.
        // Quizá se trate de un Tecnico.
    }
}
```

Como se ha visto anteriormente, es posible apuntar a un objeto con una variable del tipo de la clase padre. Así, es posible crear arrays de objetos que contengan elementos de distintos tipos. Obsérvese el ejemplo:

```
public class Tecnico extends cap2.Empleado2{

    private String especialidad = "Ninguna";

    public Tecnico(String nombre, double sueldo, String especialidad){
        super(nombre,sueldo);
        this.especialidad = especialidad;
    }
    public void printInfo(){
        super.printInfo();
        System.out.println("\nEspecialidad: " + especialidad);
    }
    public void setEspecialidad(String especialidad){
        this.especialidad = especialidad;
    }
    public String getEspecialidad(){
        return especialidad;
    }
}
```

```
public class PruebaPolimorfismo{

    public static void main(String[] args){

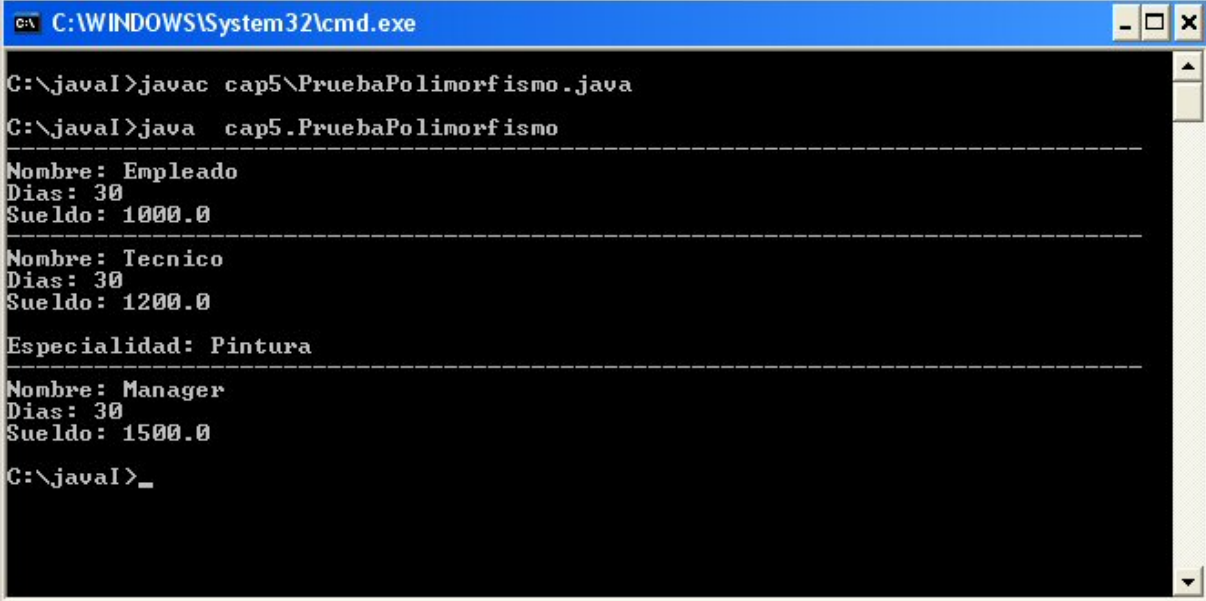
        Empleado2 e = new Empleado2("Empleado",1000);
        Tecnico t = new Tecnico("Tecnico",1200,"Pintura");
        Manager m = new Manager("Manager",1500);

        Empleado2[] empleados = {e,t,m};

        for(int i =0;i< empleados.length;i++){
            System.out.println("-----");
            empleados[i].printInfo();
        }
    }
}
```

En primer lugar tenemos la clase Tecnico modificada, se ha sobrescrito el método printInfo, para que también muestre la especialidad. Obsérvese como se llama al método printInfo desde la subclase, con la palabra reservada 'super'.

En segundo lugar, la clase PruebaPolimorfismo, muestra como un array de Empleado2 puede almacenar referencias a objetos Empleado2, Tecnico o Manager. También se puede observar en la salida, el efecto que tiene una llamada a un método sobrecargado según el objeto:



```
C:\WINDOWS\System32\cmd.exe

C:\>javaI>javac cap5\PruebaPolimorfismo.java
C:\>javaI>java cap5.PruebaPolimorfismo
-----
Nombre: Empleado
Dias: 30
Sueldo: 1000.0
-----
Nombre: Tecnico
Dias: 30
Sueldo: 1200.0
Especialidad: Pintura
-----
Nombre: Manager
Dias: 30
Sueldo: 1500.0
C:\>javaI>_
```

La clase Obj

En Java, todas las clases heredan de Object. Así, el conjunto de métodos de Object estarán disponibles en cualquier objeto Java. Uno de estos métodos es 'toString()'.

El método toString()

El método toString() se utiliza para convertir un objeto en su representación en forma de cadena de caracteres, por ejemplo, al llamar al método System.out.println(). La implementación de este método en la clase Object devuelve el nombre de la clase y la dirección de su puntero, por ejemplo:

```
Empleado e = new Empleado("alvarez",12000);
System.out.println("el empleado es : " + e);
// Resultado → El empleado es : Empleado@5d038b78
```

Es conveniente sobreescribirlo para que devuelva información más útil a cerca de los valores de las variables del objeto correspondiente.

Por ejemplo la Clase String tiene sobreescrito el método toString(), para que aparezca la cadena de caracteres que lo contiene.

```
String nombre = "tomas";
System.out.println("la cadena contiene : " + nombre);
// Resultado ----> la cadena contiene : tomas
```

Clases Abstractas

Existen ocasiones en las cuales se desarrolla una jerarquía de clases y existen ciertos comportamientos en todas ellas, pero se realizan de forma distinta. Para estos casos, Java proporciona las clases abstractas. Las clases abstractas permiten crear clases con métodos implementados y otros solamente declarados, llamados métodos abstractos. Estos métodos abstractos tendrán que ser obligatoriamente sobreescritos por sus subclases.

- La palabra reservada 'abstract' sirve para indicar que una clase es abstracta.
- También se utiliza para marcar a los métodos abstractos.
- Generalmente, una clase abstracta contiene uno o más métodos abstractos, aunque no es un requisito indispensable.
- Las clases abstractas no se pueden instanciar, no se pueden crear objetos de estas clases, se provoca un error.
- Las subclases de clases abstractas tienen que implementar todos los métodos abstractos o también tendrán que ser marcadas como abstractas.

```
abstract class FiguraGeometrica {  
    abstract void dibujar();  
}  
  
class Circulo extends FiguraGeometrica {  
    void dibujar() {  
        // codigo para dibujar Circulo  
    }  
}
```

Interfaces

Los interface son una variación del concepto de clase abstracta. Un interface consiste en un conjunto de declaraciones de métodos, es decir, un conjunto de métodos abstractos. Los interface tienen las siguientes características:

- La palabra reservada 'interface' identifica un interface, en lugar de class.
- Las interface se implementan con la palabra reservada 'implements'.
- Un interface puede ser heredado por otro, y a diferencia de las clases, se permite heredar de múltiples interface.
- Una clase puede implementar varias interface, separadas por comas.
- Ningún método puede tener cuerpo.
- Solo se podrán definir variables 'static final'.
- Todos los métodos son implícitamente públicos y abstractos.

Al igual que sucede en la herencia, se puede definir una variable de tipo interface que apunte a un objeto siempre que su clase implemente ese interface.

Los Interface al no contener ningún tipo de implementación sino solo la declaración de los métodos, son mucho más genéricos y más permisivos.

Esto, se puede observar en el siguiente ejemplo:

```
public interface SerVivo {  
    void saludar();  
}
```

```
public class Criador implements SerVivo {  
  
    @Override  
    public void saludar() {  
        System.out.println("Soy un Criador ,Hola");  
    }  
}
```

```
public abstract class Animal implements SerVivo {  
  
    public abstract void hacerRuido();  
    public void saludar(){  
        hacerRuido();  
    }  
}
```

```
public class Perro extends Animal {  
    public void hacerRuido(){  
        System.out.println("guauuuuuu");  
    }  
    public void moverColita(){  
        System.out.println("muevo la colita");  
    }  
}
```

```
public class Gato extends Animal{  
  
    public void hacerRuido(){  
        System.out.println("miauuuuuu");  
    }  
}
```

```
public class Principal {  
    public static void main(String args[]){  
        SerVivo arra[]=new SerVivo[3];  
        arra[0]=new Criador();  
        arra[1]=new Perro();  
        arra[2]=new Gato();  
        System.out.println("****LECTURA****");  
        for (int i=0;i<arra.length;i++){  
            arra[i].saludar();  
            if (arra[i]instanceof Perro){  
                ((Perro) arra[i]).moverColita(); //casting a Perro  
            }  
        }  
    }  
}
```

```
//Resultado  
Soy un Criador, Hola  
Guauuuu  
muevo la colita  
miauuuuuu
```

Las interfaces normalmente se utilizan para:

- Declarar métodos que serán implementados en una o más clases.
- Implementar similitudes entre clases sin forzarlas a tener una relación como ocurriría con las clases abstractas.

Ámbito de visibilidad

- **public:** Para que los métodos y/o atributos objeto sea accesible desde el exterior. LO VE TODO EL MUNDO. (Se puede crear un objeto y luego se puede acceder con “punto”: objeto.xxxx).

```
ClaseObjeto  objMio = new ClaseObjeto( );  
ObjMio.metodos( );
```

- **private:** Solo se puede acceder al objeto desde el interior de la clase. Desde el exterior de la clase NO LO VE NADIE.
- **protected:** (protegido). Solo accesible si se hereda. Solo lo ven las clases que lo extienden.
- **Nada**(no poner modificador): visible entre las clases pertenecientes al mismo paquete.
- **static:** Se le puede llamar sin crear un objeto de la clase: “clase.método”. Los métodos static solo pueden llamar a métodos de su misma clase que sean también static.

```
NombreClase.Metodo( );           Ej: Math.random( );  
NombreClase.DatoMiembro;        Ej: Math.PI;
```

RESUMEN:

```
(mayor visibilidad)  PUBLIC > PROTECTED > PRIVATE  (menor visibilidad)
```

Cadenas y Clases envolventes de tipos básicos de datos

TABLA DE CONTENIDOS

Cadenas. La Clase String	2
Métodos de la Clase String	4
Clases envolventes de Tipos Básicos de Datos	7
Métodos de las clases envolventes	7

Cadenas. La Clase String

Las instancias de la clase String representan secuencias de caracteres Unicode. Un objeto String tiene un valor constante e inmutable. Cualquier modificación realizada en una instancia de String implicará la creación de un nuevo objeto. Para manipular cadenas de caracteres, Java proporciona la clase StringBuffer.

A diferencia del resto de clases, String posee optimizaciones particulares en la máquina virtual de Java, por eso su comportamiento también es distinto.

Los objetos String se pueden crear de dos formas:

```
String s1 = new String("Modo normal") ;  
String s2 = "Modo Particular";
```

Cuando se empieza a trabajar con Java tendemos a pensar que una variable de tipo String se comporta igual que una variable de tipo primitivo o básica (entero, real, booleano y carácter). Analicemos este código:

```
int i = 10;  
int j = 3;  
j=i;  
if (j==i)  
    System.out.println("ambos numeros son iguales");  
else  
    System.out.println("son numeros distintos");
```

Resulta obvio que las variables i y j tienen el mismo valor.

Ahora bien analicemos ahora el siguiente código:

```
String cadena1, cadena2;  
cadena1= "pepe";  
cadena2 = "pepe";  
  
if (cadena1 == cadena2)  
    System.out.println("ambas cadenas son iguales");  
else  
    System.out.println("son cadenas distintas");
```

Parece obvio que el resultado es que "ambas cadenas son iguales".

Pero ahora vamos a probar el siguiente código, que aparentemente es igual al anterior:

```
String cadena1, cadena2;  
cadena1= new String("pepe");  
cadena2 = new String("pepe");  
  
if (cadena1 == cadena2)  
    System.out.println("ambas cadenas son iguales");  
else  
    System.out.println("son cadenas distintas");
```

El resultado es que ambas cadenas son distintas. Y te preguntarás cómo puede ser esto. Pues bien, String es una clase, cadena1 y cadena2 son dos variables que contienen la dirección (referencia) de donde se encuentra el objeto en memoria. Por tanto cuando preguntamos

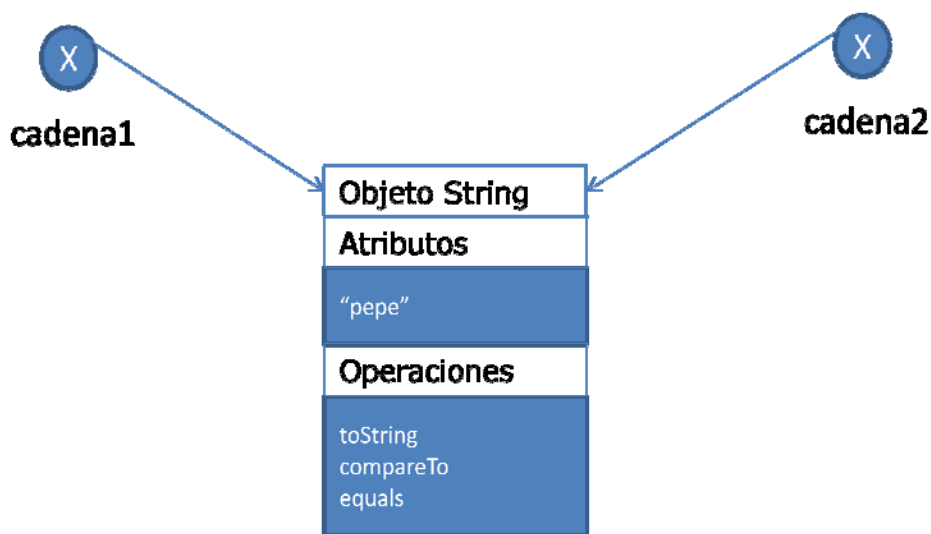
```
If (cadena1 == cadena2)
```

Estamos diciendo que si la dirección de cadena1 es la misma que la dirección de cadena2, no que si el contenido de cadena1 es el mismo que el contenido de cadena2.

Analizamos las dos situaciones:

1.

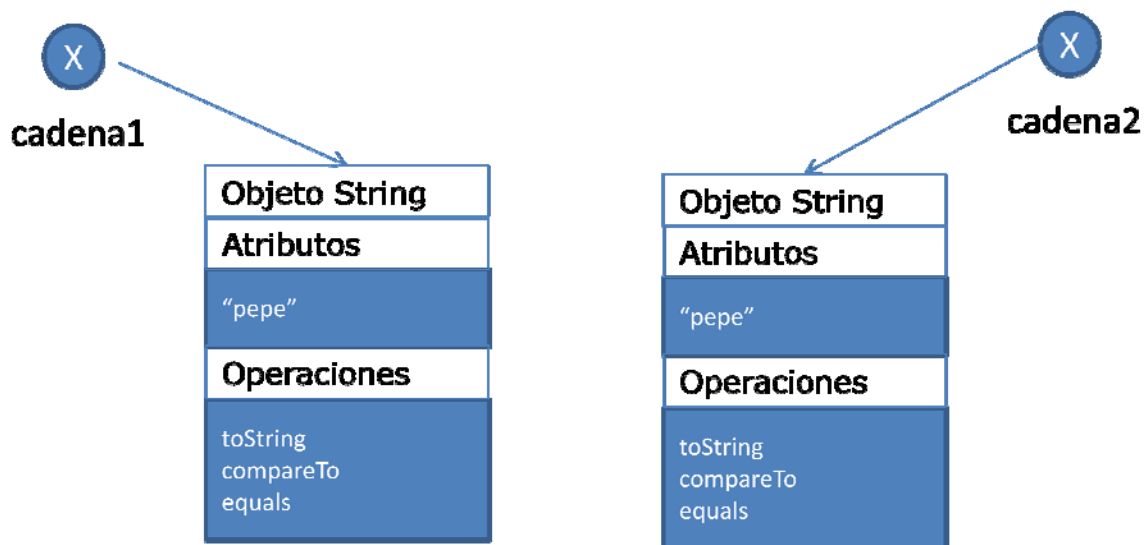
```
String cadena1, cadena2;  
cadena1= "pepe";  
cadena2 = "pepe";  
  
if (cadena1 == cadena2)  
    System.out.println("ambas cadenas son iguales");  
else  
    System.out.println("son cadenas distintas");
```



String cadena1 = "pepe", crea un objeto que contiene "pepe"; String cadena2="pepe", apunta al mismo objeto que cadena1. Por eso ambas referencias, cadena1 y cadena2, son iguales.

2.

```
String cadena1, cadena2;  
cadena1= new String("pepe");  
cadena2 = new String("pepe");  
  
if (cadena1 == cadena2)  
    System.out.println("ambas cadenas son iguales");  
else  
    System.out.println("son cadenas distintas");
```

String cadena1 = new String("pepe"), crea un objeto que contiene "pepe"; String cadena2= new String("pepe"), crea otro objeto que también contiene "pepe". Por eso ambas referencias, cadena1 y cadena2, son distintas.

Cuando hablamos de objetos solo preguntaremos que dos variables o referencias son iguales, cuando queremos saber si apuntan al mismo objeto. Si queremos saber si contienen lo mismo usaremos el método "equals", que es el que compara el contenido de ambos objetos, en este caso de ambas cadenas de caracteres.

El método equals lo tienen todos los objetos, porque recordemos que equals es un método de la Clase Object, y de esta clase heredan todos los objetos. Cada clase deberá determinar la manera en que estimamos que el contenido de dos objetos de la misma clase son iguales, por tanto deberá sobrescribir el método equals para adaptarlo a sus necesidades.

Métodos de la Clase String

La variable "cadena" antes creada almacena un objeto cadena por lo que a través de ella podemos acceder a las propiedades y a los métodos de la clase String.

Veamos los métodos más habituales de los objetos String para la gestión de cadenas.

- public int cadena.length()

Devuelve el número de caracteres que tiene la cadena.

```
System.out.println ("Tiene " + cadena.length() + "caracteres" );
//Resultado Tiene n caracteres
```

- public char cadena.charAt(int indice)

Devuelve el carácter que ocupa la posición índice. El primer carácter estará siempre en la posición cero.

- `public String cadena.substring(int beginIndex, int endIndex)`

Devuelve la cadena comprendida entre la posición del primer índice (incluido) hasta el segundo índice (excluido). (Si fuera de rango: `IndexOutOfBoundsException`).

```
cadena = "Primer Objeto String" ;
System.out.println (cadena.substring( 7,13 ) );
```

- `public int cadena.indexOf(String str)`

Devuelve la primera posición que ocupa la subcadena que se pasa como parámetro, (-1 si no existe la subcadena).

```
cadena.indexOf( "er" ) -----> 4
```

```
cadena.indexOf( "O" ) -----> 7
```

- `public int cadena.indexOf(String str, int fromIndex)`

Devuelve la posición que ocupa la cadena str en su primera ocurrencia a partir de la posición Index.

```
cadena = "Primer Objeto String Primer Programa JAVA" ;
System.out.println ("La subcadena comienza en: " +
cadena.indexOf( "Primer" , 7 ) ); 21
System.out.println ("La subcadena comienza en: " +
cadena.indexOf( "primer" , 7 ) ); -1
```

- `public boolean cadena.equals(Object unObjeto)`

Devuelve true si la cadena parámetro es igual que la cadena objeto. Distingue como siempre entre mayúsculas y minúsculas.

```
String cadenal = new String("pepe");
String cadena2= new String("pepe");
if (cadenal.equals(cadena2))
    System.out.println("ambas cadenas son iguales");
else
    System.out.println("son cadenas distintas");
//Resultado: ambas cadenas son iguales
```

- `public boolean cadena.equalsIgnoreCase(String otroString)`

Devuelve true si la cadena parámetro es igual que la cadena objeto. NO Distingue entre mayúsculas y minúsculas.

- `public static String String.valueOf(tipogenerico dato)`

Convierte a String cualquier tipo de dato que le pasemos como parámetro.

- `public String cadena.concat(String str)`

Concatena el valor del parámetro con el valor del objeto.

```
cadena = "HOLA" ;  
cadena = cadena.concat(" BUENOS DIAS");  
System.out.println ("Toda la cadena completa= " + cadena );
```

- `public static String String.toUpperCase|toLowerCase(String cad)`

Convierte a mayúsculas o minúsculas una cadena.

```
cadenaMayusculas = String.concat(" BUENOS DIAS");
```

Clases envolventes de Tipos Básicos de Datos

Multitud de Clases al definir sus métodos admiten como parámetros referencias a objetos. Si quiero pasarles un número de cualquier tipo, o un carácter o un booleano, no podría pues estos son tipos primitivos.

Para estos casos el Java incorpora dentro de su paquete `java.lang`, un conjunto de clases que envuelven a cada uno de estos tipos básicos, de esta forma tendremos las clases:

- `Byte`
- `Short`
- `Integer`
- `Long`
- `Float`
- `Double`
- `Boolean`
- `Character`

Como ves la mayoría se llaman igual que el tipo primitivo, excepto porque empiezan, al ser clases, por mayúsculas.

Métodos de las clases envolventes

Estas clases están dotadas de métodos estáticos y no estáticos. Para su estudio vamos a tomar como referencia la clase `Integer`. Entre las que cabe destacar los siguientes métodos:

- `Integer(int value)` y `Integer(String s)`

Los dos constructores de la clase, uno admite un entero, y el otro una cadena de caracteres con contenido numérico. No existe constructor sin parámetros.

- `public static Integer valueOf(int)`

Conversión de entero a Objeto `Integer`

```
Integer numero;  
numero= Integer.valueOf(3);  
System.out.println("numero contiene : " + numero);  
//Resultado numero contiene : 3
```

- `public static Integer valueOf(String)`

Conversión de `String` que contiene un literal numérico a `Integer`

```
Integer numero;  
numero= Integer.valueOf("233");  
System.out.println("numero contiene : " + numero);  
//Resultado numero contiene : 233
```

- `public static int parseInt(String s)`

Convierte una cadena con contenido numérico a entero

```
int numero;  
numero= Integer.parseInt("1250");  
System.out.println("numero contiene : " + numero);  
//Resultado numero contiene : 1250
```

- `int intValue(), long longValue(), double doubleValue() ...`

Convierte el Integer sobre el que se aplica el método a entero, o double o long ...

```
int numero = 1000;  
Integer numeroDos;  
numeroDos= new Integer("250");  
numero = numeroDos.intValue();  
System.out.println("numero contiene : " + numero);  
//Resultado numero contiene : 250
```

- `static String toString(int i)`

Devuelve un String que contiene el entero convertido a literal por el método.

```
String cadena;  
cadena= Integer.toString(300);  
System.out.println("cadena contiene : " + cadena);  
//Resultado cadena contiene : 300
```

Documentación

TABLA DE CONTENIDOS

Paquetes	2
Crear paquetes de clases	2
El paquete java.lang (El Paquete de Lenguaje Java)	4
El paquete java.util (El Paquete de utilidades).....	5
La Clase Scanner	5
Documentar proyectos con javadoc	7
Comentarios para documentación	7
Una clase comentada	8
Convenciones de nombres	8
Un ejemplo: class Empleado.....	8
El API de java.....	12

Paquetes

Las clases en java se agrupan en paquetes.

Un paquete en java es un conjunto de clases que se almacenan en un directorio cuyo nombre es el nombre del paquete.

Cuando queremos utilizar en nuestro programa una clase que se encuentra en un determinado paquete, hay que importar el paquete.

Para importar los paquetes se utiliza la instrucción `import` que debe situarse siempre al principio del código fuente incluso antes de la definición de la clase.

Toda clase que utilicemos debemos de importarla.

Formato:

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

Se están importando todas las clases del paquete "io" que a su vez se encuentra dentro del paquete "java".

Los paquetes fundamentales de java son:

- El paquete `java.lang` que es el corazón de java. (Se amplía más adelante).
- El paquete `java.io` que define una serie de clases para dar soporte a la E/S.
- El paquete `java.math` que soporta operaciones matemáticas.
- El paquete `java.net` que soporta E/S de red basada en URL's.
- El paquete `java.awt` que proporciona soporte a aplicaciones gráficas.
- El paquete `java.applet` que proporciona soporte para los applets.
- El paquete `java.text` que tiene clases para internacionalizar aplic, incluyendo soporte para fechas, formatos de monedas, ordenes de clasificación, etc.
- El paquete `java.util` que contiene gran variedad de clases que soportan estructuras de datos, números aleatorios, fecha, zonas horarias, calendarios, etc...

Crear paquetes de clases

Al compilar un programa.java los ficheros con el "programa.class" se guardan por defecto en el directorio en el que estamos trabajando.

Como toda clase por definición pertenece a un paquete, se dice que una clase en el raíz del proyecto pertenece la "paquete por defecto".

Cuando el número de clases de un proyecto aumenta mucho, su gestión se hace impracticable, por lo tanto tenderemos a organizar nuestros proyectos en directorios diferentes, es decir paquetes distintos.

- Paquete Modelo: en donde van las clases del negocio.
- Paquete Test: en donde van las clases de Pruebas de clases.

- Paquete Escenario: en donde van las pruebas de ejecución de situaciones de casos de uso.

Cuando una clase pertenece a un directorio concreto, es decir a un paquete, en la implementación de la clase debemos indicarlo usando la palabra `package`, en la clase donde se vaya a utilizar debemos poner la palabra `import`. (Los paquetes son directorios que contienen clases).

Ejemplo:

```
package Modelo;

public class Empleado {
    private String nombre;
    private double salario;
    public Empleado(String nombre, double salario) {
        super();
        this.nombre = nombre;
        this.salario = salario;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public double getSalario() {
        return salario;
    }
    public void setSalario(double salario) {
        this.salario = salario;
    }
    public void pintar() {
        System.out.println("nombre : " + nombre + "  salario : "+ salario);
    }
}
```

Y en la clase de otro paquete que use esta clase, debemos importar el paquete:

```
import Modelo.Empleado;

public class TestHerencia {

    public static void main(String[] args) {
        Empleado e1 = new Empleado("eva perez", 10000);
        Empleado e2 = new Empleado("susana garcia", 20000);

        e1.pintar();
        System.out.println("*****");
        e2.pintar();

    }

}
```

La importación de las clases está íntimamente ligado al contenido del parámetro CLASSPATH. Este indica los directorios a partir de los cuales encontrar los paquetes (directorios) con las clases de JAVA.

SET CLASSPATH=.;C:\;C:\jdk1.7...\lib\classes.zip;C:\cursojavabasicos\;



En el ejemplo de arriba al compilar Clase1.java y Clase2.java sus .class correspondientes quedarán por defecto en sus respectivos directorios.

Cuando se van a usar clases todas ellas en el mismo directorio no es necesario importar las clases ya que el "." primero del classpath facilita su localización y asume que las clases están en el directorio actual.

El paquete java.lang (El Paquete de Lenguaje Java)

El paquete de lenguaje Java, también conocido como java.lang, contiene las clases que son el corazón del lenguaje Java.

El compilador importa automáticamente este paquete. Ningún otro paquete se importa de forma automática.

Las clases del paquete java.lang se agrupan de la siguiente manera:

- **Object:**

El padre de todas las clases. Es la clase de la que parten todas las demás. Y toda clase en Java hereda de Object.

- **Tipos de Datos Encubiertos:**

Una colección de clases utilizadas para encubrir variables de tipos primitivos: Boolean, Character, Double, Float, Integer y Long. Cada una de estas clases es una subclase de la clase abstracta Number.

- **Strings:**

Dos clases que implementan los datos de caracteres. Las Clases String y StringBuffer .

- **System y Runtime:**

Estas dos clases permiten a los programas utilizar los recursos del sistema. System proporciona un interface de programación independiente del sistema para recursos del sistema y Runtime da acceso directo al entorno de ejecución específico de un sistema.

- **Thread:**

Las clases Thread, ThreadDeath y ThreadGroup implementan las capacidades multitareas tan importantes en el lenguaje Java. El paquete java.lang también define el interface Runnable. Este interface es conveniente para activar la clase Java sin utilizar la clase Thread.

- **Class:**

La clase Class proporciona una descripción en tiempo de ejecución de una clase y la clase ClassLoader permite cargar clases en los programas durante la ejecución.

- **Math:**

Una librería de rutinas matemáticas y valores matemáticos como PI. Todos los métodos son estáticos.

- **Exceptions, Errors y Throwable:**

Cuando ocurre un error en un programa Java, el programa lanza un objeto que indica qué problema era y el estado del intérprete cuando ocurrió el error. Sólo los objetos derivados de la clase Throwable pueden ser lanzados para manejar errores en los programas Java.

- **Process:**

Los objetos Process representa el proceso del sistema que se crea cuando se utiliza el sistema en tiempo de ejecución para ejecutar comandos del sistema. El paquete java.lang define e implementa la clase genérica Process.

El paquete java.util (El Paquete de utilidades)

Lo conforman un conjunto de clases englobadas en tres grandes apartados:

- Colecciones, las tratamos con profundidad en la siguiente unidad.
- Entrada datos por consola: Scanner.

La Clase Scanner

Hasta la versión 5 de Java, la entrada de datos por consola se hacía a través de clases del paquete java.io. La dificultad era que:

- estas clases generan excepciones que hay que controlar, y

- Lo que lees de consola eran cadenas de caracteres, y tenías que usar métodos de las clases envolventes para convertir de String a tipos básicos.

Con Scanner, no se generan excepciones, y además la clase está provista de métodos para transformar la cadena de caracteres en el tipo básico de datos que te interese en cada momento.

Veamos un ejemplo de cómo se implementa:

```
import java.util.Scanner;

public class pruebaScanner {
    public void main(String [] args){
        System.out.println("introduce un número : ");
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("el número es : " + i);
        sc.close();
    }
}

Resultado
introduce un número :
7
el número es : 7
```

Analizamos las siguientes consideraciones:

- La clase Scanner pertenece al paquete java.util, por eso hay que importarlo antes de la definición de la clase.
- Cuando nos creamos un objeto de la clase Scanner, le pasamos en el constructor **System.in**, que representa a la Consola del ordenador.
- El método **nextInt()**, devuelve un entero a partir de la cadena con contenido numérico que introducimos por consola. Evidentemente si la cadena contiene algún carácter no numérico se nos levanta una excepción del tipo "java.util.InputMismatchException".
- Existen métodos para todos los tipos básicos de datos: nextDouble(), nextFloat()....
- Existe un método para leer una cadena de caracteres: nextLine().

Antes de terminar el método hay que ejecutar el método **sc.close()**; para cerrar el objeto de la clase Scanner.

Documentar proyectos con javadoc

En Java existen comentarios de línea con `//` y bloques de comentario que comienzan con `/*` y terminan con `*/`. Por ejemplo:

```
// Comentario de una línea
/* comienzo de comentario
   continua comentario
   fin de comentario */
```

Comentarios para documentación

El JDK proporciona una herramienta para generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. El nombre de la herramienta es javadoc. Para que javadoc pueda generar los textos HTML es necesario que se sigan unas normas de documentación en el fuente, que son las siguientes:

Los comentarios de documentación deben empezar con `/**` y terminar con `*/`.

Se pueden incorporar comentarios de documentación a nivel de clase, a nivel de variable (dato miembro) y a nivel de método.

Se genera la documentación para miembros public y protected.

Se pueden usar tags para documentar ciertos aspectos concretos como listas de parámetros o valores devueltos. Los tags se indican a continuación.

Tipo de tag	Formato	Descripción
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Toda la documentación del API de Java está creada usando esta técnica y la herramienta javadoc.

Una clase comentada

```
import java.util.*;

/** Un programa Java simple.
 * Envía un saludo y dice que día es hoy.
 * @author Antonio Bel
 * @version 1
 */
public class HolaATodos {

    /** Unico punto de entrada.
     * @param args Array de Strings.
     * @return No devuelve ningun valor.
     * @throws No dispara ninguna excepcion.
     */
    public static void main(String [ ] args) {
        System.out.println("Hola a todos");
        System.out.println(new Date());
    }
}
```

Convenciones de nombres

SUN recomienda un estilo de codificación que es seguido en el API de Java y en estos apuntes que consiste en:

- Utilizar nombres descriptivos para las clases, evitando los nombres muy largos.
- Para los nombres de clases poner la primera letra en mayúsculas y las demás en minúsculas. Por ejemplo: Empleado
- Si el nombre tiene varias palabras ponerlas todas juntas (sin separar con - o _) y poner la primera letra de cada palabra en mayúsculas. Por ejemplo: InstrumentoMusical.
- Para los nombres de miembros (datos y métodos) seguir la misma norma, pero con la primera letra de la primera palabra en minúsculas. Por ejemplo: registrarOyente.
- Para las constantes (datos con el modificador final) usar nombres en mayúsculas, separando las palabras con _

Un ejemplo: class Empleado

Tenemos la siguiente definición de Clase Empleado

```
public class Empleado {

    private String nombre,apellido,email;
    private int edad;
    private double salario;
    public Empleado(String nombre, String apellido, int edad) {
        super();
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
}
```

```

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
    public double getSalario() {
        return salario;
    }
    public void setSalario(double salario) {
        this.salario = salario;
    }
}

```

Al ejecutar el programa de utilidad javadoc nos aparece la siguiente información:

CLASS EMPLEADO

java.lang.Object

Empleado

```

public class Empleado
extends java.lang.Object

```

CONSTRUCTOR SUMMARY

Constructors

Constructor and Description

Empleado(java.lang.String nombre, java.lang.String apellido, int edad)

METHOD SUMMARY

Methods

Modifier and Type	Method and Description
java.lang.String	getApellido()
int	getEdad()
java.lang.String	getEmail()

java.lang.String	getNombre()
double	getSalario()
void	setApellido(java.lang.String apellido)
void	setEdad(int edad)
void	setEmail(java.lang.String email)
void	setNombre(java.lang.String nombre)
void	setSalario(double salario)

METHODS INHERITED FROM CLASS JAVA.LANG.OBJECT

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

CONSTRUCTOR DETAIL**EMPLEADO**

public Empleado(java.lang.String nombre, java.lang.String apellido, int edad)

METHOD DETAIL**GETNOMBRE**

public java.lang.String getNombre()

SETNOMBRE

public void setNombre(java.lang.String nombre)

GETAPELLIDO

public java.lang.String getApellido()

SETAPELLIDO

public void setApellido(java.lang.String apellido)

GETEMAIL

public java.lang.String getEmail()

SETEMAIL

public void setEmail(java.lang.String email)

GETEDAD

public int getEdad()

SETEDAD

public void setEdad(int edad)

GETSALARIO

```
public double getSalario()
```

SETSALARIO

```
public void setSalario(double salario)
```

El API de java

Toda la documentación de todos los paquetes que java proporciona de forma gratuita, para que cualquier desarrollador pueda realizar aplicaciones en Java, está totalmente documentada siguiendo los criterios expuestos hasta ahora.

Como bien sabes la parte visible de java, es decir su interface, está formado por los métodos que son visibles, es decir por su parte pública, que es lo que se muestra en cada una de las clases, interfaces, clases abstractas...

La página principal del API de Java en la versión que elijas(en este caso el JSE 7, tiene siempre la misma apariencia:

The screenshot shows the Java Platform Standard Edition 7 API Specification website. Three blue boxes with arrows point to specific features:

- Todas las clases** (All classes) points to the 'All Classes' link in the left sidebar.
- Todos los paquetes** (All packages) points to the 'Overview' link in the top navigation bar.
- Menú de opciones** (Options menu) points to the 'Package' link in the top navigation bar.

The main content area displays the 'Java™ Platform, Standard Edition 7 API Specification'. It includes a table of packages with the following data:

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.

Cuando decimos todas las clases, queremos decir, la lista ordenada alfabéticamente de:

- Clases
- Clases Abstractas
- Interfaces

Si seleccionamos un paquete, por ejemplo java.util, obtenemos a la izquierda de la pantalla, la lista ordenada de Interfaces, Clases...

Interfaces

```
Collection
Comparator
Deque
Enumeration
EventListener
Formattable
Iterator
List
ListIterator
Map
Map.Entry
NavigableMap
NavigableSet
Observer
Queue
RandomAccess
Set
SortedMap
SortedSet
```

Classes

```
AbstractCollection
AbstractList
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractQueue
AbstractSequentialList
AbstractSet
ArrayDeque
ArrayList
Arrays
BitSet
Calendar
Collections
Currency
Date
Dictionary
EnumMap
EnumSet
EventListenerProxy
EventObject
FormattableFlags
Formatter
GregorianCalendar
HashMap
HashSet
Hashtable
IdentityHashMap
LinkedHashMap
LinkedHashSet
LinkedList
ListResourceBundle
Locale
Locale.Builder
Objects
Observable
PriorityQueue
Properties
PropertyPermission
PropertyResourceBundle
Random
ResourceBundle
ResourceBundle.Control
```

```
Scanner  
ServiceLoader  
SimpleTimeZone  
Stack  
StringTokenizer  
Timer  
TimerTask  
TimeZone  
TreeMap  
TreeSet  
UUID  
Vector  
WeakHashMap
```

Enums

```
Formatter.BigDecimalLayoutForm  
Locale.Category  
Exceptions  
ConcurrentModificationException  
DuplicateFormatFlagsException  
EmptyStackException  
FormatFlagsConversionMismatchException  
FormatterClosedException  
IllegalFormatCodePointException  
IllegalFormatConversionException  
IllegalFormatException  
IllegalFormatFlagsException  
IllegalFormatPrecisionException  
IllegalFormatWidthException  
IllformedLocaleException  
InputMismatchException  
InvalidPropertiesFormatException  
MissingFormatArgumentException  
MissingFormatWidthException  
MissingResourceException  
NoSuchElementException  
TooManyListenersException  
UnknownFormatConversionException  
UnknownFormatFlagsException
```

Errors

```
ServiceConfigurationError
```

Y a la derecha cada una de explicada.

Si seleccionamos una clase, o interface, nos da su herencia, constructores, constantes, y lista de métodos. Su diagrama de herencia, sus interfaces... Y la lista de métodos con su explicación, y en ocasiones con ejemplos.

Colecciones

TABLA DE CONTENIDOS

Introducción.....	2
Interfaces	3
Collection	3
List	4
Set.....	5
Map.....	5
SortedMap	6
Implementaciones	7
ArrayList.....	7
LinkedList.....	9
HashSet	10
TreeSet.....	11
HashMap.....	12
TreeMap.....	12
Ejemplo de uso.....	14

Introducción

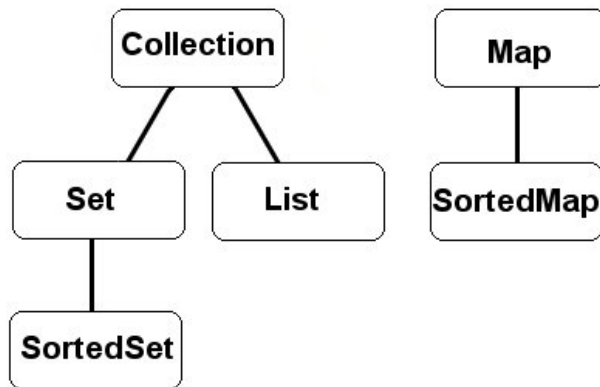
Una colección es un objeto que agrupa en sí múltiples objetos. Permiten la manipulación, almacenaje y obtención de datos así como la transmisión como parámetros de métodos. Las colecciones generalmente almacenan objetos del mismo tipo, pero al estar basadas en Object, pueden almacenar objetos de cualquier tipo.

Las colecciones se basan en una arquitectura unificada llamada "Collections Framework". Ésta contiene:

- Interfaces → Tipos abstractos de datos que representan a las colecciones. Los interfaces permiten la manipulación de las colecciones independientemente de su implementación.
- Implementaciones → Clases java que implementan los interfaces definidos en este framework.
- Algoritmos → Son un conjunto de métodos que realizan funciones tales como búsqueda y ordenamiento en los objetos almacenados en la colección. Estos métodos son reusables puesto que pueden ser utilizados en múltiples implementaciones de un mismo interface.
- List → Es un interface que permite almacenar objetos con un orden. Existen dos implementaciones:
 - ArrayList → Generalmente se utiliza esta implementación de List porque es más eficiente. El acceso a una posición concreta de ArrayList es constante.
 - LinkedList → El acceso a una posición concreta de LinkedList es lineal y por lo tanto, mucho más lento. Su uso está justificado solamente en aquellos casos que se vayan a realizar muchas inserciones al principio o borrado de elementos iterando.
- Set → Es un interface que no permite duplicados. Existen dos implementaciones:
 - HashSet → Esta implementación es mucho más rápida.
 - TreeSet → Permite el ordenamiento de los objetos.
- Iterator → Es un interface que unifica todas las colecciones. Permite el acceso a Collection de una manera uniforme. Es similar a Enumeration pero como se observa en el ejemplo, también permite el borrado de elementos.

Interfaces

En el siguiente gráfico se muestra la jerarquía de las interfaces:



Collection

El interface Collection es el padre de la jerarquía Collection. Una colección representa a un conjunto de objetos de la manera más genérica posible. Proporciona un conjunto de métodos para:

- Contar elementos
 - `int size()`
 - `boolean isEmpty()`
- Comprobar existencia de elementos en la colección
 - `boolean contains(Object element)`
 - `boolean containsAll(Collection c)`
- Inclusión o borrado de elementos
 - `boolean add(Object element)`
 - `boolean remove(Object element)`
 - `boolean addAll(Collection c)`
 - `boolean removeAll(Collection c)`
 - `void clear()`
 - `boolean retainAll(Collection c)`
- Obtención del array de objetos
 - `Object[] toArray()`
 - `Object[] toArray(Object a[])`
- Obtención del iterador
 - `Iterator iterator()`

El método `iterator`, devuelve un objeto `Iterator`. Éste es muy similar a `Enumeration` pero lo mejora en dos aspectos:

- Permite la eliminación de los elementos contenidos en la `Collection`
- Los nombres de los métodos han sido mejorados

La interfaz `Iterator` define los siguientes métodos:

- `boolean hasNext()`
- `Object next()`
- `void remove()`

El método `remove` solo se puede llamar una vez por cada llamada a `next` y lanzará una excepción si esto no se cumple.

En el siguiente fragmento de código se puede ver el uso de `Iterator`:

```
public void utilizaIterator(Collection collection){
    Iterator iterator = collection.iterator();
    while(iterator.hasNext()){
        System.out.println(iterator.next());
    }
}
```

List

`List` es una colección ordenada de elementos. Puede contener objetos duplicados. Añade a los métodos de `Collection`, métodos para:

- Inclusión o borrado de elementos según su posición
 - `void add(int index, Object element)`
 - `Object remove(int index)`
 - `boolean addAll(int index, Collection c)`
- Obtención y sustitución de elementos según su posición
 - `Object get(int index)`
 - `Object set(int index, Object element)`
- Búsqueda de elementos
 - `int indexOf(Object o)`
 - `int lastIndexOf(Object o)`
- Obtención del iterador para `List`
 - `ListIterator listIterator()`

- ListIterator listIterator(int index)
- Obtención de un rango de elementos
 - List subList(int from, int to);

List, proporciona un iterador llamada ListIterator, que permite el recorrido en ambas direcciones, modificar la lista durante su iteración y obtener la posición actual del iterador. ListIterator extiende a Iterator y añade los siguientes métodos:

- boolean hasPrevious()
- Object previous()
- int nextIndex()
- int previousIndex()
- void remove()
- void set(Object o)
- void add(Object o)

Set

La interfaz Set no añade ningún método nuevo a Collection. Añade la restricción de no permitir elementos duplicados.

SORTEDSET

SortedSet es un Set que mantiene los elementos ordenados de forma ascendente, de acuerdo con su orden natural o bien de acuerdo a un Comparator proporcionado en el momento de la creación del SortedSet. El orden natural y los Comparator serán comentados mas adelante en este mismo capítulo.

Añade a los métodos existentes en Set, métodos para:

- Obtener un rango de elementos
 - SortedSet subSet(Object fromElement, Object toElement)
 - SortedSet headSet(Object toElement)
 - SortedSet tailSet(Object fromElement)
- Obtener el primer elemento o el último
 - Object first()
 - Object last()
- Obtener el comparador asociado a este SortedSet o nulo si utiliza orden natural
 - Comparator comparator()

Map

La interfaz Map representa a un objeto que relaciona el par clave-valor. No permite claves duplicadas pero si valores. Como se ha visto en el gráfico anterior, Map no hereda de la interfaz Collection sino que empieza su propia jerarquía.

Proporciona los siguientes métodos para:

- Inclusión y eliminación de objetos
 - `Object put(Object key, Object value)`
 - `Object remove(Object key)`
 - `void putAll(Map t)`
 - `void clear()`
- Obtención de objetos
 - `Object get(Object key)`
 - `public Set keySet()`
 - `public Collection values()`
 - `public Set entrySet()`
- Comprobación de existencia de objetos
 - `boolean containsKey(Object key)`
 - `boolean containsValue(Object value)`
- Contar elementos
 - `int size()`
 - `boolean isEmpty()`

Map contiene un subinterface llamado Entry (Map.Entry) que contiene el par clave-valor para representar una entrada.

SortedMap

SortedMap es un Map que mantiene los elementos ordenados de forma ascendente, de acuerdo con su orden natural o bien de acuerdo a un Comparator proporcionado en el momento de la creación del SortedMap. El orden natural y los Comparator serán comentados mas adelante en este mismo capítulo.

Añade a los métodos existentes en Map, métodos para:

- Obtener un rango de elementos
 - `SortedMap subMap(Object fromKey, Object toKey)`
 - `SortedMap headMap(Object toKey)`
 - `SortedMap tailMap(Object fromElement)`
- Obtener el primer elemento o el último
 - `Object firstKey()`
 - `Object lastKey()`
- Obtener el comparador asociado a este SortedSet o nulo si utiliza orden natural
 - `Comparator comparator()`

Implementaciones

Todas las implementaciones incorporadas a JDK permiten objetos nulos, no están sincronizadas y son serializables.

ArrayList

Implementa el interface List. Ofrece mejor rendimiento que LinkedList en términos generales.

```
java.util
```

Class ArrayList<E>

```
java.lang.Object
```

```
    java.util.AbstractCollection<E>
```

```
        java.util.AbstractList<E>
```

```
            java.util.ArrayList<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

Alguno de sus métodos:

boolean	add(E e) Añade el element especificado al final de la lista.
void	add(int index, E element) Inserta el elemento especificado en la posición indicada.
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order returned by the specified collection's Iterator.
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified index.
void	clear() Borra todos los elementos de la lista
boolean	contains(Object o) Devuelve true si esta lista contiene al elemento especificado.
E	get(int index)

	Devuelve el elemento de la posición especificada.
int	indexOf(Object o) devuelve el índice de la primera ocurrencia del elemento especificado en esta lista , o -1 si esta lista no contiene este elemento
boolean	isEmpty() Devuelve true si la lista no contiene elementos.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or contain the element.
E	remove(int index) Borra el elemento de la posición especificada
boolean	remove(Object o) Borra la primera ocurrencia del objeto especificado, si existe.
boolean	removeAll(Collection<?> c) Borra todos los elementos de la Colección especificada.
int	size() Returns the number of elements in this list.

Obsérvese el siguiente ejemplo:

```
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListTest{
    public static void main(String[] args){
        ArrayList l1 = new ArrayList();

        l1.add("a");
        l1.add("b");
        l1.add("c");
        l1.add("d");
        l1.add("e");

        System.out.println(l1);
        l1.set(1, "X");
        System.out.println(l1);

        for(int i=0, t=l1.size();i<t;i++){
            System.out.println(l1.get(i));
        }
    }
}
```

En este ejemplo se crea un ArrayList y se añaden distintos objetos. Se cambia el objeto situado en la posición 1 y seguidamente se recorre la lista mediante un bucle. La salida del ejemplo es:

```
[a, b, c, d, e]
[a, X, c, d, e]
a
X
c
d
e
```

LinkedList

Implementa el interface List. Tiene mejor rendimiento que ArrayList cuando se quiera insertar o eliminar elementos en cualquier posición y no al principio o final. También puede mejorar a ArrayList cuando se quiera acceder a los elementos de forma secuencial repetidamente.

Obsérvese el ejemplo.

```
import java.util.LinkedList;
import java.util.Iterator;

public class LinkedListTest{

    public static void main(String[] args){

        LinkedList list = new LinkedList();

        list.add("a");
        list.add(0, "b");
        list.add("c");
        list.add("d");
        list.addLast("f");
        list.addFirst("e");

        System.out.println("Numero de elementos: " + list.size());
        System.out.println(list);

        list.removeLast();
        System.out.println(list);
        System.out.println("Ultimo elemento: " + list.get(list.size()-1));
        System.out.println("Primer elemento: " + list.get(0));

        System.out.println("Eliminamos el elemento 'b': " +
list.remove("b"));
        System.out.println("Eliminamos el elemento 'b': " +
list.remove("b"));
        System.out.println("Eliminamos el primer elemento: " +
list.remove(0));
        System.out.println(list);
    }
}
```


En este ejemplo se crea un LinkedList y se añaden distintos objetos. También se añaden al principio y al final. Se muestra como obtener elementos según su posición así como eliminar elementos de varias formas. La salida del ejemplo es la siguiente:

```
Numero de elementos: 6
[e, b, a, c, d, f]
[e, b, a, c, d]
Ultimo elemento: d
Primer elemento: e
Eliminamos el elemento 'b': true
Eliminamos el elemento 'b': false
Eliminamos el primer elemento: e
[a, c, d]
```

HashSet

Implementa el interface Set. HashSet ofrece mejor rendimiento que TreeSet pero no permite obtener los elementos de una manera ordenada. Los objetos añadidos a un HashSet deben redefinir el método hashCode() para un mejor rendimiento.

En el siguiente ejemplo se muestra el uso de HashSet:

```
package cap1;

import java.util.HashSet;
import java.util.Iterator;

public class HashSetTest{

    public static void main(String[] args){

        HashSet set = new HashSet();

        set.add("a");
        set.add("b");
        set.add("c");

        set.remove("c");
        System.out.println("Numero de elementos: " + set.size());

        set.add("a");

        System.out.println(set.contains("a"));
        System.out.println(set.contains("d"));

        Iterator iterator = set.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
}
```

En este ejemplo se crea un HashSet y se añaden distintos objetos. Se muestra como comprobar si la colección contiene un determinado elemento y se itera mediante un objeto Iterator. La salida es:

```
Numero de elementos: 2
true
false
a
b
```

TreeSet

Implementa el interface SortedSet. Es útil cuando se necesita extraer los elementos de la colección de una forma ordenada. Los elementos añadidos a TreeSet deben ser ordenables. Generalmente es más rápido introducir los elementos a un HashSet y después crear un TreeSet a partir del HashSet.

Obsérvese el ejemplo:

```
package cap1;

import java.util.HashSet;
import java.util.TreeSet;
import java.util.Iterator;

public class TreeSetTest{

    public static void main(String[] args){

        HashSet hashSet = new HashSet();

        hashSet.add("a");
        hashSet.add("b");
        hashSet.add("c");

        TreeSet set = new TreeSet(hashSet);
        System.out.println("Primer Elemento: " + set.first());

        Iterator it = set.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

En este ejemplo se crea un TreeSet a partir de los elementos contenidos en un HashSet. Seguidamente se muestra el contenido del TreeSet mediante un Iterator. La salida del ejemplo es:

```
Primer Elemento: a
a
b
c
```

HashMap

Implementa el interface Map. HashMap ofrece mucho mejor rendimiento que TreeMap pero no permite ordenamientos. Los objetos que representan la clave, deben redefinir el método hashCode() para un mejor rendimiento.

Obsérvese el siguiente ejemplo:

```
import java.util.HashMap;

public class HashMapTest{

    public static void main(String[] args){
        HashMap map = new HashMap();

        map.put("clave1", new Integer(3));
        map.put("clave2", "Otro elemento");

        Integer i = (Integer) map.get("clave1");
        System.out.println(i);

        String temp = (String) map.get("clave2");
        System.out.println(temp);
    }
}
```

En este ejemplo se crea un HashMap y se añaden varios pares clave-valor. Se muestra como obtener el valor conociendo la clave. La salida es:

```
3
Otro elemento
```

TreeMap

Implementa el interface SortedMap. Es mas lento que HashMap pero permite realizar ordenaciones.

En el siguiente ejemplo se muestra el uso de TreeMap:

```
import java.util.*;

public class TreeMapTest{

    public static void main(String[] args){
        TreeMap tree = new TreeMap();

        tree.put("clave1", new Integer(3));
        tree.put("clave2", "Otro elemento");

        Set set = tree.entrySet();
        for (Iterator iterator = set.iterator(); iterator.hasNext();){
            Map.Entry me = (Map.Entry)iterator.next();
            System.out.print(me.getKey() + "    " + me.getValue()+"\n");
        }
    }
}
```

En este ejemplo se crea un `TreeMap` y se añaden distintos pares clave-valor. Se obtiene un `Set` conteniendo un conjunto de objetos de tipo `Map.Entry` y se muestra la información mediante un `Iterator`. La salida del ejemplo es la siguiente:

```
clave1    3
clave2    Otro elemento
```

Ejemplo de uso

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

public class EjemploGeneral {

    public static void main(String[] args) {
        List l1 = new ArrayList();
        List l2 = new LinkedList();

        String[] cadenas = { "uno", "dos", "tres", "cuatro", "uno", "tres"
};
        for (int i = 0; i < cadenas.length; i++) {
            l1.add(cadenas[i]);
        }
        printList(l1, "Lista primera:");

        for (int i = 0; i < cadenas.length; i++) {
            l2.add(cadenas[i]);
        }
        printList(l2, "Lista segunda (con los mismos elementos que la
primera:)");

        System.out.println("El elemento 3 es " + l2.get(2));
        l2.remove(2);
        l2.remove("uno");
        printList(l2, "Lista segunda modificada:");

        Set s = new HashSet();
        for (int i = 0; i < cadenas.length; ++i) {
            s.add(cadenas[i]);
        }
        printCollection(s, "Set normal sin orden:");

        s.clear();
        printCollection(s, "Set vacío");

        s = new HashSet(l1);
        printCollection(s, "Set inicializado con List");

        s = new TreeSet(l1);
        printCollection(s, "Set inicializado con List y ordenado");

        printCollection(l1, "List imprimida como Collection");

        for (Iterator iter = l1.iterator(); iter.hasNext();) {
            String element = (String)iter.next();
            if("tres".equals(element)) {
                iter.remove();
            }
        }
        printCollection(l1, "List anterior con elementos borrados a través
de un Iterator");
    }

    private static void printList(List l1, String message) {
        System.out.println("*****");
    }
}
```

```

        System.out.println(message);
        for (int i = 0; i < ll.size(); ++i) {
            System.out.println(i + ": " + ll.get(i));
        }
        System.out.println("*****\n");
    }

    private static void printCollection(Collection col, String message) {
        System.out.println("*****");
        System.out.println(message);
        for (Iterator iter = col.iterator(); iter.hasNext();) {
            String element = (String)iter.next();
            System.out.println("\t" + element);
        }
        System.out.println("*****\n");
    }
}

```

La salida del ejemplo es:

```

*****
Lista primera:
0: uno
1: dos
2: tres
3: cuatro
4: uno
5: tres
*****

*****
Lista segunda (con los mismos elementos que la primera:
0: uno
1: dos
2: tres
3: cuatro
4: uno
5: tres
*****
El elemento 3 es tres
*****
Lista segunda modificada:
0: dos
1: cuatro
2: uno
3: tres
*****

*****
Set normal sin orden:
    cuatro
    tres
    uno
    dos
*****

*****
Set vacío
*****

```

```
*****
Set inicializado con List
    cuatro
    tres
    uno
    dos
*****

*****
Set inicializado con List y ordenado
    cuatro
    dos
    tres
    uno
*****

*****
List imprimida como Collection
    uno
    dos
    tres
    cuatro
    uno
    tres
*****

*****
List anterior con elementos borrados a través de un Iterator
    uno
    dos
    cuatro
    uno
```

Construcción básica de interfaces gráficos de usuario

TABLA DE CONTENIDOS

Introducción.....	2
Desarrollo con Swing.....	3
Contenedores alto nivel	4
Contenedores intermedios	7
componentes ligeros	11
tipos de componentes ligeros	12
Gestión de eventos	25
Creación de eventos	25
Interfaces o listener	26
Clases adaptadoras	28

Introducción

Existe un conjunto de librerías capaces de desarrollar componentes gráficos en Java. Esas librerías están disponibles en el software JDK1.7. Existen dos tipos de librerías:

- AWT (Abstract Windows Toolkit). Todas las clases que pueden implementarse se encuentran en el paquete `java.awt`. Define un conjunto de componentes para el desarrollo de interfaces gráficas o GUI. El aspecto visual del componente lo define la plataforma con lo que son elementos muy pesados para desarrollo.
- Swing .Desarrollado en 1997,define un conjunto de librerías que tiene componentes independientes de la plataforma que podemos encontrar en `javax.swing`. Desarrolla componentes más avanzados además de estar escrito completamente en Java y ser independiente de la plataforma. Considerados más rápidos y ligeros este paquete, utiliza frecuentemente clases de la librerías `java.awt`, con lo que es habitual encontrar que nuestros programas realizados en Swing contengan esta librería.
- Para trabajar con Swing es necesario bajarse el software del JDK1.7 y acceder al Api de Java que podemos encontrarla en la siguiente dirección, <http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>, para saber como desarrollar las clases necesarias.



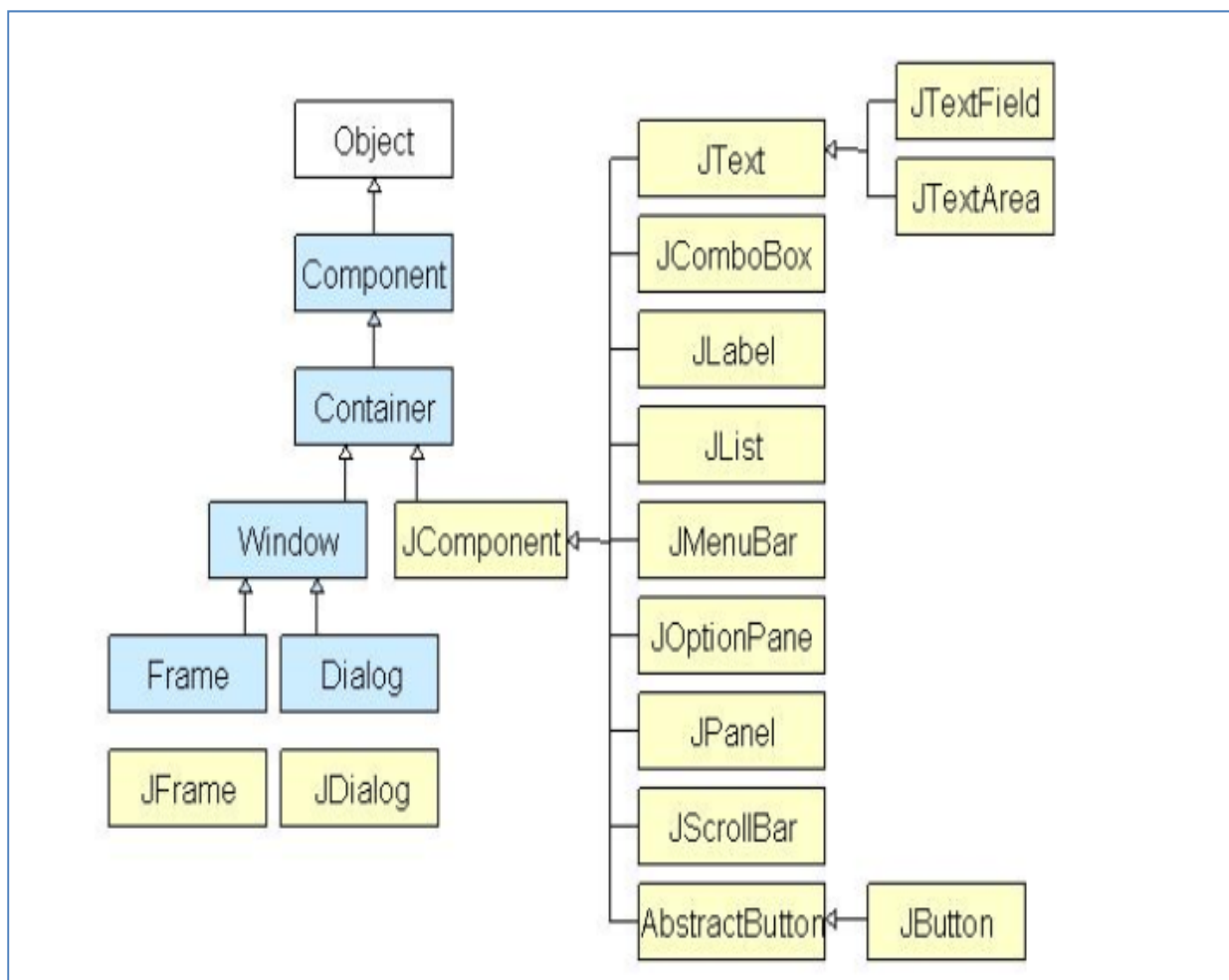
Desarrollo con Swing

Existen distintos plug-ins que se pueden utilizar en función de la herramienta que usemos para desarrollar estos tipos de componentes. En nuestro caso, al utilizar Eclipse existe un módulo instalable llamado WindowBuilder, que facilita la tarea de desarrollar estos componentes. Ver instalación en el audiovisual adjunto.

Swing define tres tipos de componentes:

- Contenedores de Alto nivel → Existen para proporcionar un lugar donde pintarse otros componentes Swing. Son las JFrame , JWindow, JDialog y JApplet
- Contenedores Intermedios → Permiten simplificar el posicionamiento de los componentes ligeros. Son JPanel, JTabbedPane y JScrollPane.
- Componentes Ligeros → Son componentes con algún tipo de comportamiento que tienen que ser añadidos a un panel de contenidos, nunca a un contenedor de alto nivel directamente. Existen muchos componentes ligeros como por ejemplo JButton, JMenu, JLabel, etc..

El siguiente dibujo, muestra la jerarquía de utilización de componentes así como las clases necesarias.



Contenedores alto nivel

Los contenedores son los que soportan los elementos de los cuales están formados la aplicación. El más utilizado es JFrame, es el contenedor por excelencia.

DESARROLLAR CON JFRAME

Para desarrollar con JFrame es necesario que nuestras clases hereden de este. Para ello debemos de importar el paquete javax.Swing. A continuación se muestra algunos de los métodos más utilizado.

Métodos	Definición
setVisible(boolean)	Permite hacer visual el contenedor
setSize(ancho,alto)	En pixels permite ver el tamaño del contenedor
Pack()	Ajusta componentes al tamaño de la ventana.
Add(Object)	Permite añadir componentes intermedios y ligeros al JFrame
setLayout(Layout)	Permite distribuir los componentes a lo largo del JFrame
setTitle(texto)	Cambia el título del marco
setDefaultCloseOperation	Permite que el botón de aspa de la ventana realice distintas opciones: <ul style="list-style-type: none"> • DISPOSE_ON_CLOSE: Se cierra la ventana actual y puede ser mostrada de nuevo • EXIT_ON_CLOSE: Se cierra el programa actual.
setPreferredSize(new Dimension(ancho, alto));	Establece las dimensiones del JFrame establecidas en pixels.

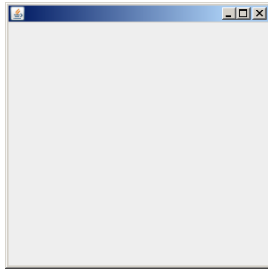
EJEMPLO DE UTILIZACIÓN JFRAME

```
import java.awt.Button;

public class BasicoP extends JFrame
{public BasicoP(){
    this.setLayout(new FlowLayout());

    this.setSize(300, 300);
    this.setVisible(true);
    this.setDefaultCloseOperation(this.DISPOSE_ON_CLOSE);
    this.add(b1);
}
public static void main(String args[]){
    BasicoP ventana=new BasicoP();
}
}
```

La visualización del JFrame quedaría así:



ADMINISTRADORES DE ESQUEMA

Antes de añadir elementos al JFrame o JPanel, debemos de indicar la forma de distribución de los elementos dentro de ese Panel. Para ello, están los distintos administradores de diseño, que añadidos al método `setLayout`, permitirán que los elementos añadidos al JFrame se incluyan en un orden determinado. Los distintos Administradores de Diseño son:

FlowLayout

FlowLayout es el contenedor por defecto en todos los JPanel. Es el gestor más simple, va posicionando componentes de izquierda a derecha, creando nuevas filas en caso necesario. A veces, en función del sistema operativo puede cambiar el posicionamiento.

```
package cap7;

import java.awt.*;
import javax.swing.JButton;
import javax.swing.JFrame;

public class FlowTest extends JFrame{

    public FlowTest(){
        this.setLayout(new FlowLayout());
        this.add(new JButton("Boton 1"));
        this.add(new JButton("2"));
        this.add(new JButton("Boton 3"));
        this.add(new JButton("4"));
        this.add(new JButton("Boton 5"));
    }

    public static void main(String args[]){
        FlowTest instancia = new FlowTest();
        instancia.setTitle("FlowLayout");
        instancia.pack();
        instancia.setVisible(true);
    }
}
```

La ejecución del programa quedaría de la siguiente manera:



GridLayout

GridLayout divide el contenedor en partes iguales, según el número de filas y columnas indicado.

```
package cap7;

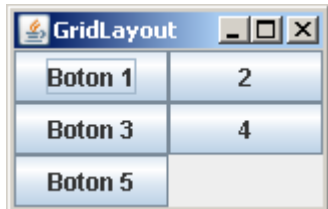
import java.awt.*;

public class GridTest extends JFrame{

    public GridTest(){
        this.setLayout(new GridLayout(0, 2));
        this.add(new JButton("Boton 1"));
        this.add(new JButton("2"));
        this.add(new JButton("Boton 3"));
        this.add(new JButton("4"));
        this.add(new JButton("Boton 5"));
    }

    public static void main(String args[]){
        GridTest instancia = new GridTest();
        instancia.setTitle("GridLayout");
        instancia.pack();
        instancia.setVisible(true);
    }
}
```

La ejecución del programa quedaría de la siguiente manera:



BorderLayout

BorderLayout es el gestor por defecto para cada JFrame, que es el contenedor principal en todos los frames, applets y dialogs.

Contiene cinco áreas disponibles para contener componentes: norte, sur, este, oeste y centro.

```
package cap7;

import java.awt.*;
import javax.swing.JButton;
import javax.swing.JFrame;

public class BorderTest extends JFrame{

    public BorderTest(){

        this.add(new JButton("Norte"), BorderLayout.NORTH);
        this.add(new JButton("Centro"), BorderLayout.CENTER);
        this.add(new JButton("Oeste"), BorderLayout.WEST);
        this.add(new JButton("Sur"), BorderLayout.SOUTH);
        this.add(new JButton("Este"), BorderLayout.EAST);
    }
}
```

```

public static void main(String args[]){
    BorderTest instancia = new BorderTest();
    instancia.setTitle("BorderLayout");
    instancia.pack();
    instancia.setVisible(true);
}

```

La ejecución del programa quedaría así:



Contenedores intermedios

Se posicionan en un Frame y son los siguientes:

JPanel
 JTabbedPane
 JScrollPane.

DESARROLLAR CON JPANEL

Es un panel que se añade a un frame que contiene un conjunto de componentes. De esta manera JPanel facilita la manera de distribuir un conjunto de componentes en un JFrame.

Métodos	Definición
setVisible(boolean)	Permite hacer visual el contenedor
setSize(ancho,alto)	En pixels permite ver el tamaño del contenedor
Pack()	Ajusta componentes al tamaño de la ventana.
Add(Object)	Permite añadir componentes intermedios y ligeros al JPanel
setLayout(administrador de diseño)	Permite distribuir los componentes a lo largo del JPanel
setBackground(Color)	Cambia el color del Panel

EJEMPLO DE UTILIZACIÓN JPANEL.

```

package cap7;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Panel;

import javax.swing.JButton;

```



```

import javax.swing.JFrame;
import javax.swing.JPanel;

public class EjemPanel extends JFrame{
    public EjemPanel(){
        //Creamos el panel
        JPanel p=new JPanel();
        //Indicamos la distribución de los elementos en el Panel
        p.setLayout(new FlowLayout());
        //Creamos los componentes Componentes ligeros
        JButton b1=new JButton("Boton Panel 1");
        JButton b2=new JButton("Boton Panel 2");
        JButton b3=new JButton("Boton Panel 3");
        //Añadimos al panel los componentes ligeros
        p.add(b1);p.add(b2);p.add(b3);
        //Indicamos la distribución de componentes en el JFrame
        this.setLayout(new BorderLayout());
        //Añadimos el panel al JFrame
        this.add(p,BorderLayout.NORTH);
        //Podemos añadir más paneles o elementos al JFrame
        JButton boton =new JButton("Boton Centro");
        this.add(boton,BorderLayout.CENTER);
        this.setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemPanel ej=new EjemPanel();
    }
}

```

DESARROLLAR CON JTABBEDPANE

Es un tipo de componente intermedio que permite la definición de un contenedor de pestañas. Los métodos más utilizados son los siguientes.

Métodos	Definición
JTabbedPane()	Crea una pestaña.
addTab(String,Icon,Component,String)	Añade un panel a la pestaña correspondiente. Con los parámetros de etiqueta,icono, panel y comentarios.

EJEMPLO DE UTILIZACIÓN JTABBEDPANE

```

package cap7;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Panel;

import javax.swing.JButton;
import javax.swing.JFrame;

```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.SwingConstants;

public class ejemTab extends JFrame{
public ejemTab(){
// crear objeto de Pestañas
JTabbedPane ContenedorPestaña = new JTabbedPane();
// Creamos un Panel1 con una etiqueta
JLabel etiqueta1 = new JLabel( "panel uno", SwingConstants.CENTER );
JPanel panel1 = new JPanel();
panel1.add( etiqueta1 );
//Añadir el panel al contenedor de pestañas
ContenedorPestaña.addTab( "Tab 1", null, panel1, "Primer panel" );

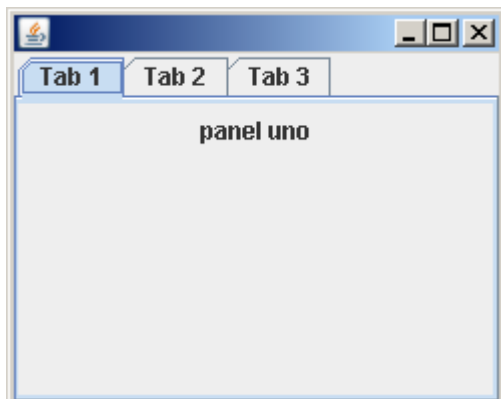
// Creamos un Panel2 con una etiqueta
JLabel etiqueta2 = new JLabel( "panel dos", SwingConstants.CENTER );
JPanel panel2 = new JPanel();
panel2.setBackground( Color.YELLOW );
panel2.add( etiqueta2 );
//Añadir Panel2 al contenedor de pestañas
ContenedorPestaña.addTab( "Tab 2", null, panel2, "Segundo panel" );

// Creamos un Panel3 con una etiqueta
JLabel etiqueta3 = new JLabel( "panel tres" );
JPanel panel3 = new JPanel();
//Modificamos a Panel3 la distribución de compoentes
panel3.setLayout( new BorderLayout() );
//Añadimos a Panel3 los elementos
panel3.add( new JButton( "Norte" ), BorderLayout.NORTH );
panel3.add( new JButton( "Oeste" ), BorderLayout.WEST );
panel3.add( new JButton( "Este" ), BorderLayout.EAST );
panel3.add( new JButton( "Sur" ), BorderLayout.SOUTH );
panel3.add( etiqueta3, BorderLayout.CENTER );
//Añadimos al contenedor de pestañas Panel3
ContenedorPestaña.addTab( "Ficha tres", null, panel3, "Tercer panel" );

// agregamos el contenedor de pestañas al JFrame
this.add( ContenedorPestaña );

setSize( 250, 200 );
setVisible( true );
}
public static void main(String args[]){
ejemTab ej=new ejemTab();
}
}
```

La ejecución del programa quedaría así.



DESARROLLAR CON JSCROLLPANE

Es un tipo de componente intermedio, que crea un panel con barras de desplazamiento horizontal y lateral. Los métodos más importantes son:

Métodos	Definición
JScrollPane(elemento)	Se añade un elemento con barras de desplazamiento
setHorizontalScrollBarPolicy	<p>Tiene las siguientes constantes definidas en la clase ScrollPaneConstants:</p> <p>VERTICAL_SCROLLBAR_AS_NEEDED:Verticalmente solo visualice cando se está fuera de la longitud del eje de las Y.</p> <p>HORIZONTAL_SCROLLBAR_AS_NEEDED:Horizontalmente solo se visualice cando se está fuera de la longitud del eje de las X</p> <p>VERTICAL_SCROLLBAR_AS_ALWAYS:Verticalmente se visualiza siempre</p> <p>HORIZONTAL_SCROLLBAR_AS_ALWAYS:Horizontalmente solo se visualiza siempre.</p> <p>VERTICAL_SCROLLBAR_AS_NEVER:Verticalmente no se visualiza</p> <p>HORIZONTAL_SCROLLBAR_AS_NEVER:Horizontalmente no se visualiza.</p>

EJEMPLO DE UTILIZACIÓN JSCROLLPANE

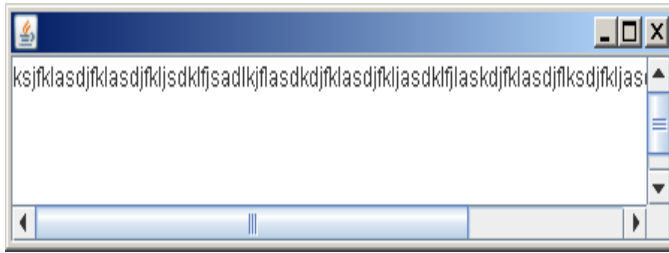
```
package cap7;
import javax.swing.*;
import java.awt.Dimension;
import java.awt.FlowLayout;
import javax.swing.border.BevelBorder;
import java.awt.Font;
public class ejemScrollbar extends JFrame{
    public ejemScrollbar() {
        // Crear panel de Scroll
        JScrollPane scroll = new JScrollPane(new JTextArea(5,30));
        //Indicar visualización del las barras de desplazamiento

        scroll.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);

        //Tamaño del JFrame
        setPreferredSize(new Dimension(450, 110));
        //Añadir el scroll al Panel
        add(scroll);
        setVisible(true);
        pack();
    }
    public static void main(String[] args) {
        new ejemScrollbar();
    }
}
```

}

La ejecución del programa quedaría así:



componentes ligeros

Los componentes ligeros son los elementos adicionales que queremos añadir a un JPanel , JFrame o JScrollPane. Estos nos permiten la introducción de datos, selección y generación de eventos. Todos los componentes ligeros tienen algunos métodos comunes pues heredan de una clase abstracta denominada JComponent.

Class JComponent

```
java.lang.Object
    java.awt.Component
        java.awt.Container
            javax.swing.JComponent
```

Esta clase tiene una serie de métodos comunes para todos los componentes ligeros de Swing. Como se puede apreciar según el Api de JSE 7, JComponent hereda de una clase padre Component perteneciente al paquete java.awt. Por lo que debemos de estar atentos, pues habrá métodos disponibles válidos en el paquete java.awt que podemos utilizar.

Algunos de los métodos comunes a todos los componentes del objeto JComponent son los siguientes:

Métodos	Definición
createToolTip()	Retorna la referencia a JToolTip que debería ser usado para visualizar la tooltip
getAutoscrolls()	Devuelve true o false de la propiedad autoscrolls
getBorder()	Retorna el objeto Border sino tiene null
getMaximumSize()	Devuelve un objeto de tipo Dimension, con las dimensiones del objeto máximas
getMinimumSize()	Devuelve un objeto de tipo Dimension, con las dimensiones del objeto mínimas
getSize()	Devuelve un objeto de tipo Dimension con las dimensiones del objeto actuales.

getWidth()	Retorna el ancho del componente
getX() getY()	Retorna el valor x e y del componente
setAutoScrolls(boolean)	Permite que el elemento muestre barra de desplazamiento
setAlignmentX(float alignmentX)	Modifica el valor X del componente
setAlignmentY(float alignmentY)	Modifiva el valor Y del componente
setBackground(Color bg)	Especifica un color para el fondo del componente
setEnabled(boolean)	Permite activar el componente
setDisable(boolean)	Permite desactivar el componente
setFont (Font Font)	Modifica la fuente del componente

Algunos de los componentes comunes del objeto Container son los siguientes:

Métodos	Definición
Add(PopupMenu)	Añade menus al objeto
addComponentListener(Component Listener)	Añade eventos al elemento
getBackground()	Retorna el color del Objeto
getName()	Retorna el nombre del componente
isVisible()	Devuelve true si está visible y false en caso contrario

tipos de componentes ligeros

A la hora de crear los componentes o elementos en nuestras pantallas debemos de seguir los siguientes pasos:

1. Creación del Componente
2. Modificación de propiedades
3. Adición al Panel, JFrame o ventana
4. Generación de eventos en caso necesario.

JButton

Es un elemento que permite presionarse y que para la generación de funcionalidad cuando se presiona sobre él , debemos de capturar el evento. Algunos métodos concretos de este elemento son los siguientes:

Métodos	Definición
JButton()	Permite crear el Botón
JButton(Action a)	Permite crear el Botón y asociarle un evento
JButton(Icon icon)	Crear el botón con un icono.
JButton(String text)	Crea un botón con texto.
JButton(String text, Icon icon)	Crea un botón con texto e icono.
doClick()	Programáticamente ejecuta un "Click"
getIcon()	Por defecto retorna un icono
getText()	Retorna la etiqueta
setEnabled(boolean)	Activa o desactiva el botón
setText(String)	Genera una etiqueta en botón
setIcon(Icon)	Permite modificar el icono del botón

Creación de un JButton

La creación de un JButton se haría de la siguiente manera:

```
package cap7;

import java.awt.Color;

import javax.swing.JButton;
import javax.swing.JFrame;

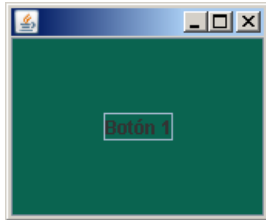
public class EjemComponentes extends JFrame{
    public EjemComponentes(){
        //Creación del componente
        JButton j=new JButton("Botón 1");
        //Modificación de propiedades
        j.setBackground(new Color(10,100,80));
        //Adición al JFrame
        add(j);
        setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

```

    }
}

```

La ejecución del programa es la siguiente:



JTextField y JTextArea

Son componentes ligeros. Derivan de `JTextComponent` con lo que tienen métodos comunes. De este objeto, los métodos que contiene son los siguientes

Métodos	Definición
<code>Copy()</code>	Transfiere la selección al portapapeles
<code>Cut()</code>	Corta el text seleccionado.
<code>getDisabledTextColor()</code>	Devuelve el color del objeto desactivado
<code>getSelectedText()</code>	Retorna el texto seleccionado
<code>getSelectTextColor()</code>	Devuelve el color del texto del componente seleccionado
<code>getText()</code>	Retorna el texto.
<code>Paste()</code>	Devuelve al componente de texto lo cortado o copiado del portapapeles.
<code>setText(String)</code>	Modifica el Texto del componente
<code>setColumns(ancho)</code>	Indica el ancho del componente
<code>setForeground(Color)</code>	Color de texto del componente

JTextField

`JTextField` es un elemento que permite la entrada de datos en una sola línea. Contiene los siguientes métodos disponibles:

Métodos	Definición
---------	------------

JTextField()	Crear un TextField vacío.
JTextField(int columns)	Crea un TextField vacío con un número de columnas concreto
JTextField(String text)	Crea un TextField con el texto por defecto.
JTextField(String text, int columns)	Crea un TextField con un texto y el número de columnas especificado.

Creación de un JTextField

Un ejemplo sería de la siguiente forma:

```
package cap7;

import java.awt.Color;

public class EjemComponentes extends JFrame{
    //Creamos el componente de Texto
    private final JTextField texto = new JTextField();
    public EjemComponentes(){
        //Añadimos el componente al Panel
        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
        getContentPane().add(texto);
        /* Modificamos Propiedades
         *
         */
        //alineación del texto
        texto.setHorizontalAlignment(SwingConstants.LEFT);
        //Crear etiqueta flotante
        texto.setToolTipText("Etiqueta Flotante");
        //Modificar el texto del componente
        texto.setText("Introduce lo que quieras");
        //Color de Fondo del componente
        texto.setForeground(Color.PINK);
        //Ancho de Componente
        texto.setColumns(20);
        setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

La ejecución sería de la siguiente manera:



JTextArea

TextArea es un elemento que permite la entrada de datos en múltiples líneas contiene los siguiente métodos disponibles:

Métodos	Definición
TextArea ()	Crea un TextArea
TextArea (int rows,int columns)	Crea un TextArea con el número de filas y columnas concreto.
TextArea (String text)	Crea un TextArea con el texto por defecto.
TextArea (String text, int rows,int columns)	Crea un TextArea con un texto y el número de filas y columnas concreto.
getColumnns()	Retorna el número de columnas del componente
getRows()	Retorna el número de filas del componente
getColumnnsWidth()	Devuelve el ancho de las columnas del TextArea
getLineCount()	Determina el número de líneas que contiene el área
paramString()	Retorna la cadena que tiene el JTextArea
Setcolumnns(int columns)	Modifica el número de columnas
setRows(int rows)	Modifica el número de filas
setLineWrap(boolean)	Permite generar multilínea en el componente.

Creación de un JTextArea

Un ejemplo de este componente sería el siguiente:

```
package cap7;

import java.awt.Color;

public class EjemComponentes extends JFrame{
    //Crear el JTextArea
    private final JTextArea textArea = new JTextArea();

    public EjemComponentes(){

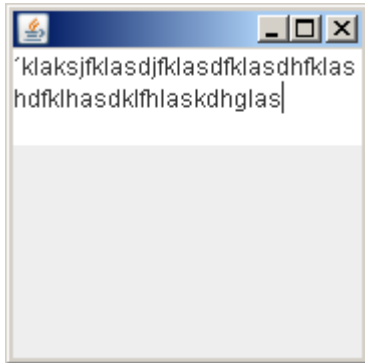
        setVisible(true);
        this.pack();
        /* Modificar propiedades*/
    }
}
```

```

        textArea.setLineWrap(true);
        textArea.setRows(3);
        textArea.setColumns(5);
        //Añadirlo al JFrame
        getContentPane().add(textArea, BorderLayout.NORTH);
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}

```

La ejecución quedaría así:



JLABEL

Es una etiqueta con una línea de Texto.

Métodos	Definición
JLabel()	Crea un JLabel
setHorizontalAlignment(alineación)	Indica la alineación del texto: SwingConstants.LEFT SwingConstants.CENTER SwingConstants.RIGHT
JLabel(Icon image)	Crea un JLabel con un icono
JLabel(Icon image, int horizontalAlignment)	Crea un JLabel con alineación horizontal
JLabel(String text)	Crea un JLabel con texto.
getText()	Retorna el texto de la etiqueta
setText(String)	Modifica el texto de la etiqueta

Creación de un JLabel

Un ejemplo de este componente sería el siguiente:

```
package cap7;

import java.awt.Color;

public class EjemComponentes extends JFrame{

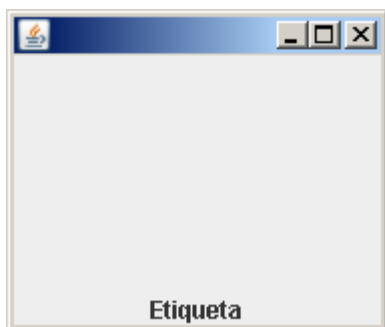
    public EjemComponentes(){

        setVisible(true);
        this.pack();
        /* Modificar propiedades*/

        JLabel lblNewLabel = new JLabel("Etiqueta");
        lblNewLabel.setToolTipText("Un ejemplo");
        lblNewLabel.setHorizontalAlignment(SwingConstants.CENTER);
        getContentPane().add(lblNewLabel, BorderLayout.SOUTH);
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }

}
```

La ejecución quedaría de la siguiente manera:



JList

Es un elemento que permite la selección de algunos de los valores que lo contiene. Para crear un JList se puede de distintas maneras:

- Crear una array de elementos
- Crear el JList y asociarle la caja de lista
- Asociarle el panel

Los posible métodos son los siguientes:

Métodos	Definición
JList()	Crea una caja de lista vacía.
JList(E[] listaData)	Crear un Jlist con los valores expuesto en el array
JList(Vector listData)	Crea un vector de elementos de caja de lista

getMaxSelectionIndex()	Retorna el índice de la máxima selección o -1 si no tiene valores.
getModel()	Retorna la lista que contiene la caja de lista
getSeleccionBackground()	Retorna el color de los valores seleccionados
paramString()	Retorna los valores en un String de la caja de lista

Creación de un JList

Un ejemplo de este componente:

```
package cap7;

import java.awt.Color;

public class EjemComponentes extends JFrame{
    public EjemComponentes(){
        //Se crea un array con los valores
        String[] values = new String[] {"uno", "dos", "tres"};
        //Se asocia el array al JList
        JList list=new JList(values);
        list.setValueIsAdjusting(true);
        /* Se modifican propiedades */
        //Seleccionar un elemento del JList indicando el índice de la posición
        list.setSelectedIndex(1);
        //Número de filas visible que queremos
        list.setVisibleRowCount(1);
        //Modificar el tipo de selección e la caja de lista
        list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
        getContentPane().add(list, BorderLayout.NORTH);
        this.setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

La ejecución del ejemplo quedaría así:



Otra posibilidad es crear un JList basado en una clase llamada DefaultListModel, que contiene elementos para añadir, borrar o modificar un JList. Esta clase tiene los siguientes métodos específicos. La utilización de esta clase es de la siguiente manera:

- Crear un objeto vacío
- Añade los elementos
- Asocia el Objeto DefaultListModel al JList

Métodos	Definición
DefaultListModel	Crea una lista vacía
Add(int pos, element)	Añade un elemento a esa posición
addElement(element)	Añade un elemento independiente de la posición
Clear	Borra los elementos
Elements	Retorna un Enumeration con todos los elementos
firstElements	Retorna el primer elemento de la lista
Set(index,element)	Modifica un elemento en esa posición
Size()	Devuelve el número de elementos que tiene

Creación de un JList II

Un ejemplo de esta utilización sería la siguiente:

```
public class EjemComponentes extends JFrame{

    public EjemComponentes(){
        //Intancio la clase
        DefaultListModel listModel = new DefaultListModel();
        //Utilizo los métodos de adición de componentes
        listModel.addElement("Jane Doe");
        listModel.addElement("John Smith");
        listModel.addElement("Kathy Green");
        //creo la lista y le asocio el DefaultListModel
        JList list = new JList(listModel);
        //Asocio la lista al panel
        getContentPane().add(list, BorderLayout.NORTH);
        this.add(list);
        this.setVisible(true);
        this.pack();
    }

    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

La visualización quedaría de la siguiente manera:



JCOMBOX

Son elementos parecidos a las JList, con la diferencia que en vez de ser un elemento donde los datos se representa en cada fila tiene sola una fila. Son los más habituales.

Los métodos más habituales son los siguientes:

Métodos	Definición
JComboBox()	Crea una caja de lista desplegable vacía.
JComboBox(ComboBoxModel)	Crear una caja de lista con la clase que implementa los métodos adicionales de crear, borrar y modificar elementos
JComboBox(Object[])	Crear una caja de lista con elementos desplegables basados en un objeto.
addItem(object)	Inserta un objeto en la caja de lista
removeItem(Object) removeAllItem()	Borra un elemento o bien, borra todos los elementos de la caja de lista.

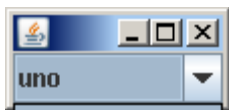
Creación de un JComboBox

Un ejemplo de este componente sería:

```
public class EjemComponentes extends JFrame{

    public EjemComponentes(){
        //Creación del componente
        JComboBox comboBox = new JComboBox();
        //Añadir los valores al componente
        comboBox.setModel(new DefaultComboBoxModel(new String[] { "uno", "dos",
        "tres" }));
        //Asociarlo al Panel
        getContentPane().add(comboBox, BorderLayout.NORTH);
        this.setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

La ejecución de este componente quedaría así:



JCHECKBOX

Es una caja de chequeo permite seleccionar y deseleccionar un valor. Los posibles métodos disponibles son:

Métodos	Definición
JCheckbox()	Crea una caja de chequeo
JCheckBox(Icon)	Crea la caja de chequeo con un icono
JCheckBox(texto, boolean)	Crea la caja con texto y seleccionado si es true y false en caso contrario.
JCheckBox(texto, icon)	Crea la caja de chequeo con texto e icono
getFont	Regresa la fuente que tenga el componente
getHeight	Regresa la altura del componente en píxeles
getWidth	Regresa la anchura del componente en píxeles
getName	Regresa el nombre del objeto
getText	Igual que getActionCommand
getX	Devuelve la posición en el eje X del componente
getY	Devuelve la posición en el eje Y del componente
isEnabled	Devuelve true si el componente esta activo, en caso contrario, devuelve false
isFocusable	Devuelve true si el componente puede recibir el "foco", en caso contrario devuelve false
isSelected	Devuelve true si el componente esta seleccionado, en caso contrario devuelve false
isShowing	Devuelve true si el componente se puede ver en pantalla , en caso contrario devuelve false

isValidate	Devuelve true si el componente es válido , en caso contrario devuelve false
isVisible	Devuelve true si el componente esta visible, en caso contrario devuelve false

CREACIÓN DE UN JCHECKBOX

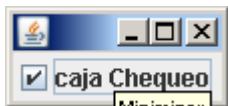
A continuación se detalla un programa de utilización de este componente.

```
package cap7;

import java.awt.Color;

public class EjemComponentes extends JFrame{
    public EjemComponentes(){
        //Se crea la caja de chequeo
        JCheckBox caja = new JCheckBox("caja Chequeo");
        //Se modifican propiedades. Marcada con el valor de true
        caja.setSelected(true);
        //Se añade al panel
        getContentPane().add(caja, BorderLayout.SOUTH);
        this.setVisible(true);
        this.pack();
    }
    public static void main(String args[]){
        EjemComponentes ej=new EjemComponentes();
    }
}
```

La ejecución quedaría así:



JCOLORCHOOSER

Es un componente que contiene una paleta de colores . Sobre este componente se puede seleccionar un color y asociarlo a algún componente de Swing. Los métodos más utilizados son:

Métodos	Definición
showDialog(panel,"text",color)	Muestra la paleta de colores con un texto y un color predeterminado.Devuelve un objeto con el color elegido

CREACIÓN DE UN JCOLORCHOOSER

A continuación se detalla un programa de utilización de este componente.

```
public class ColorElegir extends JFrame {

    private Color color=Color.LIGHT_GRAY;
    public ColorElegir(){
```



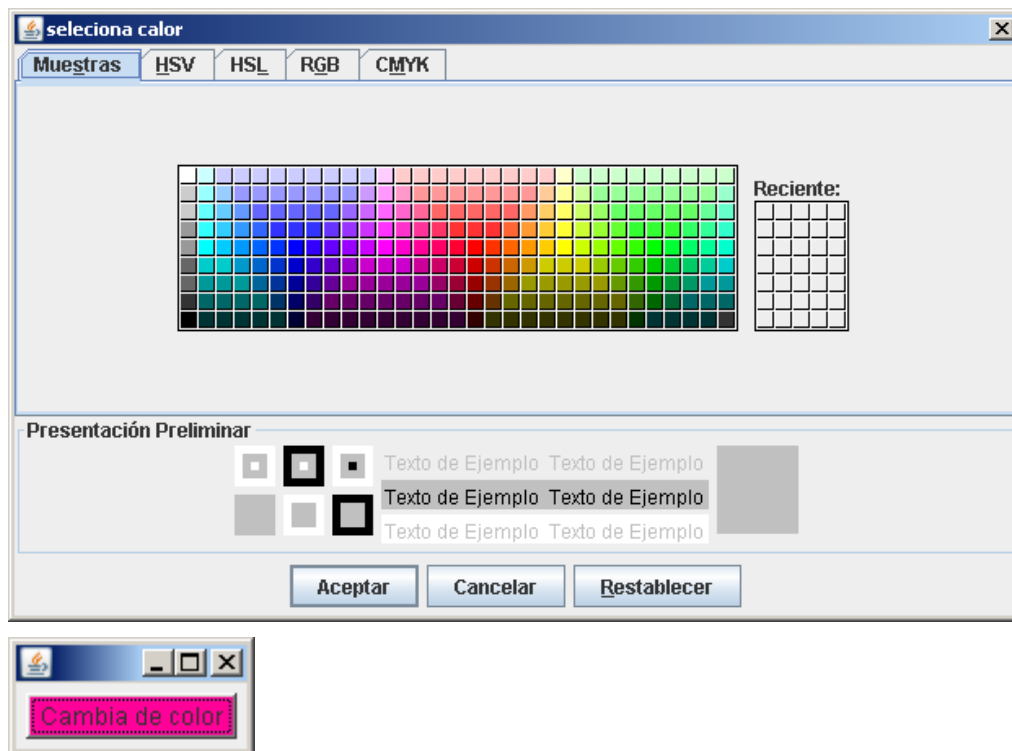
```

this.setLayout(new FlowLayout());
//Mostramos la paleta de colores con un color determiando

color=JColorChooser.showDialog(this, "selecciona calor", color);
//Creamos un boton
Button b=new Button("Cambia de color");
this.add(b);
//Cambiamos al botón el color con la selección elegida
b.setBackground(color);
this.setVisible(true);
this.pack();
}
public static void main(String args[]){
    ColorElegir c=new ColorElegir();
}
}

```

La ejecución quedaría de la siguiente manera:



Gestión de eventos

Cada vez que el usuario realiza una acción, es decir, mueve el ratón, pulsa un botón o una tecla, ocurre un evento. Para poder tratar un evento, es necesario seguir los siguientes pasos:

- Se tiene que crear un manejador, capaz de tratar el evento.
- Registrar el evento para que el sistema sepa a que manejador tiene que enviarlo.

La máquina virtual de Java crea el evento automáticamente. Este evento creado, se hereda de la clase padre `java.awt.Event`.

Para poder capturar los eventos se debe de seguir los siguientes pasos:

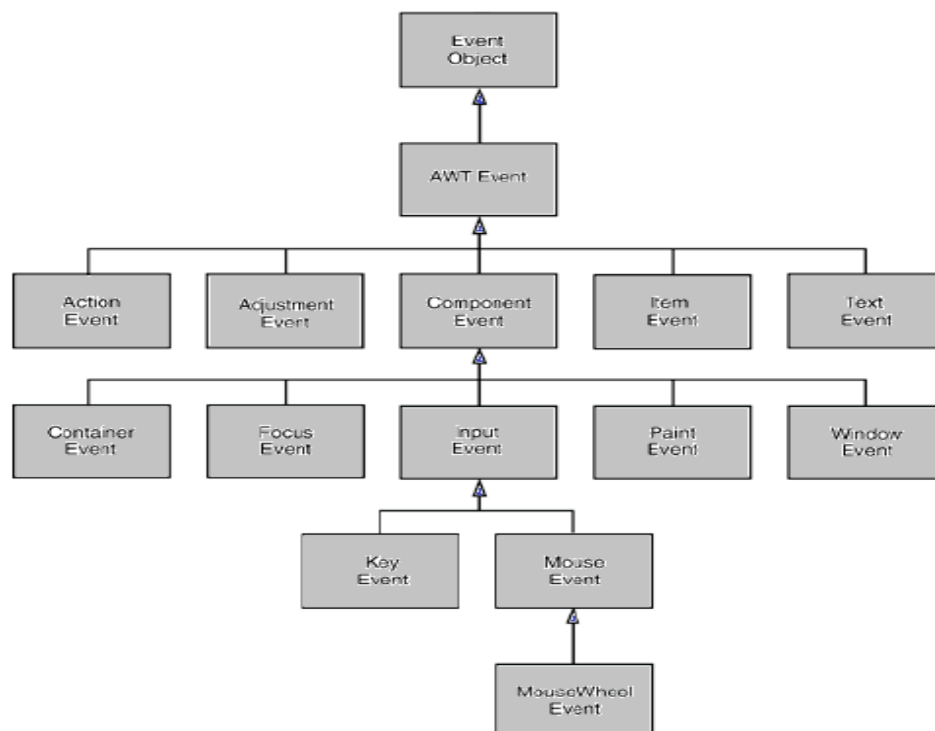
Crear el componente que genera el evento

Crear las clases “oyentes” que van a capturar el evento y desarrollar la funcionalidad.

Asociar estas clases a los componentes

Creación de eventos

Para crear la clase “Oyente” que captura el evento esta clase debe de implementar una interfaz que será la que tendrá los métodos encargados de capturar los eventos. A continuación se muestra la jerarquía de objetos.



Para capturar un evento se puede hacer de dos maneras:

A través de una interfaz

A través de una clase adaptadora

Interfaces o listener

Las interfaces son la que disponen de los métodos que contienen los Eventos que yo quiero capturar cuando se produce una acción sobre un componente. El inconveniente de utilizar las interfaces es que se deben implementar todos los métodos. Existen distintas interfaces en función del evento que yo quiero capturar. Son las siguientes:

Interface	Métodos	Definición
ActionListener	actionPerformed	<ul style="list-style-type: none"> • Se ejecuta cuando se realiza una acción sobre un elemento como: • pulsar un botón. • doble clic en un elemento de lista. • INTRO en una caja de texto. • Elegir un menú.
CaretListener	caretUpdate	Se cambia de posición un elemento
ChangeListener	stateChanged	Cuando cambia de estado un elemento
ComponentListener	componentHidden componentMoved componentResized componentShown	Se mueven los componentes
ContainerListener	componentAdded componentRemoved	Se añaden o eliminan componentes
FocusListener	focusGained focusLost	Cuando un componente coge el foco o lo pierde.
InternalFrameListener	internalFrameActivated internalFrameClosed internalFrameClosing internalFrameDeactivated internalFrameDeiconified	Estos eventos están asociados al Frame: <ul style="list-style-type: none"> • Cuando se activa • Se cierra

	internalFrameIconified internalFrameOpened	<ul style="list-style-type: none"> • Se desactiva • Se abre
--	---	---

Interface	Métodos	Definición
ItemListener	itemStateChanged	Cambia de estado el elemento
KeyListener	keyPressed keyReleased keyTyped	Se realiza una acción de teclado
MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	Se realiza una acción con el ratón tales como presionar, click
MouseMotionListener	mouseDragged mouseMoved	Arrastrar el ratón o mover
WindowListener	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified	Realizar acciones sobre la ventana

	windowOpened	
--	--------------	--

Clases adaptadoras

La clase adaptadoras es una clase que ya implementa automáticamente la interfaz y que por lo tanto tiene implementado estos métodos. Existen distintos tipos de Clases adaptadoras en función de los métodos que queramos utilizar.

Interface	Clase Adaptadora
ActionListener	ActionListener
CaretListener	ninguna
ChangeListener	ninguna
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
DocumentListener	ninguna
FocusListener	FocusAdapter
InternalFrameListener	InternalFrameAdapter
ItemListener	ninguna
KeyListener	KeyAdapter
ListSelectionListener	ninguna
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
UndoableEditListener	ninguna
WindowListener	WindowAdapter

EVENTOS

Los métodos que contiene las interfaces o clases Adaptadoras reciben el objeto que ha generado el componente como evento. Este objeto, contiene un método denominado `getSource()` que permite recoger una referencia al objeto que ha generado el evento.

IMPLEMENTACIÓN DE LISTENER Y CLASE ADAPTADORA

Al implementar estas clases, debemos de tener en cuenta, que la clase debe de implementar la interfaz y redefinir los métodos además de añadirlo al componente adecuado. Los componentes tienen métodos `addNombreListener(new ClaseAdaptadora o new Listener)` que permite asociar al componente el listener adecuado. Es posible a la hora de asociar el evento que queremos capturar al componente, hacerlo de varias maneras:

- Creando una clase independiente o dentro del método `addEventoListener`
- Crear la clase dentro del método `addEventoListener` del componente.

IMPLEMENTACIÓN DE UN LISTENER EN UNA CLASE INDEPENDIENTE

Si creo un Listener a través de una clase independiente, tendremos la posibilidad de tener varios objetos asociados a esta clase. Para ello debemos:

- Crear una clase que herede de la interfaz que contiene los eventos que queremos capturar y redefinir los métodos

```
package cap7;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class PresionaButon1 implements ActionListener{
    @Override
    public void actionPerformed(ActionEvent e) {
        //Obtengo la referencia del objeto que ha provocado el evento
        JButton boton=(JButton) e.getSource();
        //Modifico la propiedad de ese elemento cuando se genera ese evento
        boton.setText("evento generado");
    }
}
```

Asociar el Listener a nuestro componente:

```
package cap7;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Eventos extends JFrame{
    public Eventos(){
        JButton b=new JButton("evento1");
        //Asociamos nuestro evento al componente
        b.addActionListener(new PresionaButon1());
        this.add(b);
        this.setVisible(true);
        pack();
    }
    public static void main(String args[]){
        Eventos e=new Eventos();
    }
}
```

IMPLEMENTACIÓN DE UNA CLASE ADAPTADORA

Normalmente la realización de este tipo de “oyentes” o clases escuchadoras es para no tener que crear clases independientes. Normalmente este tipo de clases se crean dentro del método del objeto al cual le queremos asociar `addEventListener`, de tal manera que tenemos referencia a todos nuestros objetos.

```
package cap7;

import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class Eventos extends JFrame{
    public Eventos(){

        final JButton btnEvento = new JButton("evento2");
        //En el método de asociación del listener que contiene los métodos
        correspondientes
        //A la captura de eventos del ratón creo la clase Adaptadora.
        btnEvento.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                btnEvento.setText("click con el ratón");
            }
        });
        getContentPane().add(btnEvento, BorderLayout.NORTH);
        this.setVisible(true);
        pack();

    }
    public static void main(String args[]){
        Eventos e=new Eventos();
    }
}
```

Entrada y Salida estándar

TABLA DE CONTENIDOS

Introducción a los Elementos Básicos del Api.....	1
Flujos de datos	1
Nodos	1
Filtros.....	1
Clases básicas flujos de bytes.....	2
InputStream	2
OutputStream	2
Clases Básicas Flujos de Caracteres	3
Reader	3
Writer	3
Flujos predefinidos.....	5
Leer Entradas en la Consola	5
Un ejemplo de lectura de bytes por pantalla.....	5
Escribir Entradas en la Consola.....	6
Tipos de Nodos Disponibles	7
Leer y escribir archivos en flujos de bytes	7
Piped.....	8
Tipos de Filtros Disponibles.....	12
Utilización de filtros en entrada y salida.....	12
Utilización del filtro BufferedReader	12
Serialización de Objetos	13
La Clase File.....	15
Archivos de acceso aleatorio	17

Introducción a los Elementos Básicos del Api

Todos los programas que trabajan con Entrada y Salida maneja esta funcionalidad a través de flujos.

Los flujos de datos (Stream) son una fuente o sumidero de bytes secuencial con una dirección específica. Un flujo se vincula un dispositivo físico a través del sistema de entrada y salida de Java. Java implementa los flujos en las jerarquías de clases definidas en el paquete java.io.

Flujos de datos

Las direcciones posibles de un stream pueden ser:

- Entrada: Se puede leer del stream.
- Salida: Sirven para leer del stream.
- Las versiones actualizadas de Java definen dos tipos de flujos: Bytes y caracteres, esto da lugar a cuatro clases básicas en el paquete java.io.

	Streams Bytes	Streams caracteres
Entrada(Lectura)	<i>InputStream</i>	<i>Reader</i>
Salida(Escritura)	<i>OutputStream</i>	<i>Writer</i>

Dentro del paquete java.io existen muchas clases derivadas de las cuatro anteriores que son específicas para manejar distintos elementos que pueden considerarse como nodos o filtros.

Nodos

Representa una fuente o sumidero de datos producido por algún elemento concreto. Un ejemplo sería la memoria, un archivo o un socket.

Filtros

Se crean a partir de un Stream de tipo nodo y su misión consiste en alterar o dar mayor funcionalidad a los nodos

Clases básicas flujos de bytes

Existen dos clases básicas como hemos comentado antes, que proporcionan métodos para la manipulación básica de un flujo de bytes: InputStream y OutputStream

InputStream

Proporciona métodos necesarios para la lectura de bytes. Sus métodos son los siguientes:

Métodos	Definición
int read(byte[] b)	<i>Intenta rellenar el array de bytes pasado como argumento y devuelve un entero indicando el número de bytes leídos. Si no hay datos para leer devuelve</i>
int read(byte[] b, int off, int len)	<i>Intenta rellenar la porción indicada del array de bytes. Devuelve el número de bytes leídos</i>
int available()	<i>Disponibilidad de bytes</i>
void skip(long)	<i>Descarte de bytes</i>
void close()	<i>Cierre del Stream</i>

OutputStream

La clase OutputStream proporciona los métodos para la manipulación básica de los datos hacia un flujo de bytes de salida. Dichos métodos tienen la siguiente funcionalidad:

Métodos	Definición
void write(byte[] b, int off, int len)	<i>Escritura de bytes en un array por posición y longitud</i>
abstract void write(int b)	<i>Escritura de byte a byte</i>
void write(byte[] b)	<i>Escritura de un array de bytes</i>
void flush()	<i>Forzado de escritura</i>
void close()	<i>Cierre del Stream</i>

Clases Básicas Flujos de Caracteres

Existen dos clases básicas como hemos comentado antes, que proporcionan métodos para la manipulación básica de un flujo de caracteres Reader y Writer

Reader

La clase Reader representa un flujo de caracteres de entrada. Los métodos que aporta esta clase son:

Métodos	Definición
Int read()	<i>Lectura carácter a carácter</i>
Int read(char[])	<i>Lectura de array de caracteres</i>
Boolean ready()	<i>Disponibilidad de caracteres</i>
Void skip(long numbytes)	<i>Ignora el número de bytes indicado y los devuelve.</i>
Boolean mark(int numbytes)	<i>Añade una marca al punto actual del flujo de entrada hasta que se lea numbytes</i>
Void reset()	<i>Restablece el puntero de entrada a la marca definida previamente</i>
Void close()	<i>Cierre del stream</i>

Writer

Esta clase representa un sumidero de caracteres y aporta los siguientes métodos para la escritura:

Métodos	Definición
Void write(char)	<i>Escribe carácter a carácter</i>
Void write (char[])	<i>Escritura de array de caracteres completo</i>
Void write (char[],int,int)	<i>Disponibilidad de caracteres</i>
Void write (String)	<i>Ignora el número de bytes indicado y los devuelve.</i>
Void write(String ,int,int)	<i>Escribe un subintervalo de número de caracteres desde el</i>

	<i>buffer de la matriz.</i>
Void flush()	<i>Vacía el buffer de salida</i>
Void close()	<i>Cierre del stream</i>

Flujos predefinidos

El paquete `java.lang` es importado automáticamente por todas las aplicaciones Java. De este paquete tenemos la clase `System` que encapsula distintos aspectos del entorno de ejecución. De esta clase podemos destacar:

- `System.out`: Hace referencia a la salida estándar, que por defecto, es la consola.
- `System.in`: Hace referencia a la entrada estándar, que por defecto, es el teclado. Es una instancia de `InputStream`
- Todos ellos son flujos de bytes, aunque suele usarse para leer y escribir caracteres en y desde la consola.

Leer Entradas en la Consola

Originalmente la única forma que teníamos para poder leer desde la consola era a través de flujos de bytes. Afortunadamente esto ha cambiado y tenemos disponibles, también por comodidad la lectura de caracteres. Por defecto, y debido que implementa la clase `InputStream` tiene acceso automático a estos métodos:

Métodos	Definición
<code>int read()</code>	<i>Escribe carácter a carácter</i>
<code>int read(byte[] datos)</code>	<i>Escritura de array de caracteres completo</i>

Al ser métodos estáticos estos métodos pueden invocarse directamente con `System.in.read`. El método `read`, en cuestión, espera a que el usuario pulse una tecla y luego devuelve el resultado. El carácter lo devuelve como entero, por lo que hay que convertirlo a una variable `char`.

Un ejemplo de lectura de bytes por pantalla

En este ejemplo se muestra como a partir de un array de bytes leemos los datos introducidos por pantalla y los mostramos:

```
import java.io.IOException;

public class Ejercicio1 {

    public static void main(String[] args) throws IOException {
        byte datos[]=new byte[10];
        System.out.println("Introduce datos:");
        //lee una matriz de bytes desde el teclado.Lanza excepción
        System.in.read(datos);
        //leemos los datos almacenamos en el array de byte
        for (int i =0;i<datos.length;i++){
            System.out.print((char)(datos[i]));
        }

    }

}
```


Escribir Entradas en la Consola

Como se ha nombrado anteriormente, Java en sus orígenes solo permitía flujos de bytes aunque en la versión 1.1 ya añadió los flujos de caracteres. Con `System.out` se puede enviar datos a la consola basada en bytes con `write`. El formato más sencillo de `Write` es el siguiente:

Métodos	Definición
<ul style="list-style-type: none">• <code>Write(int valorbyte)</code>	<i>Escribe el byte especificado en <code>valorbyte</code>.</i>

No es normal usar este método, pues para la salida por consola es más habitual utilizar `Print()` y `Println()` que son más fáciles de usar y se definen en la clase `PrintStream` que hereda de `OutputStream`.

Tipos de Nodos Disponibles

Para comenzar la programación de entrada y salida es necesario indicar cual es el nodo o dispositivo sobre el que se va a trabajar esta entrada y salida. Estos nodos en función de lo que necesitamos, si leer o escribir, heredan de los flujos de bytes o Caracteres especificados anteriormente. Tenemos las siguientes clases disponibles:

Nodos	Stream de bytes	Stream de Caracteres
Ficheros	<i>FileInputStream</i> <i>FileOutputStream</i>	<i>FileReader</i> <i>FileWriter</i>
Arrays	<i>ByteArrayInputStream</i> <i>ByteArrayOutputStream</i>	<i>CharArrayReader</i> <i>CharArrayWriter</i>
String		<i>StringReader</i> <i>StringWriter</i>
Pipes	<i>PipedInputStream</i> <i>PipedOutputStream</i>	<i>PipedReader</i> <i>PipedWriter</i>

Por lo tanto, para comenzar la programación de un proceso de entrada o salida , será necesario crear un objeto de algunas de las clases ,elegida la función del tipo de dispositivo y si está orientada a bytes o Caracteres.

Leer y escribir archivos en flujos de bytes

Si queremos crear un flujo de byte vinculado a un archivo, debe de usarse *FileInputStream* o *FileOutputStream*. Para abrir un archivo, basta con crear un objeto de una de estas clases y especificar el nombre del archivo como argumento del constructor.

Para la generación de este tipo de nodos hay que seguir los siguientes pasos si queremos leer:

1. *FileInputStream*(String nombreArchivos) throws *FileNotFoundException*
2. Para leer del archivo utilizaremos *read*.
3. Controlar las excepciones

Para escribir en un fichero de bytes los pasos serían los siguiente:

4. *FileOutputStream*(String nombreArchivo) Throws *FileNotFoundException*, abrimos el fichero en modo escritura
5. *Write*(int valbyte) .Escribe el byte especificado.
6. *Close*() . Se cerrarían todos los recursos disponibles.

A continuación se muestra un ejemplo de una clase que escribe y lee en un fichero de bytes.

```
public class ES_bytes {
```

```

FileOutputStream fout;
FileInputStream fin;
String fichero;
public ES_bytes(String fichero,String cadena) throws IOException{
    this.fichero=fichero;
    //Abre en modo bytes para escritura
    fout=new FileOutputStream(fichero);
    byte arra[]=cadena.getBytes();
    //Se escribe en el fichero un array de bytes de la cadena
    introducida
        fout.write(arra);
        fout.close();

}
public void lectura() throws IOException{
    //Abre en modo bytes para lectura
    fin=new FileInputStream(fichero);
    int i=0;
    System.out.println("Contenido del fichero");
do{
    //Lee byte a byte mientras i no de -1, quiere decir final del fichero.
    i=fin.read();
    System.out.print((char)i);
}while(i!=-1);
fin.close();

}

    public static void main(String[] args) throws IOException {
        ES_bytes ej=new ES_bytes("c:/temp/mio.text","esto es una prueba");
        ej.lectura();

    }

}

```

Piped

Los Objetos Piped, denominados tuberías, permiten que diferentes procesos se comuniquen entre sí, permitiéndoles a su vez la compartición de datos. Normalmente se utilizan para que los datos que manipulan un Thread , estén accesibles desde otro Thread.

Las tuberías para trabajo con caracteres son PipedReader y PipedWriter. Funcionan de la misma forma que sus correspondientes versiones de 8 bits.

Métodos de PipedReader:

Métodos	Definición
Ready()	<i>Indica si el stream está listo para ser leído.</i>
Close()	<i>Cierra el PipedReader y libera los recursos asociados.</i>
connect(PipedWriter src)	<i>Conecta esta tubería a la tubería de escritura pasada como parámetro en la llamada la método.</i>

<code>Read()</code>	<i>Lee el siguiente carácter del stream. Devuelve el carácter leído o -1 cuando se llega al final del stream.</i>
<code>read(char[] cbuf, int off, int len)</code>	<i>Lee en bytes de datos y los pone en la posición indicada por off dentro del array. Devuelve -1 cuando se llega al final del stream o el número de caracteres leídos.</i>

Métodos de `PipedWriter`:

Métodos	Definición
<code>connect(PipedReader snk)</code>	<i>Conecta esta tubería a una tubería receptora de datos.</i>
<code>Close()</code>	<i>Cierra el <code>PipedReader</code> y libera los recursos asociados.</i>
<code>Flush()</code>	<i>Fuerza la escritura de los caracteres</i>
<code>write(char[] cbuf, int off, int len)</code>	<i>Escribe len caracteres de datos empezando en la posición off del array.</i>
<code>Write(int c)</code>	<i>Escribe el carácter pasado como parámetro por el stream.</i>

Obsérvese el siguiente ejemplo:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PipedReader;
import java.io.PipedWriter;
import java.io.Reader;
import java.io.Writer;

import cap6.Util;

public class Pipes {
    static class Escritor implements Runnable {
        BufferedWriter writer = null;
```

```

        BufferedReader reader;

        Escritor(Writer writer, String fileName)
            throws FileNotFoundException {
            this.writer = new BufferedWriter(writer);
            reader = new BufferedReader(new FileReader(fileName));
        }

        public void run() {
            while (true) {
                String line;
                try {
                    line = reader.readLine();
                    if (line == null) {
                        writer.close();
                        return;
                    }
                    writer.write(line + "\n");
                    writer.flush();
                }
                catch (IOException e) {
                    e.printStackTrace();
                    return;
                }
            }
        }
    }

    static class Lector implements Runnable {
        Reader reader = null;

        Lector(Reader reader) {
            this.reader = reader;
        }

        public void run() {
            int c;
            BufferedReader buffer = new BufferedReader( reader );

            try {
                String line;
                while( ( line = buffer.readLine() ) != null) {
                    System.out.println(line);
                }
            }
            catch (IOException e) {
                System.exit(1);
            }
        }
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            Util.failn("Hay que pasar un nombre de fichero");
        }
        try {
            PipedReader r = new PipedReader();
            PipedWriter w = new PipedWriter(r);

            new Thread(new Lector(r)).start();
            new Thread(new Escritor( w, args[0])).start();
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}

```

En el ejemplo se observa como en el método main se crea un PipedReader y seguidamente un PipedWriter. En la llamada al constructor de éste, se pasa como parámetro el PipedReader para conectar ambos streams.

La idea de este programa es mostrar por pantalla el contenido de un fichero. En vez de utilizar un solo hilo de ejecución leyendo del fichero y escribiendo a la pantalla creamos dos hilos para que puedan trabajar en paralelo. Uno de ellos (Escritor) lee del fichero y lo envía al Pipe de escritura y otro (Lector) lee del Pipe de lectura y lo escribe en pantalla. De este modo, los dos threads pueden trabajar en paralelo sin tener que esperarse.

Tipos de Filtros Disponibles

Como se ha apreciado anteriormente, el tratamiento de lectura y escritura con los métodos que proporcionan los nodos son muy básicos y costosos. Por esta razón y para implementar mayor funcionalidad se crean los filtros. Los filtros siempre irán asociado a un nodo, como podemos apreciar posteriormente. En la siguiente tabla se muestran los filtros disponibles:

Stream de bytes	Stream de caracteres	Descripción
BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter	<i>Mejoran el rendimiento de entrada y salida.</i>
	InputStreamReader OutputStreamWriter	<i>Conversión de stream de bytes a caracteres</i>
ObjectInputStream ObjectOutputStream		<i>Permiten la serialización de objetos</i>
DataInputStream DataOutputStream		<i>Funcionalidad para el trabajo con tipos simples</i>
LineNumberInputStream	LineNumberReader	<i>Contiene métodos para el conteo de líneas</i>
PushbackInputStream	PushBackReader	<i>Amplía funcionalidad para el retroceso de lectura</i>
PrintStream	PrintWriter	<i>Funcionalidad para la impresión por consola.</i>

Utilización de filtros en entrada y salida

Para poder utilizar los filtros debemos de implementar las siguientes fases:

- Creación de un nodo de entrada o salida
- Creación de un filtro de entrada o salida al cual se le pasará como argumento la referencia del nodo anterior.

```
FileReader fi=new FileReader("fichero.txt");
BufferedReader br=new BufferedReader(fi);
```

Utilización del filtro BufferedReader

En ejemplos anteriores, la lectura de caracteres tenía que ser carácter a carácter. Esta clase contiene métodos para una lectura más rápida. Ver el siguiente ejemplo.

```
public class Ejercicio_Buffered {
    FileWriter fout;
    FileReader fin;
    String fichero;
    public void Escritura(String fichero,String cadena) throws IOException{
```

```
        this.fichero=fichero;
//Abre en modo bytes para escritura
        fout=new FileWriter(fichero);
        fout.write(cadena);
        fout.close();

    }
    public void Lectura() throws IOException{
        FileReader fin=new FileReader(fichero);
        String salida;
//Utilizo un filtro de mejora de rendimiento.
//Pues la clase FileReader no proporciona métodos rápidos para lectura de
//caracteres.
//Por defecto , FileReader lee carácter a carácter.
//BufferedReader proporciona el método readLine que lee líneas del fichero.
        BufferedReader buf=new BufferedReader(fin);
        //Mientras la línea no sea nulo lee e imprime por pantalla
        while((salida=buf.readLine())!=null){
            System.out.print(salida);
        }
        buf.close();
        fin.close();
    }

    public static void main(String[] args) throws IOException {
        Ejercicio_Buffered ej=new Ejercicio_Buffered();
        ej.Escritura("c:/temp/fichero.txt", "Ejemplo filtros...");
        ej.Lectura();
    }
}
```

Serialización de Objetos

Se llama serialización de objetos cuando un objeto es persistente, es decir sobrevive a la aplicación que lo creó.

Estos Objetos se quedan persistentes cuando se almacenan en un Stream de bytes y a este proceso se le denomina serialización.

Para poder serializar objetos, Java proporciona una interfaz en `java.io.Serializable`, por lo que las clases deberán de implementar esta interfaz. Su implementación no implica la redefinición de ningún método.

La manipulación de los objetos serializados se hace a través de los filtros `ObjectInputStream` y `ObjectOutputStream`, ambas pertenecen al paquete `java.io`.

Por lo tanto , para poder Serializar un Objeto hay que realizar los siguiente pasos:

- Implementar la interfaz `Serializable`
- Crear una instancia de las clases `ObjectInputStream` u `ObjectOutputStream`
- Utilizar los métodos correspondientes a estas clases para la entrada y salida `writeObject` y `readObject`.

A continuación vamos a Serializar un objeto de tipo `String`:

```
public class ES_Serializacion {
    FileOutputStream fout;
```



```
FileInputStream fin;
String fichero;
public ES_bytes(String fichero,String cadena) throws IOException{
    this.fichero=fichero;
    //Abre en modo bytes para escritura
    fout=new FileOutputStream(fichero);
    ObjectOutputStream out=new ObjectOutputStream(fout);

    //Se guarda todo el objeto String
    out.writeObject(cadena);
    out.close();
}
public void lectura() throws IOException, ClassNotFoundException{
    //Abre en modo bytes para lectura
    fin=new FileInputStream(fichero);
    ObjectInputStream in =new ObjectInputStream(fin);
    //Se lee todo el objeto String almacenado
    System.out.println("Contenido del fichero :" + in.readObject());
    in.close();
}

    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        ES_bytes ej=new ES_bytes("c:/temp/mio.text","esto es una prueba");
        ej.lectura();
    }
}
```

La Clase File

La clase File permite obtener información acerca de un fichero o de un directorio. Uno de sus objetivos es esconder al programador de las distintas convenciones de nombrado de ficheros que tienen los sistemas operativos. Destacan los siguientes métodos:

Métodos	Definición
renameTo(File f)	<i>Permite cambiar el nombre del fichero, incluso cambiarlo de directorio.</i>
mkdir()	<i>crear un nuevo directorio</i>
list()	<i>En caso que este objeto represente a un directorio, lista su contenido.</i>
length()	<i>Devuelve el tamaño del fichero.</i>
lastModified()	<i>Devuelve el momento de la última modificación realizada en este fichero.</i>
isFile()	<i>Indica si es un fichero o no.</i>
isDirectory()	<i>Indica si es o no un directorio.</i>
getName()	<i>Devuelve el nombre del fichero.</i>
canRead()	<i>Indica si se tienen permisos para leer el fichero</i>
canWrite()	<i>Indica si se tienen permisos para escribir el fichero</i>
delete()	<i>Intenta eliminar el fichero representado por este objeto File.</i>
exists()	<i>Comprueba si el fichero existe realmente.</i>
getAbsolutePath()	<i>Devuelve la ruta absoluta del fichero.</i>

getCanonicalPath	<i>Devuelve el camino canónico de la ruta.</i>
------------------	--

Con el objeto File no se realiza lectura o escritura de información. Obsérvese el siguiente ejemplo.

```
package cap4;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class Espia {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            File f = new File(args[i]);
            if (f.exists() == false) {
                System.out.println("No puedo encontrar el fichero " +
                    args[i]);
                continue;
            }

            System.out.println("Nombre: " + f.getName());
            System.out.println("Camino absoluto: " +
                f.getAbsolutePath());
            try {
                System.out.println("Camino canónico: " +
                    f.getCanonicalPath());
            }
            catch (IOException e) {
                System.out.println
                    ("No pude determinar el camino canónico.");
            }

            String parent = f.getParent();
            if (parent != null) {
                System.out.println("Padre: " + f.getParent());
            }

            if (f.canWrite())
                System.out.println(f.getName() +
                    " se puede escribir.");

            if (f.canRead())
                System.out.println(f.getName() + " se puede leer.");

            if (f.isFile()) {
                System.out.println(f.getName() +
                    " es un fichero normal.");
            }
            else if (f.isDirectory()) {
                System.out.println(f.getName() +
                    " es un directorio. Su contenido es:");
                String[] list = f.list();
                System.out.print("\t");
                for (int j = 0; j < list.length; j++) {
                    System.out.print(list[j]);
                    if (j < list.length-1) {
                        System.out.print(", ");
                    }
                }
                System.out.println();
            }
        }
    }
}
```

```
    }
    else {
        System.out.println("qué es esto?");
    }

    if (f.isAbsolute()) {
        System.out.println(f.getPath() +
            " es un nombre absoluto.");
    }
    else {
        System.out.println(f.getPath() +
            " no es un nombre absoluto.");
    }

    long lm = f.lastModified();
    if (lm != 0)
        System.out.println("Última fecha de modificación " +
            new Date(lm));

    long length = f.length();
    if (length != 0) {
        System.out.println(f.getName() + " tiene " +
            length + " bytes.");
    }
}
}
```

En este ejemplo se imprime información de los ficheros que se le pasan como argumento. Una posible salida de este programa sería:

```
C:\java2\tema4>java cap4.Espia Espia.java
Nombre: Espia.java
Camino absoluto: C:\java2\tema4\Espia.java
Camino canónico: C:\java2\tema4\Espia.java
Espia.java se puede escribir.
Espia.java se puede leer.
Espia.java es un fichero normal.
Espia.java no es un nombre absoluto.
+ltima fecha de modificaci3n Fri Sep 26 13:40:53 CEST 2003
Espia.java tiene 1917 bytes.
```

Archivos de acceso aleatorio

Existe la posibilidad de leer archivos aleatoriamente. Existe una clase denominada `RandomAccessFile`, que contiene un archivo de acceso aleatorio. Esta clase implementa las interfaces `DataInput` y `DataOutput`, que definen los métodos más comunes para entrada y salida. La clase `RandomAccessFile` se instancia de la siguiente manera:

```
RandomAccessFile(String nombrearchivo, String acceso) throws
FileNotFoundException
```

Donde el nombre de archivo es el fichero al cual quiero acceder y acceso son los permisos al fichero, donde “r” es lectura y “w” escritura. El método `seek(long nuevaposicion)` se utiliza para que el curso se sitúe en esa posición, siempre en bytes.

Entrada/Salida con NI02

TABLA DE CONTENIDOS

Introducción.....	1
Path.....	2
Rutas Relativas.....	2
Rutas Absolutas.....	2
Clase Path	2
Creación de un Path	2
Convertir un Path.....	3
Enlazar dos Path	3
Comparar dos Path	4
Borrar un fichero o Directorio.....	4
Metada y Propiedades de Ficheros.....	5
Atributos básicos de un fichero.....	5
Modificar propiedades fecha de creación del fichero.....	6
Permisos de los ficheros a través de Posix	6
Leer, escribir y crear ficheros	7
Escribir en ficheros	7
Leer ficheros	7
Creación de ficheros.....	8
Ficheros Acceso Aleatorio	9

Introducción.

Nio2, permite un enfoque basado en buffer y canales. Es un nuevo componente aparecido en Java 7. Es un formato asíncrono.

Los buffer son objetos que contienen datos y los canales son los dispositivos propiamente dichos como un archivo o socket. Para usar este sistema, normalmente se obtiene un buffer de datos para guardar los datos y generar la entrada y salida y un canal para enviarlos.

El paquete `java.nio.file` y el paquete `java.nio.file.attribute` proporcionan soporte de entrada y salida para el acceso por defecto del sistema. Esto, no sustituye al paquete `java.io` sino que da funcionalidad añadida. Es bastante intuitivo y de fácil uso.

Java NIO proporciona funcionalidad para:

- El acceso a estructuras de directorio a través de la clase `Path`.
- Métodos para manipular Ficheros como, copiar, borrar o mover , además de devolver propiedades de los ficheros a través de la clase `Files`.
- Podemos obtener información sobre el sistema de ficheros a través de la clase `FileSystem`.

Un sistema de ficheros organizado en uno o más discos duros se organizan en estructuras jerárquicas. El nivel más alto son los nodos que contienen subniveles de directorios o ficheros.



Path

Un fichero es identificado por un path en la estructura de ficheros. Las distintas rutas para acceder a estos ficheros pueden ser relativas o absolutas.

Rutas Relativas

A partir del directorio actual se realizan las búsquedas de directorios. No se incluye el directorio raíz. Por ejemplo, si estamos en la ruta C:/Raiz/bin y queremos volver al directorio raíz podemos utilizar caracteres como "..", los caracteres tradicionales de búsquedas de directorios.

Rutas Absolutas

En la búsqueda de directorios, a partir de rutas absolutas, se incluye toda la ruta donde se encuentra el fichero correspondiente. C:/raiz/...

Clase Path

Introducida en el paquete JavaSE7, se encuentra en el paquete java.nio.file. Esta clase es la representación de los path o rutas en el sistema de ficheros. El objeto Path contiene métodos para localizar, examinar y manipular ficheros.

El Path en Solaris, se especifica a través del carácter "/raiz/bin", mientras que en Windows, la búsqueda se hace con "raiz\bin".

Creación de un Path

Un Path, contiene información para especificar la localización de un fichero o directorio. La estructura de directorios debe de ser incluida para poder hacer la creación. La clase Path, proporciona los siguientes métodos:

Métodos	Definición
ToString()	Retorna la ruta del Path
getFileName()	Retorna el nombre de la última secuencia de elementos
getName(0)	Retorna la ruta de elementos correspondiente al índice especificada
getNameCount	Retorna el número de elementos del Path
subPath(0,2)	Retorna la secuencia del path, no incluye el elemento root
getParent	Retorna el directorio padre
getRoot	Retorna el elemento root

Un ejemplo sería el siguiente suponiendo como Path c:/temp/Ejemplo en Windows.

```
public class Ejer_Path {  
  
    public static void main(String[] args) {  
        //Accecedemos a un Path  
        Path p1 = Paths.get("c:\\temp\\ejemplo");  
        System.out.println(p1.toString());  
        System.out.println(p1.getFileName());  
        System.out.format("getName(0): %s\n", p1.getName(0));  
        System.out.format("getNameCount: %d\n", p1.getNameCount());  
        System.out.format("subpath(0,2): %s\n", p1.subpath(0,2));  
        System.out.format("getParent: %s\n", p1.getParent());  
        System.out.format("getRoot: %s\n", p1.getRoot());  
    }  
}
```

Convertir un Path

Se puede usar un método para convertir un string en un objeto que pueda ser abierto por el navegador. El método **toUri**, proporciona esta funcionalidad.

```
System.out.format("%s\n", p1.toUri());
```

La salida por pantalla sería la siguiente:

```
file:///c:/temp/ejemplo
```

El método **toRealPath** retorna el path real de un fichero existente. Este método ejecuta operaciones como:

- Si el path es relativo , devuelve el path absoluto
- Si el Path contiene elementos redundantes, retorna el path con esos elementos eliminados.

Este método lanza una excepción, un ejemplo sería como los siguiente:

```
try {  
    Path p2 = p1.toRealPath();  
    System.out.println("RealPath: "+p2);  
} catch (NoSuchFileException x) {  
    System.err.format("%s: no encontrado" + " fichero o directorio", p1);  
    // Logic for case when file doesn't exist.  
} catch (IOException x) {  
    System.err.format("%s\n", x);  
    // Logic for other sort of file error.  
}
```

La visualización del ejemplo sería:

```
RealPath: C:\TEMP\ejemplo
```

Enlazar dos Path

Se pueden combinar dos Path, usando el método **resolve()**. Se pueden pasar rutas parciales, no incluidas en la ruta actual y añadirlas al Path. Estando en la ruta c:\temp\ejemplo incluimos la ruta añadida.

```
System.out.println( p1.resolve("añadida"));
```

Comparar dos Path

La clase Path soporta el método equals, con lo que es posible comparar dos Path. No obstante esta clase proporciona los métodos startsWith y endsWith para comprobar el principio y fin de una cadena.

Un ejemplo de este método sería el siguiente:

```
System.out.println("Valor del Path p2: "+p2);
if(p2.startsWith(Paths.get("temp"))||p2.endsWith(Paths.get("añadida"))){

    System.out.println("encontraste la ruta");
}
```

El resultado sería:

```
Valor del Path p2: c:\temp\ejemplo\añadida
encontraste la ruta
```

Borrar un fichero o Directorio

Se puede borrar un fichero, directorio o link. La clase File del método delete(path) permite borrar un fichero o directorio de la cadena especificada.

Un ejemplo sería el siguiente:

```
try {

    Files.delete(p2);
    System.out.println("ruta borrada");
} catch (NoSuchFileException x) {
```

Otra opción, es el método deleteIfExists(Path) que borra un fichero, pero si el fichero no existe lanza una excepción.

Metada y Propiedades de Ficheros

Podemos saber los permisos, el directorio, la fecha de creación y el propietario a través de distintos métodos.

Métodos	Definición
Size(path)	<i>Retorna el tamaño en bytes</i>
isDirectory(path,linkOption)	<i>Retorna true si el fichero especificado en el Path , se encuentra</i>
isHidden(Path)	<i>Retorna true si el fichero está oculto para el sistema</i>
getLastModifiedTime(Path, LinkOption...) setLastModifiedTime(Path,	<i>Retorna o modifica la fecha de la última modificación del fichero.</i>
subPath(0,2)	<i>Retorna la secuencia del path , no incluye el elemento root</i>
getOwner(Path,LinkOption...) setOwner(Path, UserPrincipal)	<i>Retorna o modifica el propietario</i>
getPosixFilePermissions(Path, LinkOption...) setPosixFilePermissions(Path, Set<PosixFilePermission>)	<i>Retorna o modifica los permisos de un fichero en caracteres Posix.</i>

Atributos básicos de un fichero

Para poder ver los atributos de un fichero , es posible usar la clase Files.readAttributes, que devuelve una colección de tipo BasicFileAttribute que tiene las propiedades básicas de un fichero. Un ejemplo sería el siguiente:

```

System.out.println("Propiedades de un Fichero");
Path p3 = Paths.get("c:\\temp\\ejemplo\\fichero.txt");
BasicFileAttributes attr = Files.readAttributes(p3,
BasicFileAttributes.class);
System.out.println("Fecha de Creación: " + attr.creationTime());
System.out.println("Última Fecha de acceso: " +
attr.lastAccessTime());
System.out.println("Última fecha de modificación: " +
attr.lastModifiedTime());

System.out.println("isDirectorio: " + attr.isDirectory());
System.out.println("isFichero Regular: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());

```

Modificar propiedades fecha de creación del fichero

Es posible modificar las propiedades de tiempo de un fichero de esta manera:

```
long currentTime = System.currentTimeMillis();
FileTime ft = FileTime.fromMillis(currentTime);
Files.setLastModifiedTime(p3, ft);
System.out.println("Última fecha de modificación: " +
attr.lastModifiedTime());
```

Permisos de los ficheros a través de Posix

Posix es un acrónimo de Portable Operating System Interface para Unix y sistemas operativos estándar, además de IEEE.

A través de la clase `PosixFileAttributes.class` y `PosixFilePermissions` podemos obtener los permisos de un fichero. Los métodos de la clase `PosixFilePermissions` son:

Métodos	Definición
<code>ToString()</code>	<i>Convierte los permisos a un String</i>
<code>fromString()</code>	<i>Acepta los String como permisos, devolviendo los permisos en su formato correspondiente.</i>

Podemos ver los permisos de un fichero, de la siguiente forma:

```
System.out.println("*****Permisos fichero*****");
PosixFileAttributes attr2 =
    Files.readAttributes(p3, PosixFileAttributes.class);
System.out.format("%s %s %s\n",
    attr2.owner().getName(),
    attr2.group().getName(),
    PosixFilePermissions.toString(attr2.permissions()));

}
```

MODIFICAR UN FICHERO O SU GROUP OWNER

Se puede dar permisos a un fichero a través de un usuario o de un grupo de usuarios. De tal manera, que con el método `LookupPrincipalByName` se puede dar permiso a un usuario concreto, o `LookupPrincipalGroupName` se da permisos a través de un grupo.

La forma sería de la siguiente manera:

```
GroupPrincipal grupo = p2.getFileSystem().getUserPrincipalLookupService()
    .lookupPrincipalByName("cursoJava");
Files.getFileAttributeView(p2, PosixFileAttributeView.class)
    .setGroup(grupo);
```

Leer, escribir y crear ficheros

Nio2 permite además la manipulación de ficheros a través de la clase `File` y el método `newBufferedReader` que crea un objeto de tipo `Buffered` para generar entrada-salida.

Los pasos para hacer esto serían:

- Crear un Buffer
- Crear Canales
- Utilizar la clase `File`...

La clase `BufferedReader` tiene parámetros denominados `OpenOptions` donde indicamos el formato en el que se abre un fichero.

Tiene la siguientes posibilidades:

Métodos	Definición
WRITE	Abre el fichero en modo escritura
APPEND	Permite añadir nuevas filas al fichero
TRUNCATE_EXISTING	Trunca el fichero a 0 bytes
CREATE_NEW	Crea un nuevo fichero lanzando una excepción si ya existe.
CREATE	Abre el fichero si existe y si no existe lo crea
DELETE_ON_CLOSE	Borrar el fichero cuando el Stream es cerrado

Escribir en ficheros

Existe la posibilidad de escribir en fichero a través del método `write` de la clase `File`. A `write(path,buffer)` se le pasa el `Path` y un array de bytes de tal manera que el resultado sería el siguiente:

```
byte[] buf = "esto es una prueba 2".getBytes();
Files.write(p3, buf);
```

Leer ficheros

Existe la posibilidad de escribir en fichero a través del método `write` de la clase `File`. A `write(path,buffer)` se le pasa el `Path` y un array de bytes de tal manera que el resultado sería el siguiente:

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(p3, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s\n", x);
}
```

Creación de ficheros

Se puede crear un fichero vacío con un conjunto de atributos iniciales usando `createFile(path,fileAttribut)e`. Si el fichero existe se lanza una excepción. A continuación se muestra un ejemplo:

```
Path p4 = Paths.get("c:\\temp\\ejemplo\\fichero4.txt");
try {

    Files.createFile(p4);
} catch (FileAlreadyExistsException x) {
    System.err.format("Error el fichero existe");
} catch (IOException x) {

    System.err.format("Error de permisos sobre el fichero");
}
```


Ficheros Acceso Aleatorio

De manera estándar y a través del paquete Java.io, también existe la posibilidad de acceder a los ficheros de manera no secuencial. La clase `ByteBuffer` proporciona métodos para mover el puntero y distintas partes del fichero y la interface `SeekableByteChannel` que implementa la clase `FileChannel` proporciona métodos para leer y escribir de manera aleatoria. Algunos de los métodos disponibles son los siguientes:

Métodos	Definición
<code>position</code>	<i>Retorna la posición actual del cursor</i>
<code>Position(long)</code>	<i>Modifica la posición</i>
<code>Read(byteBuffer)</code>	<i>Lee los bytes del buffer</i>
<code>Write(byteBuffer)</code>	<i>Escribe en el buffer</i>
<code>Truncate(long)</code>	<i>Trunca el fichero correspondiente.</i>

```
public class Prueba
{
    public static void main(String[] args) throws IOException
    {
        RandomAccessFile File = new RandomAccessFile
        ("prueba.txt", "r");
        FileChannel inChannel = File.getChannel();
        ByteBuffer buff = ByteBuffer.allocate(1024);
        while(inChannel.read(buffer) > 0)
        {
            buff.flip();
            for (int i = 0; i < buff.limit(); i++)
            {
                System.out.print((char) buff.get());
            }
            buff.clear();
        }
        inChannel.close();
        File.close();
    }
}
```

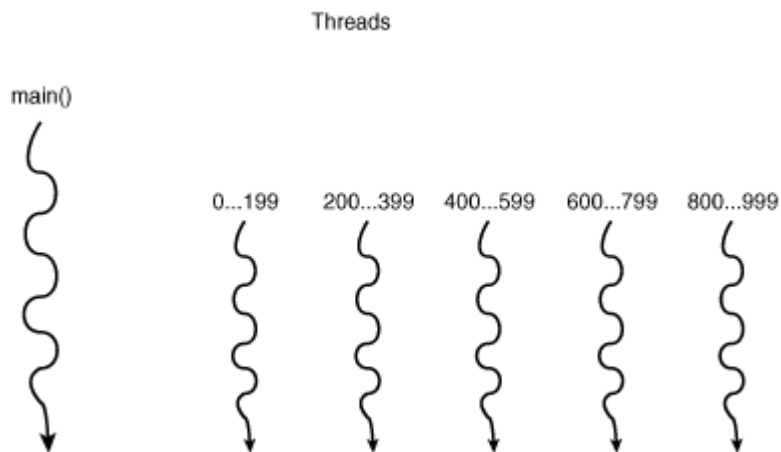
Programación Concurrente (Threads)

TABLA DE CONTENIDOS

Introducción a los Threads.....	1
Los thread.....	1
Estados de un Thread	1
Creación de un Threads.....	3
Extendiendo de la clase Thread	3
Implementación del interface Runnable.....	4
Control de threads.....	6
Parada de un Thread con sleep.....	6
Prioridades entre hilos (yield)	6
Enlazar threads (join).....	8
Sincronización de Threads	10
Utilización de synchronized	11
control de sincronización wait y notify.....	13
El método wait().....	13
El método notify/notifyall.....	14

Introducción a los Threads

Un Thread es un flujo secuencial de ejecución dentro de un programa. Dentro de la máquina virtual de Java, un thread representa una CPU virtual que ejecuta el código de un programa. Estos “hilos de ejecución” permiten que varios programas puedan ejecutarse de manera simultánea. Por ejemplo mientras un programa lee el otro escribe. Un subproceso puede tener varios estados. Puede estar en ejecución, suspendido, en espera, dormido, parado para posteriormente reanudarlo. Hay que tener en cuenta que todos los threads dependerán siempre de un hilo principal denominado “Main”. Si el hilo principal muere o se para, finaliza todos los Threads correspondientes.



Los thread

Un Thread se compone de tres partes:

- Una CPU virtual.
- El código que ejecuta en la CPU virtual.
- Los datos con los que trabaja ese código.

Varios Threads pueden:

- No compartir nada → Ejecutan código de objetos de clases distintas.
- Compartir códigos → Ejecutan código de objetos distintos pero pertenecientes a la misma clase.
- Compartir código y datos → Ejecutan código del mismo objeto.

Estados de un Thread

El comportamiento de un Thread depende del estado en el que se encuentre. Existen distintos tipos de estados:

- New: Es cuando se crea el Thread y antes de llamar al método Start.
- Runnable: Es en modo ejecución. Ha invocado al método run y esta ejecutando la lógica.
- Not Running: Es el estado por el cual los hilos están parados. Está listo para ser ejecutado de nuevo. Este estado puede ser porque:
 - Sleep: Dormido varios milisegundos

- I/O: Está bloqueado
- Dead: El Thread está muerto y por lo tanto, ya no es un objeto necesario.

Creación de un Thread

El sistema de procesamiento múltiple se basa en la clase Thread y en la interfaz Runnable, estas se encuentran en el paquete java.lang.Thread. Existen dos maneras de crear Thread:

- Extendiendo de la clase Thread: De ser así, no tendremos la posibilidad de heredar posteriormente nuestras clases de otra.
- Implementando la interfaz Runnable: Es la más habitual, pues permite realizar las mismas acciones , dejando además, que nuestra clase pueda extender posteriormente de otras clases.
- Redefinir los métodos necesarios:
 - Run: Método necesario para la lógica del Thread
 - Start: Método que se encarga de invocar al método Run. Arranca el Thread

Existen varios métodos que heredamos de esta clase Thread que permiten dar la funcionalidad a nuestro “hilo” de ejecución. Los métodos disponibles son:

Métodos	Definición
getName()	<i>Obtiene el nombre del Thread</i>
getPriority(int)	<i>Devuelve la prioridad de los hilos</i>
isAlive()	<i>Indica si un proceso sigue en ejecución</i>
Join()	<i>Espera a que termine un subproceso</i>
Run()	<i>Contiene la lógica de ejecución del Thread</i>
Sleep(long milisegundos)	<i>Deja que el hilo de ejecución se quede dormido</i>
Start	<i>Arranca el Thread ejecutando Run</i>
currentThread()	<i>Devuelve el hilo principal.</i>

Extendiendo de la clase Thread

A la hora de extender de la clase Thread, debemos de tener en cuenta que:

- Debemos de extender de la clase thread
- Debemos Redefinir el método Run

```
public class Ejercicio1_hilo extends Thread {
```



```
//se crea un Constructor con nombre para que cada Thread esté nominad
    public Ejercicio1_hilo(String nombre){
        super(nombre);
    }
//Se ejecuta la lógica de Thread.Un Bucle del 1 al 10.
    public void run(){
        for(int i=0;i<10;i++){
            System.out.println(getName() + " - " + i);
        }
    }
}
```

Ahora crearemos un programa principal que ejecute tantos Threads como necesitamos. En el programa principal debemos de prestar especial atención a la ejecución del Thread principal , pues éste finaliza después de la ejecución de todos los Threads hijos.

```
public class Principall {
    public static void main(String[] args){
        //creamos el hilo
        Ejercicio1_hilo hilo = new Ejercicio1_hilo("Hilo Hijo");
        //Arrancamos el hilo creado
        hilo.start();
        //Recogemos una referencia la hilo principal y que este también se ejecute 10 veces
        for(int i=0;i<3;i++){
            //Devuelve el nombre del Thread (siempre es main) y la posición
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                //El Thread principal se queda un dormido 1sg por cada iteración
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

Implementación del interface Runnable

El interface Runnable define el método, run(). La clase Thread implementa esta interface. En esta forma de creación de Threads, se tendrá que definir:

- Una clase que instancie la Clase Thread, a esta es necesaria pasarle la referencia This, con el nombre del Thread generalmente para diferenciarlos.
- Desarrollar el método run().
- Crear un objeto de tipo Thread como atributo de la clase.
- Llamar al método start del Thread creado.

Obsérvese el siguiente ejemplo:

```
public class Ejercicios2_hilo implements Runnable{
    Thread t;
    //Instanciar la clase Thread
```

```
public Ejercicios2_hilo(String nombre){
    t = new Thread(this,nombre);
}
@Override
//Definir el comportamiento
public void run() {
    for(int i=0;i<10;i++){
        System.out.println(t.getName() + " - " + i);
    }
}
//ejecutar el Thread
public void start(){
    t.start();
}
}
```

La ejecución desde un método main, quedaría así:

```
public class Principal2 {

    public static void main(String[] args) {
        Ejercicios2_hilo ej =new Ejercicios2_hilo("hilo1");
        ej.start();
        for(int i=0;i<10;i++){
            System.out.println(Thread.currentThread().getName() + " - " +
i);
        }

    }

}
```

Control de threads

Como se ha visto anteriormente, hemos podido crear Threads o flujos de ejecución. Pero estos también tienen una serie de estados que podemos definir.

- void start(): Arranca el Thread, ejecutando el método run del mismo
- void sleep(milisegundos): pone en modo suspensión un thread por un tiempo mínimo especificado.
- void join(): Enlaza un Thread con otro.
- void yield(): el proceso actual se queda suspendido y da prioridades a otros hilos.

Parada de un Thread con sleep

El método sleep deja que el hilo se quede suspendido los milisegundos especificados.

Un ejemplo de este método sería el siguiente:

```
import java.lang.*;

public class EjemploSleep implements Runnable {
    Thread t;

    public void run() {
        for (int i = 10; i < 13; i++) {

            System.out.println(Thread.currentThread().getName() + " " + i);
            try {
                // El hilo se queda dormido 1 seg
                Thread.sleep(1000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new EjemploSleep());
        // Esto invoca a la función run
        t.start();

        Thread t2 = new Thread(new EjemploSleep());
        //Invoca a la función run
        t2.start();
    }
}
```

Prioridades entre hilos (yield)

Es posible que ejecutemos un conjunto de hilos sobre los cuales queremos dar prioridad pues ocupa bastantes recursos. Un hilo o Thread siempre tiene una prioridad, normalmente va de 1 a 10 y por defecto tienen nivel 5. Existen algunos métodos específicos que afectan a este comportamiento:

Métodos	Definición
getPriority()	Retorna la prioridad del hilo

setPriority(int)	<i>Modifica la prioridad de un hilo</i>
Thread.MIN_PRIORITY	<i>Constante que da el valor de 1 al Thread</i>
Thread.NORM_PRIORITY	<i>Constante que da valor 5 al Thread.</i>
Thread.MAX_PRIORITY	<i>Constante que da valor 10 a nuestras aplicaciones</i>
Yield()	<i>Se queda suspendido para dar prioridad a otros hilos con la misma prioridad. El comportamiento no está garantizado, pues si vuelve a estado de ejecución no garantiza que vuelva a ser elegido para seguir con la ejecución</i>

Un ejemplo del método yield sería el siguiente:

```
import java.lang.*;

public class EjemploSleep implements Runnable {
    Thread t;

    public void run() {
        for (int i = 6; i < 13; i++) {
            t.setPriority(5);
            System.out.println(Thread.currentThread().getName() + " " + i);
            try {
                // El hilo se queda dormido 1 seg
                Thread.sleep(1000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new EjemploSleep());
        // Esto invoca a la función run
        t.start();
        //Se queda en modo parada esperando a que se ejecute los demás
        t.yield();

        Thread t2 = new Thread(new EjemploSleep());
        //Invoca a la función run
        t2.start();
        //se queda en modo parada esperando a que se ejecute los demás
        t2.yield();
        //Este será el hilo que tenga mayor prioridad
        Thread t3 = new Thread(new EjemploSleep());
        t3.start();
    }
}
```

La salida quedaría así:

```
hread-0  10
Thread-2  10
Thread-1  10
```

```
Thread-2  11
Thread-1  11
Thread-0  11
```

Hay que tener en cuenta que la finalización del programa empieza por el tercer Thread, por lo que hemos conseguido darle prioridad.

```
Thread-2  12// es el primero en dar prioridad
Thread-1  12
Thread-0  12
```

Enlazar threads (join)

El método `join()` permite que el hilo se quede a la espera hasta que finalice el otro hilo en ejecución. Si tenemos dos hilos, el hilo B que no puede comenzar a ejecutarse hasta que se complete el proceso del hilo A. Esto significa que B nunca podrá ejecutarse si A no completa su proceso. En código se utiliza así:

```
import java.lang.*;

public class EjemploJoin implements Runnable {
    Thread t;
    public EjemploJoin(String nombre){
        //Creamos un Thread
        t=new Thread(this,nombre);
        //Arrancamos el thread
        t.start();
        try {
            //Al generar Join hasta que no termine el que empezo no comienza el
            siguiente
            t.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public void run() {
        //el Thread genera una secuencia del 6 al 12
        for (int i = 6; i < 13; i++) {

            System.out.println(Thread.currentThread().getName() + "  " + i);
            try {

                Thread.sleep(1000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
    public static void main(String[] args) {
        EjemploJoin ej=new EjemploJoin("uno");

        EjemploJoin ej2=new EjemploJoin("dos");

    }
}
```

La salida quedaría de la siguiente manera:

```
uno  6
uno  7
uno  8
uno  9
```

```
uno 10  
uno 11  
uno 12  
dos 6  
dos 7  
dos 8  
dos 9  
dos 10  
dos 11
```

Sincronización de Threads

Al usar varios subprocesos, en algunos casos, es necesario que comparta recursos para coordinar estos procesos. A esta situación se le denomina sincronización. Un ejemplo habitual es si queremos escribir en un archivo, debemos de impedir que otro lo haga al mismo tiempo.

En Java, la clave de sincronización, es el concepto de monitor que va a controlar el acceso a un objeto. El monitor, se encarga de bloquear el recurso cuando un Thread lo recoge y desbloquearlo una vez utilizado.

Por ejemplo, supongamos que un determinado método de una clase muestra el mensaje "[mensaje"+ luego interrumpe la ejecución durante un segundo y luego cierra el mensaje con "]". Si existen tres Threads que intentan acceder a este método , veremos que se produce un resultado indeseable.

A continuación muestro el método que imprime el mensaje que va a ser accesible desde distintos Threads.

```
//Este es la clase que queremos compartir entre varios Threads con un mensaje
encerrado
//entre corchetes
public class Cllamada {
    //Genero un Constructor con el mensaje
    void llamada(String msg){
        System.out.println("[ "+msg);
        try{
            //Se queda el mensaje en espera de 1 segundo
            Thread.sleep(1000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        //Después de la llamada sleep cierra con un corchete
        System.out.println("]");
    }
}
```

Una vez creada las clase que queremos compartir entre distintos Threads , creamos la clase thread que utiliza esta clase.

```
//Genero el Thread
public class Cllamante implements Runnable
{
    String msg;
    Cllamada cll1;
    Thread t;
    //Le paso una referencia del objeto que quiero utilizar
    public Cllamante (Cllamada x, String s){
        cll1=x;
        msg=s;
        t=new Thread(this);
        t.start();
    }
    public void run(){
        cll1.llamada(msg);
    }
}
```

La ejecución del Thread quedaría de la siguiente manera en el Main:

```
public class Principal {  
  
    public static void main(String[] args) {  
        Cllamada c=new Cllamada();  
        Cllamante ob1=new Cllamante(c,"aaa");  
        Cllamante ob2=new Cllamante(c,"bbb");  
        Cllamante ob3=new Cllamante(c,"ccc");  
        try {  
  
            //hacemos que hasta que no finalice el primer Thread no comience el siguiente.  
            //Eso lo conseguimos con Join  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
  
    }  
  
}
```

Aun utilizando el método join, en la parada de un segundo del Thread se ejecuta los Thread siguientes. La salida al programa anterior sería la siguiente:

```
[bbb  
[aaa  
[ccc  
]  
]
```

Utilización de synchronized

Para resolver el problema anterior, se utiliza la cláusula Synchronized. Permite que un método, código o atributo tenga control de exclusión mutua.

Es posible que la cláusula se aplique a un atributo, clase o código.

Si lo aplicamos a un atributo quedaría así:

```
public void funcion2(){  
    Rectangle rect;  
    synchronized(rect){  
        rect.width+=2;  
    }  
    rect.height-=3;  
}
```

Es posible aplicarlo a un método:

```
synchronized public void funcion1(){  
    //...  
}
```

O bien a una parte de código:

```
public void mover(){  
    synchronized (this) {  
        indice++;  
    }  
}
```



```

        if (indice >= numeros.length) {
            indice = 0;
        }
    }
    //...
}

```

Aplicado al ejemplo anterior con la cláusula Synchronized, nos daremos cuenta que cada palabra se cierra con su corchete correspondiente, consiguiendo, que en efecto, haya exclusión mutua en el método.

```

public class Cllamada {
    //Genero un Constructor con el mensaje
    synchronized void llamada(String msg){
        System.out.println("[ "+msg);
        try{
            //Se queda el mensaje en espera de 1 segundo
            Thread.sleep(1000);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
        //Después de la llamada sleep cierra con un corchete
        System.out.println("]");
    }
}

```

También es posible sincronizar solamente la parte de código correspondiente que nos interesa. Esto sería de la siguiente manera:

```

//Este es la clase que queremos compartir entre varios Threads con un mensaje
//encerrado
//entre corchetes
public class Cllamada {
    //Genero un Constructor con el mensaje
    void llamada(String msg){
        synchronized(this){
            System.out.println("[ "+msg);
            try{
                //Se queda el mensaje en espera de 1 segundo
                Thread.sleep(1000);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            //Después de la llamada sleep cierra con un corchete
            System.out.println("]");
        }
    }
}

```

Si no podemos modificar el código de clase llamante podemos sincronizar la llamada a la clase correspondiente mediante el siguiente código:

```

//Genero el Thread
public class Cllamante implements Runnable
{
    String msg;
    Cllamada cll1;
    Thread t;
    //Le paso una referencia del objeto que quiero
    public Cllamante (Cllamada x, String s){

```

```
        cll1=x;
        msg=s;
        t=new Thread(this);
        t.start();
    }
    public void run(){
        synchronized(cll1){
            cll1.llamada(msg);
        }
    }
}
```

Control de sincronización wait y notify

En ocasiones resulta necesario que varios hilos realicen operaciones de sincronización, de tal forma, que se puedan notificar ciertas circunstancias a cerca de su situación respecto a los recursos críticos que dan lugar a las secciones críticas. Estas operaciones se realizan a través de un intercambio de señales de parada y reanudación.

Con este propósito, todos los objetos heredados de Object, implementa los métodos wait () y notify() , de manera que, si un Thread ejecuta una llamada al método wait() sobre un objeto X, dicho Thread quedará bloqueado hasta que otro Thread ejecute una llamada a notify() sobre el mismo objeto X.

Existen determinados problemas de programación donde el uso del este tipo de comunicación se ha estrictamente necesario, es el caso del ejemplo del productor-consumidor.

El método wait()

Para detener la ejecución y esperar a otro Thread nos envíe la seña, se utiliza:

- Public void wait();
- Public void wait(long timeout)
- Un ejemplo de Wait básico:

```
public class Wait extends Thread{
    static Wait w = null;
    //3. Se ejecuta el método run y espera
    synchronized public void run() {
        System.out.println("Antes de Esperar wait");
        try {
            w.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("No se ejecuta nunca");
    }

    public static void main(String[] args)
    {
        //1.instanciamos la clase
        w = new Wait();
        //2. Se arranca
        w.start();
    }
}
```

La salida sería la siguiente:

```
Antes de Esperar wait
```

El método notify/notifyall

El método notify/notifyall informa a uno u todos los Threads que están esperando mediante el método wait, que pueden continuar.

A continuación, muestro ejemplo de cómo utilizar el método notify

```
public class Wait1 extends Thread{
    static Wait1 w = null;
    synchronized public void run() {
        System.out.println("Antes de esperar"); // 4 Imprime antes de esperar
        try {
            w.wait(); // 5 espera
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Después de esperar y notificar que siga !!!"); // 7
    }

    public static void main(String[] args) throws InterruptedException {
        w = new Wait1(); // 1 Creamos el thread
        w.start(); // 2 Arrancamos el Thread
        Thread.sleep(10); // 3 Dejamos que duerma
        synchronized(w) {
            w.notify(); // 6 Notificamos que espere
        }
    }
}
```

Programación en Red TCP/IP

TABLA DE CONTENIDOS

Introducción	1
Dirección y puerto	1
El protocolo TCP	2
Programación del protocolo TCP	3
modelo de comunicación	3
Creación de un servidor TCP	3
creación de un cliente TCP	5
Servidor multithread basado en TCP	7

Introducción

URLs y URLConnections proporcionan un mecanismo de alto nivel para acceder a los recursos en Internet. A veces los programas requieren una comunicación de red de nivel inferior, por ejemplo, cuando se quiere escribir una aplicación cliente - servidor.

En las aplicaciones cliente-servidor, el servidor proporciona algún servicio, como el procesamiento de consultas de bases de datos o el envío de los precios actuales de las acciones.

La comunicación que se produce entre el cliente y el servidor debe ser válida. Es decir, los datos deben llegar al cliente en el mismo orden en el que se enviaron.

TCP (Protocolo de Control de Transmisión) proporciona un canal válido para comunicaciones entre aplicaciones. Para comunicarse a través de TCP, un programa cliente y un programa de servidor deben establecer una conexión entre sí. Cada programa enlaza un conector a su extremo de la conexión.

Existen dos modalidades básicas de comunicación:

- Orientada a la conexión: Primero se establece un preámbulo para la comunicación entre las dos conexiones y después comienza la comunicación propiamente dicha. (Por ejemplo, la comunicación mediante el teléfono)
- No orientada a conexión: No existe negociación de la conexión. El envío de información comienza desde un principio (ej: el servicio de correos). El protocolo UDP (User Datagram Protocol) hace uso de esta técnica. Este protocolo no otorga garantías en la entrega del mensaje.

El número de puerto en sistemas es un número de 16 bits. Lo que quiere decir que se puede tomar valores desde 0 hasta 65535. Dentro de este rango existen una serie de valores, lo que están por debajo de 1024 que reciben el nombre de “puertos conocidos” y que se reservan para procesos servidores. El resto de números de puerto son utilizados para las aplicaciones cliente.

Por otra parte, las aplicaciones cliente necesitan conocer la dirección del puerto de la aplicación servidor con la que se quieren conectar.

Un socket es un punto de enlace de comunicación entre dos programas que se ejecutan en la red. Un socket es un punto de enlace de una comunicación entre dos procesos. Cuando dos procesos se comunican a través de una red, Java utiliza su modelo de streams. Un socket tiene asociados dos streams:

- Entrada
- Salida

Dirección y puerto

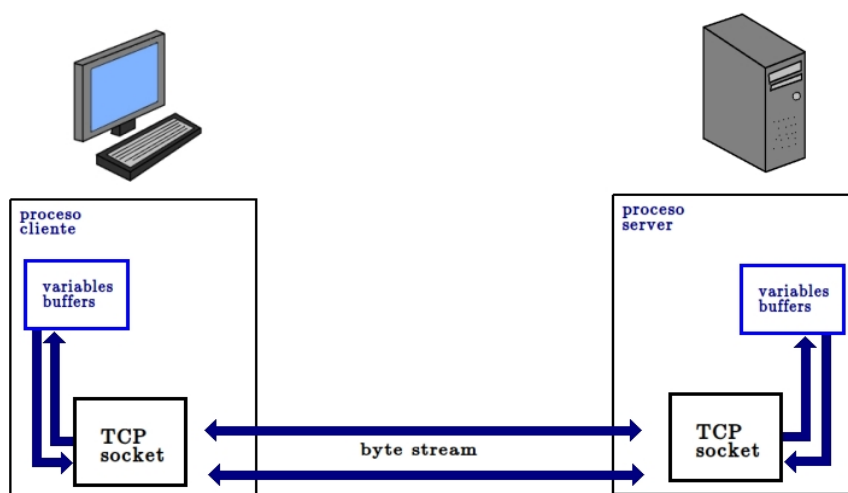
Para poder establecer la comunicación a través de una red, se tiene que conocer la dirección o nombre de la máquina remota así como el puerto en el que está escuchando el proceso servidor.

El puerto, es un numero entre 0 y 65535, aunque normalmente, los números por debajo del 1024 no se utilizan porque están reservados para servicios determinados (FTP, http, telnet, SMTP, etc...).

El protocolo TCP

TCP (Transport Control Protocol) es un protocolo orientado a conexión, es decir, se tiene que establecer una conexión previa entre los procesos para que se puedan mandar datos.

Para poder establecer tal comunicación deberá existir un proceso servidor, que atienda peticiones de conexión en una dirección y puerto conocidos por el socket cliente, que será quien solicite la conexión. Una vez la conexión se haya establecido, se pueden obtener los streams asociados para la recepción y envío de datos.



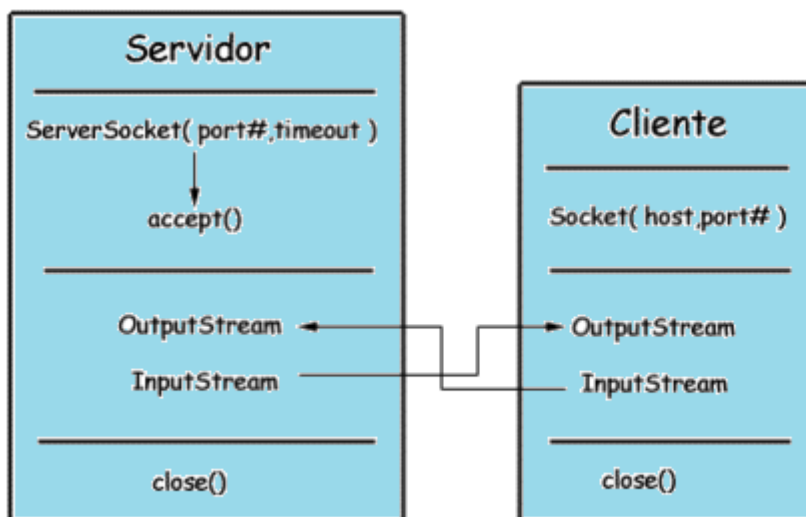
Programación del protocolo TCP

Las clase Socket se utilizan para representar la conexión entre un programa cliente y un programa servidor. El paquete java.net proporciona dos clases - Socket y ServerSocket - que implementan la parte cliente de la conexión y del lado del servidor de la conexión, respectivamente.

modelo de comunicación

El modelo de sockets más simple es:

- El servidor establece un puerto y espera durante cierto tiempo a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión socket con el método accept().
- El cliente establece la conexión con el servidor a través de puerto especificado.
- Cliente y servidor se comunican a través de InputStream y OutputStream.



Creación de un servidor TCP

El programa servidor se instala en un puerto determinado, a la espera de operaciones, a las que tratará un segundo sockets.

Para poder realizar comunicaciones por red, en primer lugar se deberá importar el paquete java.net, que contiene las clases necesarias y el paquete java.io para los streams. Los pasos necesarios serían:

- Instanciar el servidor con el número de puerto.

```
serverSocket = new ServerSocket(5000);
```

- Crear un objeto Sockets desde la clase ServerSocket para que esté atento a las conexiones que se puedan realizar y poder aceptar esas conexiones:

```
while(true){  
    socket = serverSocket.accept();
```

A través del socket se obtiene el stream para poder mandar los datos al cliente, aunque puede también recibir.

A través de la clase `InputStream` obtenida del método `getInputStream` de la clase `Socket` permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: `read()`, `readChar()`, `readInt()`, `readDouble()` y `readLine()` y con `OutputStream` obtenido de los métodos `getOutputStream` de la clase `Socket` nos permite a través los métodos `write`, `writeUTF` para poder enviar información al cliente.

```
out = socket.getOutputStream();
out.write(cadena);
```

Una vez realizada la comunicación, se finaliza el socket, terminando así la comunicación con ese cliente. El bucle vuelve a empezar y el servidor espera a nuevas peticiones en la llamada al método `accept()`. Obsérvese que este servidor solo es capaz de atender a un cliente de forma simultánea.

```
socket.close();
    }
    }catch(IOException e){
        System.out.println(e);
    }
}
```

De esta manera el servidor quedaría de la siguiente manera

```
package cap12;

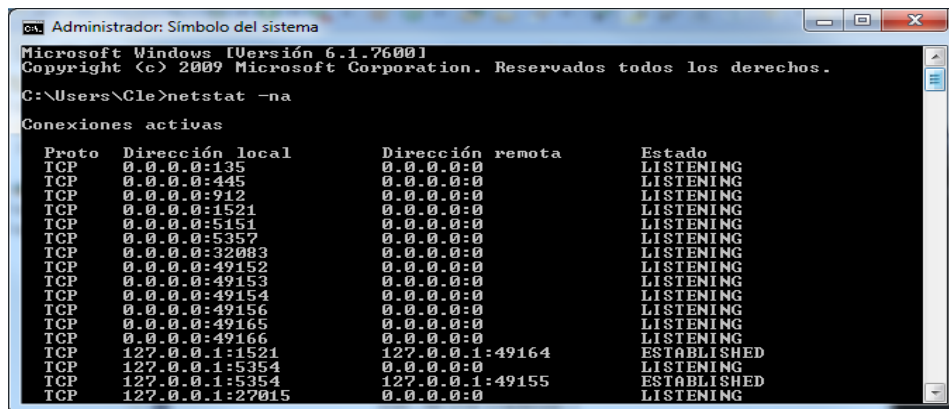
import java.net.*;
import java.io.*;

class Servidor {
    public static void main(String[] args){
        ServerSocket serverSocket;
        Socket socket;
        byte[] cadena = "Soy el servidor red...".getBytes();
        OutputStream out;

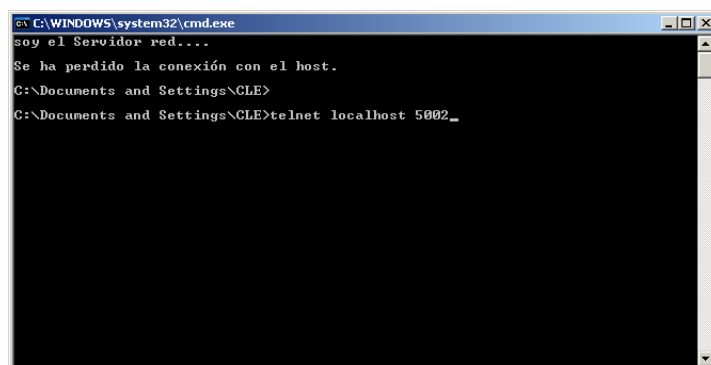
        try{
            serverSocket = new ServerSocket(5002);
            System.out.println("Servidor listo para atender peticiones");
            while(true){
                socket = serverSocket.accept();
                out = socket.getOutputStream();
                out.write(cadena);
                socket.close();
            }
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Si el servidor está a la escucha puede probarse mediante dos formas:

- Ejecutando la orden netstat en Unix como en Windows donde podemos comprobar el estado de los puertos, tal como se muestra en la siguiente pantalla.



- Lanzar un telnet hacia el puerto para reciba lo que el servidor envía a este tipo de cliente.



creación de un cliente TCP

Una vez creado el servidor, el cliente establece un puerto y espera a que el cliente establezca la conexión.

El cliente establece la conexión a través de una ip y el puerto designado.

Por lo tanto para definir el cliente debemos:

- Crear un Socket con la ip y el puerto del servidor:

```
Socket skCliente = new Socket( HOST , PUERTO );
```

- Luego se asocia el flujo de datos de dicho Socket (obtenido mediante `getInputStream()`), que es asociado a un flujo (flujo) `DataInputStream` de lectura secuencial. De dicho flujo capturamos una cadena `readUTF()`. Esto permite leer lo que envía el servidor.

```
InputStream aux = skCliente.getInputStream();
DataInputStream flujo = new DataInputStream( aux );
```

- El socket se cierra, una vez finalizadas las operaciones, mediante el método `close()`

```
skCliente.close();
```

El cliente quedaría así:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;
class Cliente{

    public static void main(String[] args) throws IOException{
        int c;
        Socket socket;
        InputStream in;

        try{
            socket = new Socket("localhost",5005);

            in = socket.getInputStream();
            while((c = in.read())!= -1){
                System.out.print((char)c);
            }
            socket.close();
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Servidor multithread basado en TCP

En el ejemplo anterior, el servidor acepta una conexión del cliente sobre el hilo de control actual y las sucesivas conexiones cliente son también controladas por el hilo actual. Esto es algo que los servidores profesionales no hacen, ya que si el hilo principal falla, fallarán también todas las conexiones de cliente. Atender las diferentes conexiones de cliente con hilos independientes permite el aislamiento de fallos de comunicación.

Para poder conseguir la independencia mencionada en el servidor del ejemplo anterior y para que el servicio de escucha del `ServerSocket` esté libre el mayor tiempo posible podemos utilizar la creación de `Threads`. A continuación se muestra un ejemplo de cómo conseguirlo.

- Inicialmente crearíamos un `Thread` que tuviera como atributos un `Socket` para que cada `Thread` se asociase a un `Socket` cliente concreto. El programa quedaría así:

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;

public class MultiServidor extends Thread {
    PrintStream out=null;
    Socket s=null;
    MultiServidor(Socket s){
        super();
        this.s=s;
    }
    public void run(){
        try{
            out=new PrintStream(s.getOutputStream());
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

- El servidor de `Socket` se crearía igual, lo único que lo diferencia es que la referencia al cliente obtenida con `accept()` se pasa a la clase anterior para que a ese cliente se le asocie un flujo de ejecución concreto.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class SocketServidor {
    public static void main(String args[]) throws IOException{
        ServerSocket s=new ServerSocket(2000);
        Socket s1 = null;
        boolean escuchando=true;
        while(escuchando){
            s1=s.accept();
            //Se asocia la referencia al cliente a un Thread.
            new MultiServidor(s1).start();
        }
        s1.close();
    }
}
```