Curso de verano de Acción Estudiantil.

Programación en Java.

Profesores: **Sergio Gálvez Rojas**

Índice

Constructores con parámetros.
Sobrecarga de funciones.
Sobrecarga del constructor.
La palabra clave *this*.
Llamada a un constructor desde otro constructor.
Uso de *this* en un constructor.
Inicialización de datos estáticos.
Sobrecarga del constructor.
Inicialización de *arrays*.

Capítulo 4: Reutilización de código.

Package.
Los ficheros .jar. Paquetes y colisiones.
Visibilidad.
Composición.
Herencia.
Visibilidad.
Inicialización en herencia.
Composición y herencia.
Upcasting.
La palabra clave *final*.
Polimorfismo.
La palabra clave *abstract*.
Interfaces.
Clases internas.
Polimorfismo y constructores.
Downcasting.
Downcasting y RTTI.

Capítulo 5: Colecciones.

Vectores.
Java no tiene clases parametrizadas.
Enumerators.
Hashtables.
Enumerators.
Colecciones y Java 1.2.
Test de List.

Capítulo 6: Tratamiento de excepciones.

Elevar y atrapar una excepción.
Captura y relanzamiento. Palabra reservada *finally*.
Ejemplo.
Detalles sobre excepciones.

Capítulo 7: E/S con Java.

Estratificación.
Ejemplo: Fichero de entrada con buffer.
Ejemplo: Entrada desde una cadena.
Ejemplo: Cadena de entrada con buffer.
Ejemplo: La entrada estándar.
Salida de datos.
Ejemplo: Escritura en un fichero de texto.

CheckboxGroup.
Choice.
List.
MenuComponent.
MenuIetm, setActionCommand(), MenuShortCut.
Canvas. Eventos de teclado, ratón y foco.
Ejemplo: foco, teclado, ratón. Canvas.
Dialog.
FileDialog.

Java 1.1.

Eventos en Java 1.1.
Eventos soportados.
Ejemplo de un botón.
Ejemplo de un botón con clases internas.
Aplicaciones y *applets* todo en uno.

Java Swing.

Bordes.
Botones.
Iconos.
Visualización de ejemplos

JDBC

Ejemplo de JDBC

Comunicación vía *sockets*

Servidor
Cliente
Servir a muchos clientes

Acceso por Internet a bases de datos: *applets* y *threads*

Internet, BB.DD. y *applets*
Creación de tablas en HTML
Internet y BB.DD. El *applet*
Internet y BB.DD. Servidor

Servlets

HTML y formularios
Ciclo de vida de un *servlet*
Clases para *servlets*
Ejemplo de *servlet*

Capítulo 12: RMI

RMI. El servidor
La comunicación
RMI. El servidor (cont.)
RMI. El cliente
Parámetros
Retrollamadas
Descarga de clases
Política de seguridad
Activación remota de objetos
Protocolo de activación
Ejemplo de activación

Capítulo 13: Ejecución de programas no Java. JNI

Capítulo 14: Internacionalización

Capítulo 0
Introducción

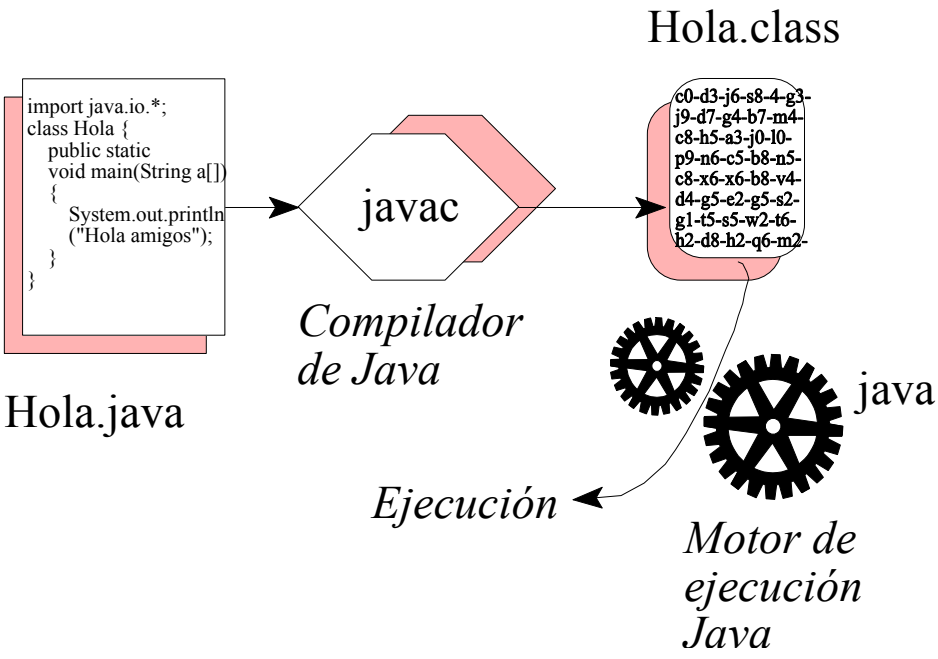
✯ Java es un lenguaje pseudointerpretado:
- Un fichero file.java se compila con:

javac file.java

y genera un fichero file.class, que se ejecuta con:

java file

* El fichero file.class contiene un código intermedio (formado por *bytecodes*) que debe ser interpretado por el ejecutor java.
* Para ejecutar un fichero .class, no es necesario poner la extensión.



Características generales

* Java es muy lento, ya que es pseudointerpretado. Es de 20 a 50 veces más lento que C++.

* ¿Para qué generar código intermedio y no código máquina puro?

Para conseguir que los programas sean portables y que se puedan ejecutar en cualquier máquina con independencia del µp que posea.

* Para evitar la lentitud hay dos aproximaciones:

- Compiladores que generan código nativo (código máquina) en lugar de pseudocódigo. Con esto se pierde portabilidad.
- Ejecutores JIT (*Just In Time*). Éstos, a medida que van traduciendo el pseudocódigo a código máquina y lo van ejecutando, dejan el código máquina traducido en un *buffer* para no tener que volver a traducirlo, con lo que una nueva ejecución de ese trozo de código será más rápida al no requerir una nueva traducción.

* Java es seguro. Impide accesos indebidos, y controla más condiciones de error que otros lenguajes, como C++.

Características generales

* Java permite tener las distintas partes que componen un programa distribuídas por la red.

* Java permite trabajar con Bases de Datos vía JDBC (*Java DataBase Connectivity*).

* Java es un lenguaje multitarea, tanto si la máquina es multiprocesador como si no.

* En Java no es tan importante el lenguaje como las librerías que ya incorpora.

* Por desgracia, Java evoluciona a un ritmo vertiginoso. Actualmente hay cuatro versiones fundamentales del JDK (*Java Development Kit*) que son: 1.0.2, 1.1.8 y 1.2 y 1.3. Las dos últimas se conocen por JDK 2.

* No podemos remitirnos exclusivamente a la última versión porque la mayoría de las características de las últimas versiones no son sustituciones a las anteriores, sino extensiones.

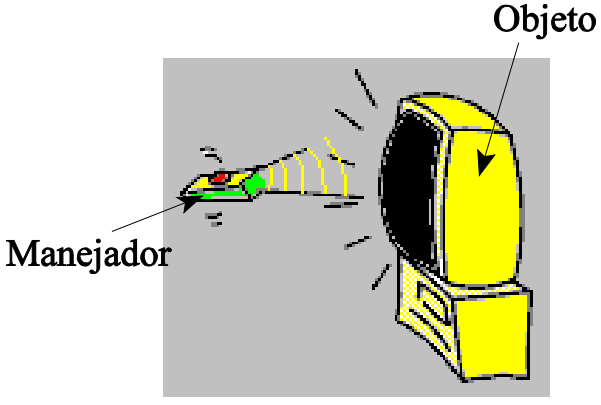
Características generales

* En Java todo son objetos. Incluso el programa principal se enmarca dentro de un objeto.

* Existe una excepción: los tipos primitivos, tales como **int**, **char**, etc., que no se consideran objetos y se tratan de forma especial.

* Los ficheros fuente tienen la extensión .java, y si un fichero se llama Xxx.java, debe contener obligatoriamente un tipo de objetos (clase) llamado Xxx.

* Java no posee punteros. Aparece en escena el concepto de **manejador**. Un manejador viene a ser como un puntero a objetos en C++, con la diferencia de que en Java todos los objetos deben ser controlados obligatoriamente a través de un manejador.



Convenciones sobre cómo escribir programas

* No es aconsejable usar el carácter de subrayado en los identificadores de usuario.

* Cualquier identificador de usuario, cuando esté formado por varias palabras, se escribira con todas las palabras juntas, y con la inicial de cada palabra en mayúscula, excepto la primera palabra que comenzará por minúscula. Ej.:
identificadorDeEjemploParaEsteCursoDeVerano

* Los nombres de las clases deben comenzar por mayúscula.
ClaseDeEjemplo

* Los nombres de los subdirectorios que conforman un paquete van completamente en minúsculas. Ej.:
mipaquete.otrodirectorio.*

* Los nombres de las constantes (**final static**) se colocan con todas las letras en mayúsculas. Si se emplean varias palabras, se separan por subrayados. Ej.:
NUMERO_DE_AVOGADRO = 6.023e+23;

Comentarios

* En Java los comentarios son como en C.
* Existe una herramienta llamada **javadoc** que extrae documentación automáticamente de nuestros programas, y la coloca en un fichero HTML.
* Para ello se utiliza un comentario que empieza por /** y acaba con */
* Estos comentarios deben colocarse siempre justo antes de una clase, un campo o un método, y pueden ser:

Para todos (el único admitido por los campos):

@see NombreDeClase

Permite referenciar a la documentación de otras clases.

Sólo para las clases:

@version información de la versión

@author información del autor

Sólo para los métodos:

@param nombreDeParametro descripción

@return descripcion

@exception NombreDeClaseCompletamenteCalificado descrip.

@deprecated

Los métodos marcados con **deprecated** hacen que el compilador emita una advertencia *(warning)* si se utilizan.

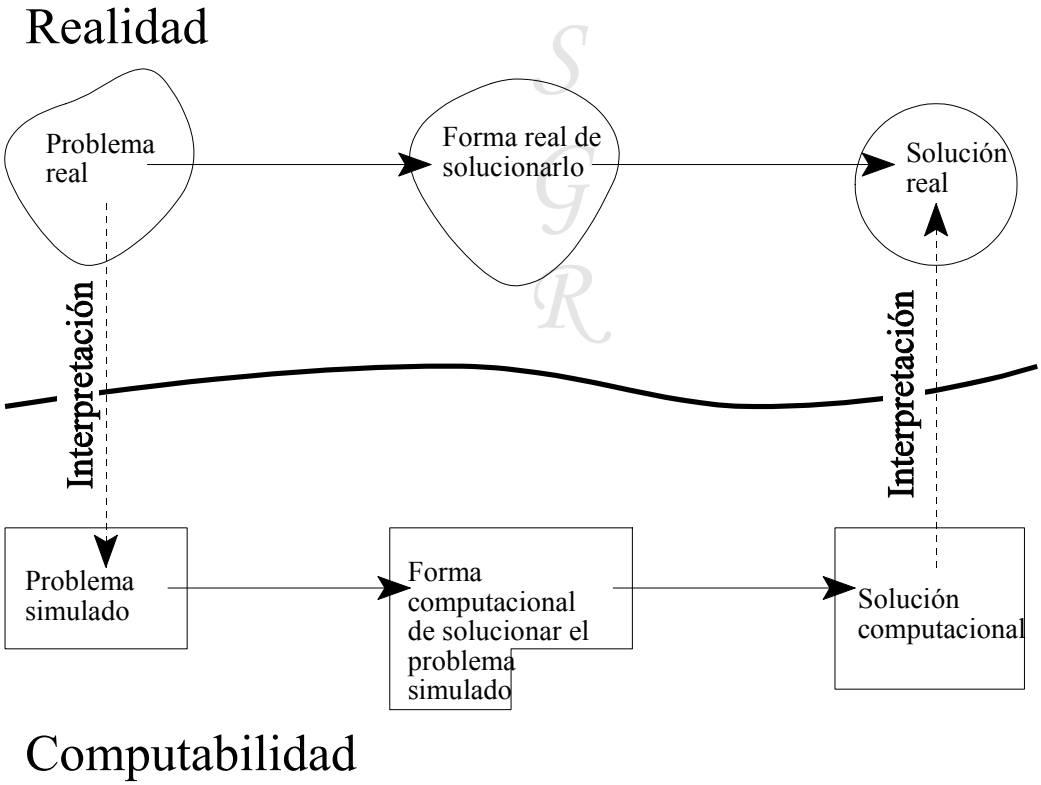
Capítulo 1
Conceptos O.O. de Java

✯ Espacio del problema:

Es el ámbito en el que se enmarca el problema que queremos solucionar: emitir unas nóminas por impresora, gestionar los pedidos de clientes, poner un satélite en órbita, etc.

✯ Espacio de las soluciones:

Es el ámbito en el que vamos a solucionar el problema, o sea, el ordenador. También llamado espacio computacional.



✯ De alguna manera, se pretende simular el problema real mediante medios computacionales para, mediante el ordenador, darle una solución que pueda ser interpretada en la vida real.

Idea de la O.O. en Java

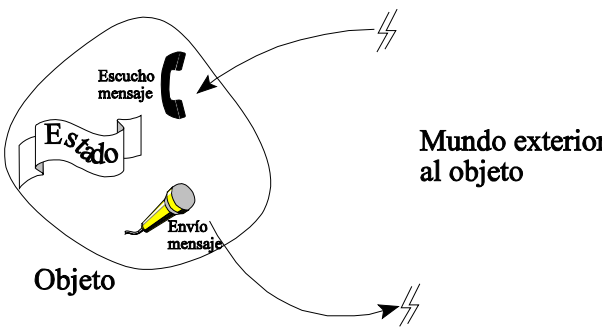
✯ La idea es:

El programa se debe adaptar a la «jerga» del problema real, creando nuevos tipos de objetos, de manera que al leer el código que implementa la solución, se esté leyendo algo que expresa el problema en sí.

✯ Esto se fundamenta en cinco puntos básicos:

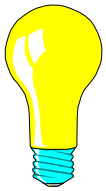
1.- Todo es un **objeto.**
2.- Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer a través de **mensajes.**
3.- Cada objeto tiene una memoria interna (**estado**), formada por otros objetos más pequeños; así hasta llegar a objetos básicos o **primitivos.**
4.- Todo objeto pertenece a un tipo. Según la terminología O.O. un tipo es lo mismo que una **clase.**
5.- Todos los objetos de un tipo concreto, son susceptibles de recibir los mismos mensajes.

✯ **Conclusión:** Un objeto tiene un estado que puede modificar como consecuencia de su respuesta a un mensaje.



Manejadores y objetos

✯ En O.O. una clase es lo mismo que un tipo.
✯ Las peticiones que se pueden hacer a un tipo quedan determinadas por su interfaz.



Clase:	Bombilla	
Interfaz:	encender()	Enciende la bombilla
	apagar()	Apaga la bombilla
	ilumina()	Dice si la bombilla esta encendida o no
	potencia()	Dice la potencia de la bombilla

```
Bombilla b = new Bombilla();
b.encender();
```

✯ Es necesario crear un **manejador** de Bombilla, simplemente declarando b. Sin embargo, todavía no existe ningún objeto bombilla. Es necesario crear un objeto Bombilla, con **new,** y asociar el objeto con el manejador.

✯ En Java no se manejan los objetos directamente, sino a través de manejadores. Todos los objetos se crean obligatoriamente con **new.**

✯ Los mensajes se le envían a los manejadores, y éstos los redirigen al objeto concreto al que apuntan.

✯ Asociado a cada mensaje de la interfaz, hay un método, que es un trozo de código que se ejecuta cuando el objeto recibe el mensaje correspondiente.

Productores y consumidores

✯ En programación O.O., el sistema se parece más a la creación de hardware. Hay productores de clases, y consumidores de clases, al igual que hay productores de chips y consumidores de chips para crear placas más grandes.

✯ ¿Por qué ocultar la implementación?

1) Así el cliente no tendrá acceso a ella, con lo que el productor la podrá cambiar a su antojo, sin preocuparse de dañar el funcionamiento del código de los clientes.

Esto es una ventaja también para el cliente, que sabe lo que puede utilizar sin ningún posible perjuicio posterior.

2) Es posible crear una clase de objetos de una forma fácil e ineficiente (para agilizar el desarrollo), y una vez concluido todo, modificar dicha clase (manteniendo su interfaz), para hacerla más eficiente.

✯ También se pueden ocultar algunos de los mensajes que puede recibir el objeto, así como algunos de los componentes de su estado.

Visibilidad

✯ Palabras reservadas para ocultar cosas internas a un objeto:

Se anteponen a cada uno de los elementos internos de que consta un objeto.

Modificador	Se aplica a	Ella misma	Package	Hijas	Otras clases
public	clases, interfaces, métodos y variables	✔	✔	✔	✔
protected	métodos y variables	✔	✔	✔	
	clases, interfaces, métodos y variables	✔	✔		
private	métodos y variables	✔			
private protected*	métodos y variables	✔		✔	

* Sólo en la versión 1.0 de Java.

Reutilización de código. Composición

✯ La forma más directa de usar una clase de objetos es creando objetos concretos.

✯ Relación Tiene-un: También se puede decidir que un elemento concreto compone a otro tipo de objetos más general. P.ej., la clase de objetos Coche posee cuatro objetos de la clase Rueda.

✯ Así, un objeto puede contener a muchos otros, y así sucesivamente. Cuando un objeto recibe un mensaje, dentro del método asociado a éste puede:

- ✗ Responder directamente.
- ✗ Reenviar el mensaje a otros objetos externos.
- ✗ Reenviar el mensaje a objetos que él mismo contiene.

Reutilización de código. Herencia

✯ Relación es-un: En O.O. Se permite coger una clase, crear un copia idéntica de la misma (clon), modificar la estructura de la copia, y crear así una nueva clase.

Esto es lo que se denomina **herencia,** aunque Java hace que si la clase original (llamada **clase base** o padre), se modifica posteriormente, la clase copia (también llamada clase heredera, derivada o **hija),** también reflejará esos cambios.

Java implementa la herencia con la palabra reservada **extends.**

✯ Cuando se hereda de una clase, se crea una nueva que contiene todos los elementos de la padre, no sólo los componentes del estado, sino, más importante aún, una copia de su interfaz.

✯ Para diferenciar la clase hija de la padre se puede:

- ✗Incluir nuevas funciones a la clase hija.
- ✗Modificar el funcionamiento de las funciones ya existentes en la clase padre. Es lo que se llama **reescritura.**

✯ La reescritura viene a decir algo así como:
«Estoy empleando la misma interfaz que mi padre, pero me quiero comportar de forma distinta».

Herencia: es-un vs es-como-un

✯ Controversia respecto a la herencia:
¿Debe limitarse una clase hija a reescribir los métodos de su padre? En otras palabras ¿deben tener las clases hijas exactamente la misma interfaz que la padre?

Si se obliga a esto, quiere decir que un manejador puede apuntar tanto a objetos de la clase padre como a los de la clase hija, ya que ambos tienen la misma interfaz. Es lo que se llama **sustitución pura.**

Hay veces en las que interesa que la interfaz de la clase hija sea más rica que la del padre. De esta forma, un manejador a objetos padre, también puede apuntar a objetos hijos, pero (debido a su definición) sólo podrá dirigir los mensajes que el padre tiene definidos.

Ej.: Sea un sistema de aire acondicionado. Sea el panel de control del aire acondicionado. El panel de control controla al sistema.

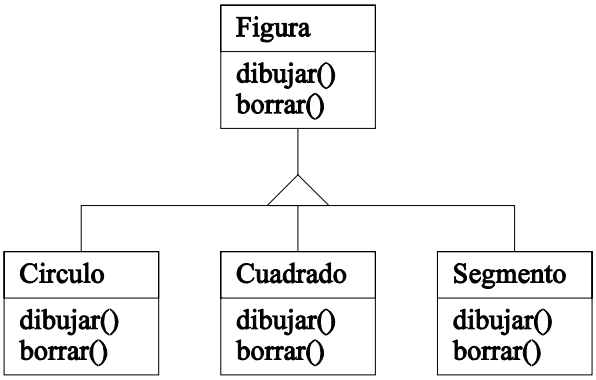
Se nos rompe el sistema de aire acondicionado. Se sustituye por una bomba de calor, que además de frío puede dar calefacción. No se cambia el panel de control.

Aunque el nuevo objeto (la bomba de calor) tiene una interfaz más completa, nuestro manejador no nos permite la comunicación.

No se puede hacer al revés (control de bomba de calor y sólo sistema de aire acondicionado).

Polimorfismo

✯ La herencia se representa mediante un árbol invertido.



✯ Sea la siguiente función Java:

```
void demo(Figura f) {
   f.borrar();
   // ...
   f.dibujar();
{
```

Esta función nos remite a cualquier Figura independientemente de ninguna subclase concreta. Si hacemos:

```
Circulo c = new Circulo;
Cuadrado q = new Cuadrado;
Segmento s = new Segmento;
demo(c);
demo(q);
demo(s);
```

todo funciona bien, puesto que un manejador de Figura también puede manejar objetos de clases hijas, como Circulo o Cuadrado, ya que, como mínimo tienen la misma interfaz. Esto es lo que se llama **upcasting.**

✯ La gran ventaja, es que si ahora decidimos hacer otra clase hija, p.ej. Triangulo, el trozo de código anterior sigue siendo útil.

Polimorfismo y Vinculación dinámica

✯ Lo curioso del código anterior es que, a pesar de todo, el código que se ejecuta en demo() no es el correspondiente a Figura, sino el correspondiente al objeto de verdad que se está gestionando, esto es, el de los Circulos cuando se llama con c, el de los Cuadrados cuando se llama con q, y el de los Segmentos cuando se llama con s.

✯ Sólo en tiempo de ejecución se sabe cuál es el verdadero objeto a que se está apuntando. Por eso, vincular el método adecuado con el tipo de objeto correspondiente, es algo que sólo puede hacerse dinámicamente.

✯ Las clases abstractas permiten crear un marco de trabajo, una especie de esqueleto, pero no permiten crear objetos de su tipo. Se emplea la palabra reservada **abstract.**

✯ Las interfaces son clases abstractas que sólo permiten declarar el prototipo de todos los métodos que deben implementar las clases que quieran ser hijas suyas. Se utiliza la palabra reservada **interface** en lugar de **class.** Para heredar de una interfaz no se usa la palabra **extends** sino la palabra **implements**.

Recolector de basura

✯ En otros lenguajes se crean objetos en distintaz zonas de memoria:

- ✗ Como variables globales. En memoria estática.
- ✗ Como variables locales. En el *stack.*
- ✗ Como variables anónimas. En el *heap.*

✯ En otros lenguajes, es necesario liberar la memoria ocupada por los objetos almacenados en el *heap.*

✯ Con Java no hay que preocuparse de recolectar los objetos inaccesibles. No existen cosas como *delete* o *free.*

✯ Java recupera automáticamente los objetos creados e inaccesibles a través de ningún manejador. Es lo que se llama **recolección de basura.**

✯ El recolector de basura actúa cuando la memoria disponible alcanza un límite demasiado pequeño.

✯ A priori es impredecible saber cuando va a comenzar a funcionar el recolector de basura.

✯ Los programas se limitan a crear todos los objetos que se necesiten, y el sistema ya se encargará de recogerlos cuando sea necesario, aunque el usuario desconoce cuando se producirá dicha recolección.

Colecciones e iteradores

✯ Java dispone de arrays. Un array es un objeto que contiene una secuencia de objetos de un tamaño concreto. ¿Qué ocurre cuando no se conoce el número de elementos a insertar?

✯ Java posee unas clases predefinidas llamadas **collections** en las que se permite insertar un número indeterminado de objetos.

✯ Entre las colecciones disponibles hay Listas, Vectores, Conjuntos, Tablas hash, etc. Para acceder a cualquiera de estas colecciones de manera uniforme, y sin depender de las características concretas de cada una de ellas, se tiene el concepto de **iterador.**

✯ La colección, sea cual sea, queda reducida al concepto de secuencia mediante el empleo de iteradores.

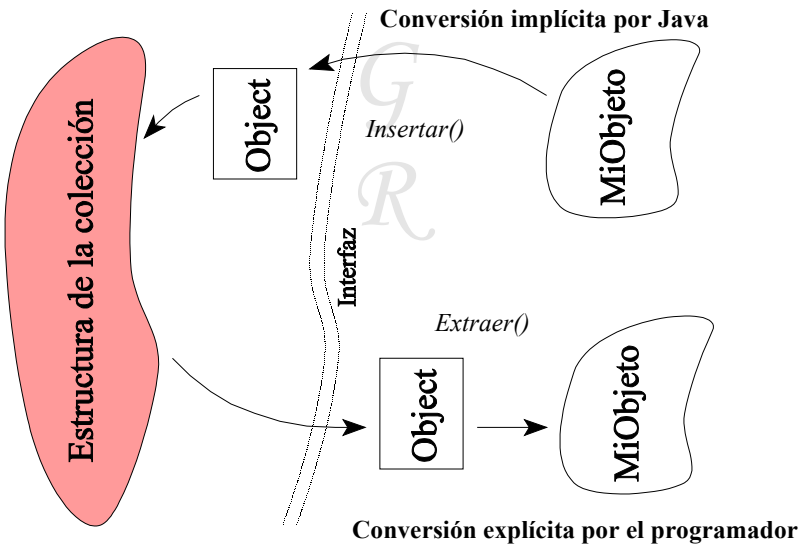
✯ De esta forma, al comienzo del diseño, se puede utilizar una lista. Si no se la maneja directamente, sino a través de un iterador, se puede decidir cambiar la Lista por un Vector, sin que cambie el código que accede a la colección.

Downcasting vs *generics*

✯ ¿Qué tipo de objetos se meten en una colección?

Se meten objetos de la clase **Object.** Pero ¡cuidado! cualquier otro objeto exista en nuestro programa, incluso los que creamos nosotros mismos, Java lo hace heredar automáticamente de **Object.** Luego por *upcasting* se puede meter cualquier objeto no primitivo.

✯ ¡Cuidado!: lo que sale de una colección es un Object, y no un objeto del mismo tipo que el que nosotros metimos. Por ello, es necesario hacerle explícitamente un *downcasting*. Java controla en tiempo de ejecución que el *downcasting* no es erróneo.



✯ Otros lenguajes permiten instanciar una clase creando una nueva clase en el que el tipo base se sustituye por uno concreto.

Manejo de excepciones

✯ En otros lenguajes el propio usuario tiene que controlar el control de errores.

✓ Esto se hace con:

- ✗ *Switches* globales que tienen que testarse cada vez que se llama a una función.
- ✗ Valores de retorno en las funciones que tienen que testarse.

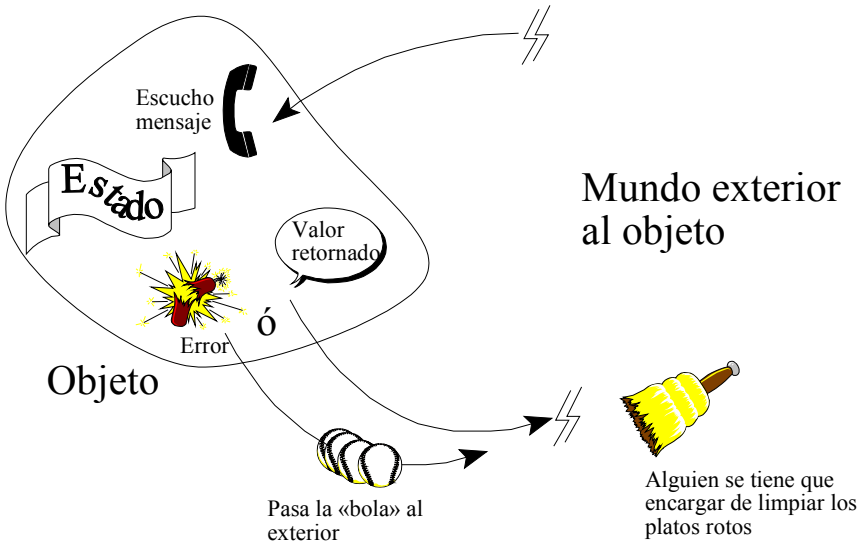
✓ Si el control es concienzudo, el resultado son programas de difícil legibilidad.

✯ Java utiliza el concepto de excepción.

Una función puede acabar correctamente, o bien lanzando una excepción.

Las excepciones se capturan en ciertos lugares bien definidos dentro del código Java.

Respuesta a un mensaje con tratamiento de excepciones



Multitarea

✯ Java emplea el concepto de *multithreading*.

✯ En lenguajes anteriores, para simular que se hacen varias cosas a la vez, había que recurrir a las interrupciones. Esto hace que el código sea muy dependiente de la máquina.

✯ Java permite la creación de tareas o subprogramas de ejecución independiente. Es lo que se llaman *threads*.

✯ Si la máquina posee varios procesadores, los *threads* se repartirán entre ellos automáticamente.

✯ Cualquier programa puede dividirse en las tareas que se quieran.

✯ Las tareas deben implementar la interfaz **Runnable.**

✯ Se utiliza la palabra **synchronized** para indicar los trozos de código que no se pueden ejecutar simultáneamente por dos *threads*.

Persistencia

✯ La persistencia permite que los objetos sobrevivan incluso después de apagar la máquina.

✯ Para ello, los objetos se almacenan en disco.

✯ Hay dos tipos de persistencia:
✗ Implícita: Todos los objetos de una clase determinada se almacenan en disco.
✗ Explícita: El programador debe indicar en el programa cuándo guardar y recuperar de disco, así como el tipo de objeto que se quiere guardar/recuperar.

✯ Es difícil de implementar debido a que los objetos pueden contener otros objetos (composición), y en disco es necesario guardar referencias para mantener dichas conexiones.

Java y los *applets*

✯ Un *applet* es un programa diseñado para ser ejecutado en un navegador como Netscape o IE.

✯ Un *applet* forma parte de una página *web*. Cuando el navegador carga la página, activa el applet que comienza así su ejecución.

- ✗ Supone una forma de distribuir programas desde un almacén de programas al ordenador cliente. Esta distribución se produce en el momento en que lo requiera el cliente, y no antes.
- ✗ El cliente obtiene la última versión del programa que haya almacenodo en el servidor de programas, sin que sean necesarias reinstalaciones ni nada por el estilo.
- ✗ El programa no se transfiere al completo, sino que se van tomando del servidor sólo aquellas partes del programa a las que el usuario quiere acceder. O sea, la descarga del programa se hace por módulos y dinámicamente a medida que lo va requieriendo el usuario en la máquina cliente.
- ✗ Para evitar programas malignos y «caballos de Troya» se imponen ciertas restricciones (quizás excesivas) a los *applets*, como la imposibilidad de leer/escribir del disco del ordenador cliente.

✯ La ventaja de un *applet* Java sobre un *script* es que los programas Java están compilados, y el cliente no tiene forma de llegar al fuente.

Java y aplicaciones *standalone*

✯ Puede emplearse el lenguaje Java para hacer aplicaciones completas, como en cualquier otro lenguaje.

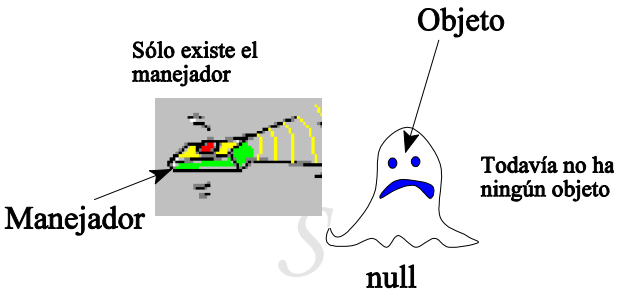
✯ Si bien los *applets* tienen ciertas restricciones en su ejecución (para evitar la proliferación de programas malignos), estas restricciones no existen en las aplicaciones independientes o *standalone*.

✯ La potencia de Java cuando se hacen aplicaciones independientes no está sólo en su portabilidad, sino en la facilidad de programar, y en que produce programas mucho más robustos, en el sentido de que:

- Se hacen potentes chequeos de tipos.
- Se hacen chequeos de límites de arrays.
- No se tiene un control directo de la memoria (no existe el concepto de puntero).
- No es necesario recoger la memoria con *dispose* o *free*, sino que el propio sistema se encarga de ello a través del recolector de basura.

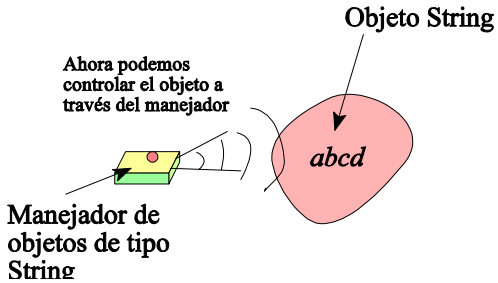
Capítulo 2
Objetos en Java

✯ En Java los objetos se gestionan a través de manejadores, y nunca directamente. P.ej., si se hace:
String s;
sólo estamos declarando el manejador, pero no ningún objeto de tipo String.



* Los manejadores sólo tienen sentido cuando se conectan a un objeto de verdad. Todos los objetos se crean **siempre** con **new**, y se conectan a los manejadores mediante el símbolo de asignación =.

```
String s;
s = new String("abcd");
```



* En este caso también estamos dando información de cómo construir el objeto de tipo String al darle un parámetro en el momento de la creación.

¿Dónde se almacenan los datos en un programa?

✯ En los registros del procesador. El programador no tiene control sobre esto.

✯ En el *stack*. Esta es la pila de registros de activación. Aquí sólo se almacenan los manejadores de objetos, pero no los objetos. Tiene la ventaja de que es muy fácil recoger los trozos de memoria que ya no se usan, y dicha recogida la hace el sistema automáticamente.

✯ En el *heap*. Es un enorme bloque de memoria del que se van cogiendo trozos según se van necesitando. Java recoge los trozos que ya no son necesarios, pero tomar y recoger trozos consume mucho tiempo.

✯ En el almacenamiento estático. Se utiliza un trozo de memoria parecido al que se emplea para guardar el código ejecutable. Java nunca guarda nada aquí.

✯ En el propio código. Cuando se usan constantes, éstas se pueden almacenar como parte del propio código ejecutable.

✯ Fuera de memoria RAM. Este método se utiliza cuando se quiere que un objeto viva incluso después de que haya finalizado la ejecución del programa.

Tipos básicos o primitivos

✯ Hay una excepción a la regla «Todo son objetos en Java». La excepción la suponen los tipos primitivos.En un tipo primitivo el tipo y el objeto son la misma cosa. Los tipos primitivos son:

Tipo primitivo	Tamaño en bits	Valor por defecto	Clase asociada
boolean	1	false	Boolean
char	16	'\u0000'	Character
byte	8	(byte)0	Byte
short	16	(short)0	Short
int	32	0	Integer
long	64	0L	Long
float	32	0.0f	Float
double	64	0.0d	Double
void	-	-	Void
		0	BigInteger
		0	BigDecimal

* El tamaño es siempre el mismo independientemente de la máquina.
* Para poder trabajar con los tipos primitivos como si fueran clases, cada tipo primitivo tiene una clase asociada, que puede almacenar el mismo tipo de información, aunque siempre a través de manejadores.

```
int i = 16;       // i es un tipo primitivo.
Integer I = new Integer(16);       // I es un manejador.
```

Tipos especiales. Arrays

✯ Los tipos BigInteger y BigDecimal permiten emplear valores enteros y decimales de cualquier precisión.
* Cualquier variable que conforme el estado de un objeto es automáticamente inicializada por Java. Si es un tipo primitivo se inicializa a **0,** y si es un manejador se inicializa a **null.** Las variables locales a una función no son inicializadas.
* Java garantiza que los elementos de un array se inicializan a cero, y que, en tiempo de ejecución, si se intenta sobrepasar los límites, se producirá un error.
* Cuando se crea un array de objetos, lo que se tiene realmente es un array de manejadores.
* En Java, un array se considera un manejador de secuencias.
* Dado que un array es un manejador, es necesario asociarle un objeto con **new**, de la forma:

```
String s1[];
s1 = new String[4];
s1[0] = new String("Uno");s1[1] = new String("Dos");
s1[2] = new String("Tres");s1[3] = new String("Cuatro");
```

* En Java una matriz de dos o más dimensiones se trata como un manejador de secuencias de arrays de una dimensión. Además, los subarrays del mismo nivel no tienen porqué tener las mismas dimensiones.

```
String s2[][];
s2 = new String[3][];
s2[0] = new String[10];
s2[1] = new String[15];
s2[2] = new String[20];
```

* También se pueden crear arrays de tipos primitivos.

Recolector de basura

✯ Java se encarga de recoger todos los objetos que ya no estén siendo usados por nuestro programa.

* Automáticamente se activa el recolector de basura cuando la memoria se está agotando.

* La recolección de basura puede consumir bastante tiempo.

* La incertidumbre sobre la recolección de basura hace que no se pueda emplear Java en sistemas en tiempo real.

* Hay implementaciones del motor de ejecución de Java, en las que el usuario tiene cierto control sobre los tiempos de ejecución del recolector de basura.

Creación de tipos nuevos: class

✯ Para crear un tipo nuevo se usa la palabra **class** seguida del nombre del tipo.

```
class MiClase {
	/* Cuerpo de la clase */
}
```

Y los objetos se crean como:

```
MiClase m = new MiClase();
```

* En el cuerpo de una clase se colocan:
- Datos miembro (campos). Componen el estado. Pueden ser objetos (manejadores) o tipos primitivos. Si son objetos deben ser inicializados en una función especial llamada **constructor**.
- Funciones miembro (métodos).
* Cada objeto mantiene su propia copia de los datos miembros.
* Ej.

```
class SoloDatos {
	int i;
	float f;
	char c;
}
```

* Esta clase viene a ser como un registro de C. Esta clase no hace nada, pero podemos crear manejadores a objetos SoloDatos, así como crear objetos SoloDatos.

```
SoloDatos sd = new SoloDatos();
```

* Los datos miembro se acceden como los campos de los registros en C:

```
sd.i = 16; sd.f =1.2365f; sd.c = 'g';
```

Métodos, parámetros y valores de retorno.

✯ Los métodos de una clase no son más que funciones en el sentido de C.
* Los métodos determinan los mensajes que un objeto puede recibir.
* El cuerpo de un método se fundamenta en que cuando se ejecute lo hará sobre un objeto concreto.
* Cuando se quiere ejecutar un método, se le asocia el objeto sobre el que se quiere ejecutar.
* Como cualquier función, un método puede recibir cualquier número y tipo de parámetros. Podemos decir que siempre hay un objeto implícito que es el que recibe mensaje.
* Como en cualquier otra situación en Java, cuando se pasan objetos como parámetros, lo que realmente se están pasando son manejadores.
* Un método puede retornar cualquier tipo de objeto, incluido un tipo primitivo. Si no se quiere retornar nada, se indica el tipo **void**. Los valores se retornan con **return**.
* Ej.:

```
class MiClase {
	int a;
	short miMetodo(char c, float f) {
		// Cuerpo del método.
		return (short)3;
	}
}
MiClase mc = new MiClase();
mc.a = 23;
short s = mc.miMetodo('h', 3.3f);
```

Utilización de componentes

✯ Para reutilizar una clase debemos emplear la palabra clave **import**, y decirle al compilador dónde la debe encontrar. P.ej.:
import java.util.Vector;
quiere decir que queremos usar Vector.class que está en el subdirectorio java\util\

* Pero, ¿de dónde cuelga java\util\? Pues Java lo buscará a partir de los directorios que se indiquen en la variable del entorno del sistema operativo llamada CLASSPATH. Java busca en los directorios en el orden en que éstos se especifiquen en CLASSPATH.

* Si queremos utilizar todas las librerías que cuelguen de un subdirectorio concreto se emplea:
import java.util.*;

* Se pueden poner tantas cláusulas **import** como se quieran.

Cosas *static*

* Cuando en una clase se dice que algo es estático (un dato o una función miembro), quiere decir que no está asociado a ningún objeto particular perteneciente a esa clase.

* Dado que una función **static** no se ejecuta sobre ningún objeto concreto, dentro de la misma no se tiene acceso a los elementos no estáticos, ya que éstos sólo tienen sentido cuando se trabaja con instancias concretas de una clase.

* Por el mismo motivo, para referenciar un elemento static, en lugar de utilizar el manejador de un objeto ya creado, se puede emplear el propio nombre de la clase.

* P.ej.:

```
class TestStatic {
        static int i = 47;
}
TestStatic ts1 = new TestStatic();
TestStatic ts2 = new TestStatic();
// ts1.i y ts2.i valen lo mismo, esto es 47
ts1.i++;
// ts1.i y ts2.i valen lo mismo, esto es 48
TestStatic.i++; // Esta es otra forma de referirse a cosas estáticas
// ts1.i y ts2.i valen lo mismo, esto es 49
```

* static es la forma de simular funciones (del estilo de las de C), en un lenguaje orientado a objetos como Java.

El primer programa

```
// Propiedades.java
import java.util.*;

public class Propiedades {
        public static void main(String[] args) }
                System.out.println(new Date());
                Properties p = System.getProperties();
                p.list(System.out);
                System.out.println("---Uso de memoria:");
                Runtime rt = Runtime.getRuntime();
                System.out.println(      "Total de memoria = "
                                                + rt.totalMemory()
                                                + "Memoria libre = "
                                                + rt.freeMemory());
        }
}
```

* El primer **import** mete todas las librerías del directorio java\util\
* System posee un dato miembro estático llamado **out** que está definido como un objeto del tipo **PrintStream**.
* La clase que se desea ejecutar debe tener una función miembro estática llamada **main()** que recibe como argumento un array de String, que serán los parámetros que se le pasen desde la línea de comandos.
* **Date** es un objeto que se crea con el único objetivo de pasárselo como parámetro a System.out.println.
* El signo + indica concatenación cuando se utiliza con cadenas de caracteres.
* El signo + se puede emplear para concatenar cadenas de caracteres junto con números (automáticamente los números se convierten a su representación textual).

Operadores Java

* Los operadores permitidos por Java son aproximadamente los mismos que los que posee C++, y tienen la precedencia que se especifica a continuación:

Mnemónico	Tipo de operador	Operador
Ulcer	Unarios	+ - ++ --
Addicts	Aritméticos y de desplazamiento	* / % + - << >> >>>
Really	Relacionales	> < >= <= == !=
Like	Lógicos	&& || & | ^
C	Condicional	A > B? X : Y
A lot	Asignación	= (y asignaciones compuestas)

* Cuando se asignan tipos primitivos, dado que no intervienen manejadores, lo que se hacen son copias de los datos concretos.

* No ocurre lo mismo cuando se trabaja con manejadores de objetos. En estos casos, lo que ocurre es que el manejador *l-valor* referencia al mismo objeto que el manejador *r-valor*.

Ejemplo de uso de operadores

```
//: MathOps.java
// Copyright (c) Bruce Eckel, 1998. Source code file from the book "Thinking in Java"
// Demonstrates the mathematical operators
import java.util.*;

public class MathOps {
  // Create a shorthand to save typing:
  static void prt(String s) {    System.out.println(s);  }
  // shorthand to print a string and an int:
  static void pInt(String s, int i) {    prt(s + " = " + i);  }
  // shorthand to print a string and a float:
  static void pFlt(String s, float f) {    prt(s + " = " + f);  }
  public static void main(String[] args) {
    // Create a random number generator, seeds with current time by default:
    Random rand = new Random();
    int i, j, k;
    // '%' limits maximum value to 99:
    j = rand.nextInt() % 100;
    k = rand.nextInt() % 100;
    pInt("j",j);  pInt("k",k);
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k * j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Floating-point number tests:
    float u,v,w;  // applies to doubles, too
    v = rand.nextFloat();
    w = rand.nextFloat();
    pFlt("v", v); pFlt("w", w);
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);
    // the following also works for char, byte, short, int, long and double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
}
```

Comparación e igualdad de objetos

```
//: Equivalence.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"

public class Equivalence {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
}
```

* Cuando se comparan manejadores, no se está preguntando realmente por el contenido de los objetos apuntados, sino si ambos manejadores se refieren físicamente al mismo objeto.

* Para que funcione bien, se utiliza el método **equals()** que debe ser reescrito para cada clase de objetos que se quieran comparar.

* Java utiliza cortocircuito en los operadores lógicos.

* Los operadores de bit (&, | y ~) para valores **boolean** se comportan igual que los relacionales (&&, || y !), excepto por el hecho de que no incorporan cortocircuito.

Casting y literales

* Java permite cualquier conversión de un tipo primitivo a otro primitivo, excepto para los **boolean**.

* Cuidado con las conversiones en que el tipo destino permite reprepresentar un rango de valores más pequeño que el origen.

```
//: Literals.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"

class Literals {
  char c = 0xffff; // max char hex value
  byte b = 0x7f; // max byte hex value
  short s = 0x7fff; // max short hex value
  int i1 = 0x2f; // Hexadecimal (lowercase)
  int i2 = 0X2F; // Hexadecimal (uppercase)
  int i3 = 0177; // Octal (leading zero)
  // Hex and Oct also work with long.
  long n1 = 200L; // long suffix
  long n2 = 200l; // long suffix
  long n3 = 200;
  //! long l6(200); // not allowed
  float f1 = 1;
  float f2 = 1F; // float suffix
  float f3 = 1f; // float suffix
  float f4 = 1e-45f; // 10 to the power
  float f5 = 1e+9f; // float suffix
  double d1 = 1d; // double suffix
  double d2 = 1D; // double suffix
  double d3 = 47e47d; // 10 to the power
}
```

* Si se hacen operaciones con valores de tipos más pequeños que el **int** (char, byte o short), dichos valores son «promocionados» a **int** antes de hacer la operación, y el resultado es **int**; luego para asignar el resultado a algo de tipo más pequeño debe ser de convertido de nuevo.

Sentencias de ctrl. de flujo

* Las condiciones deben tener siempre un valor de tipo **boolean**.

*** if-else**.

```
if(expresión boolean)
        sentencia
```

ó

```
if(expresión boolean)
        sentencia
else
        sentencia
```

*** return**

Hace que se acabe el método que la ejecuta, y si el método debe retornar algún valor, dicho valor debe seguir a la sentencia **return**.

*** switch-case**

```
switch(selector int) {
        case valor1 : sentencia; break;
        case valor2 : sentencia; break;
        case valor3 : sentencia; break;
        // ...
        default : sentencia;
}
```

La parte **default** es opcional.

Sentencias de ctrl. de flujo

*** while(expresión boolean)**
sentencia
*** do**
sentencia
while(expresión boolean);

*** break** y **continue**

break se sale del bucle en el que se encuentra.
continue acaba el ciclo actual y comienza el siguiente.

break y **continue** pueden referenciar a un bucle más externo de aquél en el que se encuentran:

```
label1:
iteración_externa {
        iteración_interna {
                ...
                break; // Se sale de iteración_interna
                ...
                continue; //Hace el siguiente ciclo de iteración_interna
...
                break label1; // Se sale de iteración_externa
                ...
                continue label1; //Hace el siguiente ciclo de iteración_externa
        }
}
```

*** for(sentencias de inicio;**
expresión boolean;
sentencias de paso)
sentencia

equivale a:

```
sentencias de inicio;
while(expresión boolean) {
        sentencia;
        sentencias de paso;
}
```

Capítulo 3
Inicialización de objetos. Constructores

* Si una clase tiene un método especial al que se le llama **constructor**, Java llama *automáticamente* a este método cada vez que se crea un objeto de esa clase.

* Un constructor es un método especial. Tiene el mismo nombre que la clase a la que pertenece, y no retorna ningún tipo (ni siquiera **void**).

```
//: SimpleConstructor.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Demonstration of a simple constructor
package c04;

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
```

* El modificador **package** obliga a que el fichero SimpleConstructor.class resultante esté en el directorio que se indique, para que se pueda ejecutar.

Constructores con parámetros

* Un constructor puede tener parámetros que indiquen cómo hay que crear al objeto. Cuando se crea el objeto hay que pasarle dicho argumento.

```
//: SimpleConstructor.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Demonstration of a constructor with one argument
package c04;

class Rock {
  Rock(int j) { // This is the constructor
    System.out.println("Creating Rock number" + j);
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock(i);
  }
}
```

* Si el programador no crea ningún constructor, el sistema crea uno por defecto sin parámetros. Si el programador crea algún constructor, el sistema ya no crea ninguno por defecto.

* El sistema llama automáticamente al método **finalize()** cuando va a recoger a un objeto como basura, a través del recolector de basura.

Sobrecarga de funciones

* Un método es el nombre que se le da a una acción, o sea, a un trozo de código.

* A menudo la misma palabra denota acciones distintas, o sea, su significado está sobrecargado. Se distingue un significado de otro a través del contexto.

* Java permite sobrecargar los nombres de los métodos, siempre y cuando sus listas de parámetros sean distintas, ya sea en longitud y/o en tipos. **No se puede sobrecargar únicamente sobre el tipo de valor retornado.**

```
//: OverloadingOrder.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Overloading based on the order of the arguments.

public class OverloadingOrder {
  static void print(String s, int i) {
    System.out.println(
      "String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println(
      "int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
  }
}
```

Sobrecarga del constructor

* De esta forma, podemos sobrecargar el constructor de una clase.

```
//: Overloading.java
// Copyright (c) Bruce Eckel, 1998. Source code file from the book "Thinking in Java"
// Demonstration of both constructor and ordinary method overloading.
import java.util.*;

class Tree {
  int height;
  Tree() {
    prt("Planting a seedling");
    height = 0;
  }
  Tree(int i) {
    prt("Creating new Tree that is "
        + i + " meters tall");
    height = i;
  }
  void info() {
    prt("Tree is " + height
        + " meters tall");
  }
  void info(String s) {
    prt(s + ": Tree is "
        + height + " meters tall");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
}
```

La palabra clave *this*

* La palabra clave *this* se emplea dentro de un método como manejador al objeto que recibe el mensaje.

* Cuando desde un método de una clase se llama a otro método de la misma clase, no hace falta poner **this**, ya que por defecto se supone. Ej.:

```
class Ciruela {
        void coger() { /*...*/ }
        void comer() { coger(); /*...*/ }
}
/* Dentro de comer() se podría haber hecho this.coger(), pero no hace falta. Se supone por defecto. */
```

* Ejemplo:

```
//: Leaf.java
// Source code file from the book "Thinking in Java"
// Simple use of the this keyword
public class Leaf {
  private int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
}
```

* Ahora se comprende mejor qué es un método **static**: es aquél en el que **this** no tiene sentido.

Llamada a un constructor desde otro constructor

* Cuando se ha sobrecargado el constructor, hay veces en que puede ser conveniente llamar a un constructor desde otro, para evitar repetir código.

* La palabra **this** dentro de un constructor permite hacer llamadas a otros constructores. Ej.:

* Si un constructor llama a otro, debe ser la primera acción que haga.

* Un constructor sólo puede llamar a otro una vez.

* **this** también se puede utilizar para distinguir entre los nombres de los parámetros y los de los campos.

Uso de *this* en un constructor

```
//: Flower.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Calling constructors with this
public class Flower {
  private int petalCount = 0;
  private String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    System.out.println("Constructor w/ int arg only, petalCount= " + petalCount);
  }
  Flower(String ss) {
    System.out.println("Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
    this.s = s; // Another use of this
    System.out.println("String and int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println("default constructor (no args)");
  }
  void print() {
//!    this(11); // Not inside non-constructor!
    System.out.println("petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
}
```

Inicialización de datos estáticos

* Adivinar el resultado de lo siguiente:

```
//: StaticInitialization.java
// Specifying initial values in a class definition.
class SuperCopa {
  SuperCopa(int marker) {    System.out.println("SuperCopa(" + marker + ")");  }
  void f(int marker) {    System.out.println("f(" + marker + ")");  }
}
class Tabla {
  static SuperCopa b1 = new SuperCopa(1);
  Tabla() {
    System.out.println("Tabla()");
    b2.f(1);
  }
  void f2(int marker) {    System.out.println("f2(" + marker + ")");  }
  static SuperCopa b2 = new SuperCopa(2);
}
class Liga {
  SuperCopa b3 = new SuperCopa(3);
  static SuperCopa b4 = new SuperCopa(4);
  Liga() {
    System.out.println("Liga()");
    b4.f(2);
  }
  void f3(int marker) {    System.out.println("f3(" + marker + ")");  }
  static SuperCopa b5 = new SuperCopa(5);
}
public class StaticInitialization {
  public static void main(String[] args) {
    System.out.println("Creating new Liga() in main");
    new Liga();
    System.out.println("Creating new Liga() in main");
    new Liga();
    t2.f2(1);
    t3.f3(1);
  }
  static Tabla t2 = new Tabla();
  static Liga t3 = new Liga();
}
```

Sobrecarga del constructor

* Java permite colococar un código de inicialización de la clase. Esto se hace precediendo el trozo de código de la palabra **static**:

```
class Prueba {
        int i;
        static {
                /* Aquí va el código que se ejecutará en cuanto se cargue la clase. */
                i = 47;
        }
}
```

* En Java se pueden inicializar los datos miembros de un objeto, aunque no sean estáticos. En el siguiente ejemplo se omite la palabra **static**:

```
//: Mugs.java  Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Java 1.1 "Initialización de objetos"
class Cordero {
  Cordero(int marker) {
    System.out.println("Cordero(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}
public class Corderos {
  Cordero c1;
  Cordero c2;
  {
    c1 = new Cordero(1);
    c2 = new Cordero(2);
    System.out.println("c1 & c2 initialized");
  }
  Corderos() {
    System.out.println("Corderos()");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Corderos x = new Corderos();
  }
}
```

Inicialización de arrays

* En Java un array es un objeto.
* Cuando se declara un array en realidad estamos declarando un manejador a un array.
* Los arrays se declaran como en C, excepto por el hecho de que en la declaración no se indica el tamaño.

```
int[] a;
```

El array debe ser creado posteriormente. Puede hacerse de dos formas:

- Mediante el **new** correspondiente, como para cualquier otro objeto.

```
int a[] = new int[10];
int b[];
b = new int[7];
```

- Enumerando el valor de sus elementos entre llaves en el momento de la declaración. Esto lleva implícito el **new**.

```
int a[] = {1, 2, 3 , 4, 5, 6, 7,}; // La última coma es opcional
```

* Un array de objetos es en realidad un array de manejadores.

```
String s[] = new String[3];
s[0] = new String("Posición 1");
s[1] = new String("Posición 2");
s[2] = new String("Posición 3");
```

* El primer elemento de un array está en la posición 0.
* En Java no existen arrays multidimensionales, sino arrays de arrays cuantas veces se quiera.
* Las dimensiones que están al mismo nivel no tienen porqué tener la misma longitud. P.ej.:

```
int[][] a = new int[3][];
a[0] = new int[5];
a[1] = new int[13];
a[2] = new int[7];
```

Inicialización de arrays

* El número de elementos de un array viene dada por un dato miembro que se llama **length**. En el caso anterior, a[0][1] vale 13.

* El número de elementos que va a poseer un array no tiene porqué conocerse en tiempo de compilación. Ej.:

```
//: ArrayNew.java
// Copyright (c) Bruce Eckel, 1998
/* Source code file from the book "Thinking in
Java" */
// Creating arrays with new.
import java.util.*;

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    prt("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      prt("a[" + i + "] = " + a[i]);
  }
  static void prt(String s) {
    System.out.println(s);
  }
}
```

* Hay dos formas de inicializar en el momento de la declaración:

```
Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
```

```
Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
```

* La segunda forma de inicialización, en conjunción con que todo objeto hereda tarde o temprano de **Object** permite crear métodos en los que el número de argumentos es variable. Ej.:

```
//: VarArgs.java
// Copyright (c) Bruce Eckel, 1998
// Using the Java 1.1 array syntax
// to create variable argument lists
class A { int i; }
public class VarArgs {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new VarArgs(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"one", "two", "three" });
    f(new Object[] {new A(), new A(), new A()});
  }
}
```

Capítulo 4
Reutilización de código

* Antes de ver conceptos para la reutilización de código, es necesario estudiar los elementos que nos da Java para distribuir el código, y para establecer la interfaz de cada clase.

* **package**. Un paquete es lo que cogemos cuando ponemos en nuestro programa una sentencia como:
import java.util.*;

Esto accede al subdirectorio java\util\ y pone a nuestra disposición todos los *.class que haya allí.

O sea, un paquete es el conjunto de ficheros *.class que está en un mismo subdirectorio.

NOTA: Java utiliza el punto (.) como separador de subdirectorios, y posteriormente lo sustituye por \ ó / según el sistema operativo (DOS ó Unix).

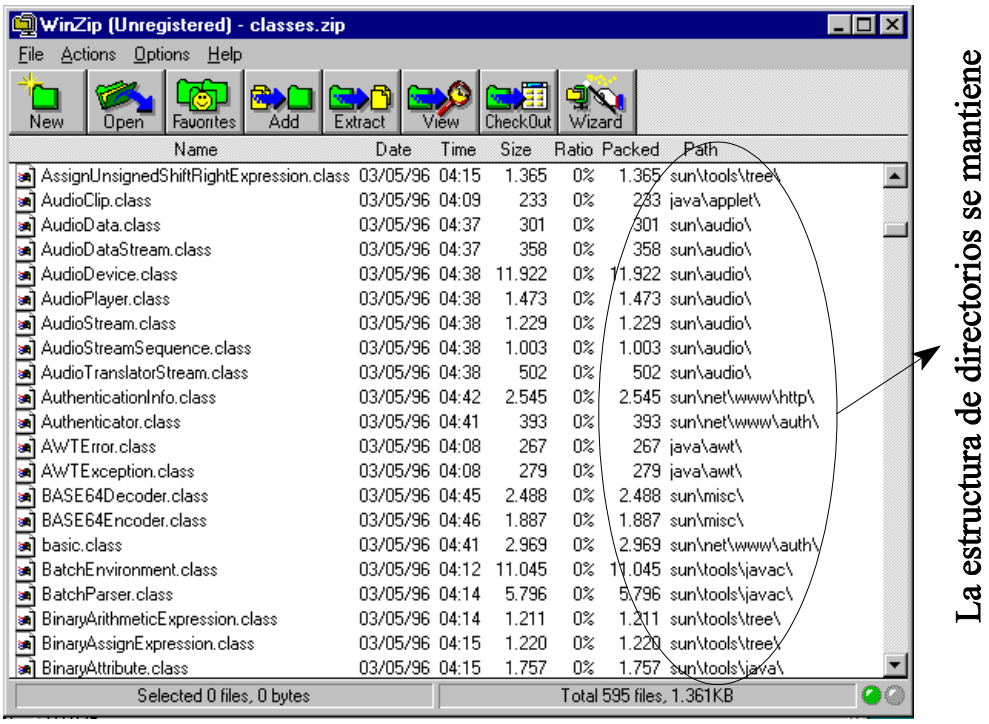
* Cuando se crea un fichero File.java (también llamado **unidad de compilación**), este fichero debe contener:
Obligatoriamente:
-Una clase pública (**public**) con el mismo nombre que el fichero (quitando el .java).
Opcionalmente:
- Cualesquiera otras clases, que no pueden ser públicas.

* Cuando se compila el fichero File.java se genera un fichero .class por cada una de las clases que contuviera File.java.

Package

* **package** es una directiva que (si se emplea) indica donde se van a colocar los .class resultantes de la compilación.
* **package** obliga a que los .class generados deban estar en el directorio especificado para su correcta ejecución.

* Como resultado de hacer un proyecto grande, podemos obtener cientos de clases. Para evitar tener cientos de ficheros, Java permite agruparlos en un fichero .zip sin comprimir, siempre que internamente se mantenga la estructura interna de subdirectorios.



* Si cuando se accede al fichero .class, para proceder a su ejecución, en el mismo directorio está su fichero fuente .java, el propio motor de ejecución comprueba las fechas de creación, y si el fuente es más reciente que el .class, se recompila el fuente en tiempo de ejecución, obteniéndose un .class actualizado que, entonces, se ejecuta.

Los ficheros *.jar* Paquetes y colisiones

* Las clases que componen nuestro proyecto también se pueden agrupar comprimidos en un fichero .jar, a través de la utilidad Java, jar.exe. Un fichero .jar, a diferencia de uno .zip, posee las siguientes características:

  - Comprime de verdad los ficheros que contiene.
  - Puede contener cualquier tipo de fichero, además de los .class.
  - Se transfiere entero en operaciones a través de la red.

* Los ficheros .jar y .zip deben declararse explícitamente en la variable del entorno CLASSPATH. En Java 2 también basta con meterlos en el directorio *jdk1.3\jre\lib\ext*, reservado para las extensiones de las librerías de Java.

* En esta variable se colocan además los directorios base en los que se desea que Java busque cuando se quiere importar una librería o una clase concreta.

* Se produce una **colisión** cuando se importan dos paquetes, que poseen una clase con el mismo nombre, y además se hace uso de dicha clase.

* Cuando se ejecuta un .class que no forma parte de ningún paquete, Java asume que ese directorio conforma el **paquete por defecto** para esa clase.

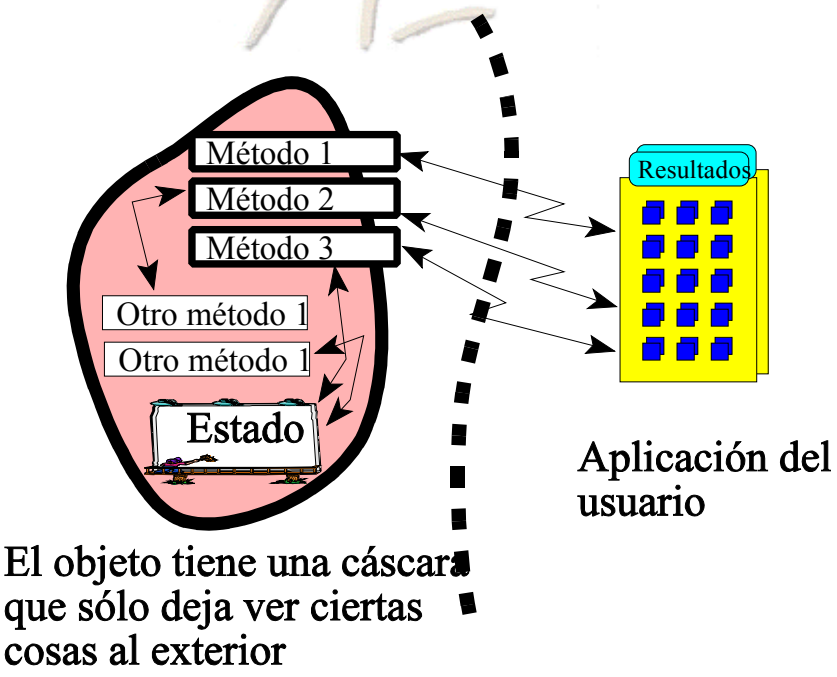
* Es un buen momento para ver la tabla de accesibilidad del capítulo 1.

Visibilidad

* Hay dos motivos para restringir al exterior el acceso a los miembros de una clase:

1.- Que los que usan dicha clase no tengan acceso a cosas que no deben tocar. O sea, cosas que forman parte de las «tripas» de la clase, pero que al usuario no le interesan para nada, y que no forman parte de la interfaz de la clase. Para el usuario también supone un alivio porque así no se pierde en un mar de detalles que no le interesan.

2.- Permitir al diseñador de la clase cambiar la implementación sin que por ello los programas del usuario se tengan que modificar. El usuario se limita a emplear la interfaz, sin importarle para nada los detalles de implementación.



Composición

* Como vimos en su momento hay dos formas de reutilizar código. La primera consiste en meter objetos dentro de otros objetos. Es algo así como crear un registro.

```
//: SprinklerSystem.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Composition for code reuse
package c06;

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source = new WaterSource();
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
}
```

* Para poder visualizar un objeto de la clase WaterSource con System.out.println(), es necesario que posea un método que lo convierta en **String.** Este método se llama toString(), y debe retornar un **String.**

Herencia

* Este es el método básico de reutilización de código en la programación O.O.
* La herencia permite crear clases que son parecidas a otra clase base y que, si quieren, pueden tener más métodos, o sea, extender funcionalmente a la clase base. Tanto en uno como en otro caso, para heredar se emplea la palabra clave **extends**.

```
//: Detergente.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Inheritance syntax & properties
class Limpiador {
  private String s = new String("Limpiador");
  public void append(String a) { s += a; }
  public void diluir() { append(" diluir()"); }
  public void aplicar() { append(" aplicar()"); }
  public void esperar() { append(" esperar()"); }
  public void print() { System.out.println(s); }
  public static void main(String[] args) {
    Limpiador x = new Limpiador();
    x.diluir(); x.aplicar(); x.esperar();
    x.print();
  }
}
public class Detergente extends Limpiador {
  // Change a method:
  public void esperar() {
    append(" Detergente.esperar()");
    super.esperar(); // Call base-class version
  }
  // Add methods to the interface:
  public void lavar() { append(" lavar()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergente x = new Detergente();
    x.diluir();
    x.aplicar();
    x.esperar();
    x.lavar();
    x.print();
    System.out.println("Testing base class:");
    Limpiador.main(args);
  }
}
```

Herencia

* El ejemplo anterior nos ilustra muchas cosas:

1.- La clase que tiene el mismo nombre que el fichero (Detergente) tiene que ser pública.

2.- Si hay otras clases, no pueden ser públicas.

3.- El operador += está sobrecargado para los **String.**

4.- Tanto Limpiador como Detergente poseen un public static void main(). Esto permite comprobar de forma independiente cada una de las clases con pocos cambios en el fichero .java.

5.- Cuando se invoca la ejecución con **java Detergente,** sólo se ejecuta el main() de dicha clase.

6.- Dado que Detergente hereda de Limpiador, hereda automáticamente todos sus miembros.

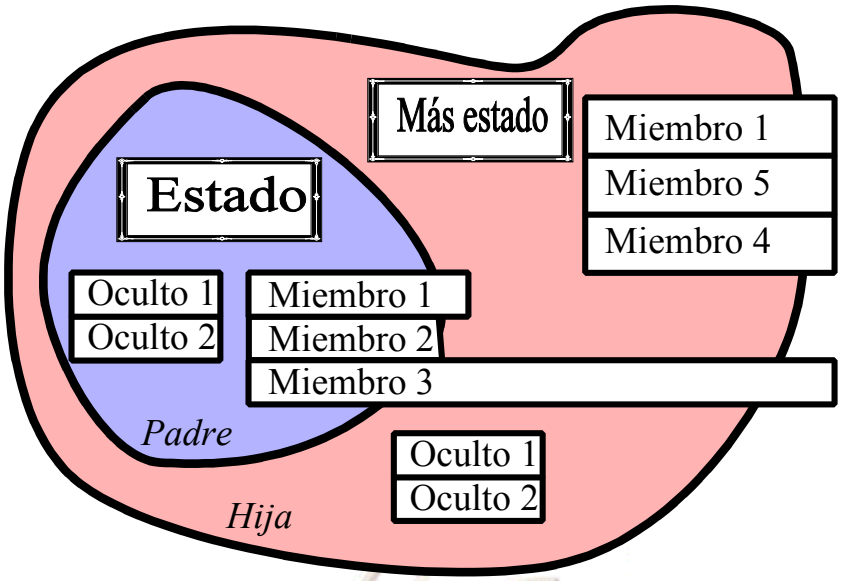
7.- En la clase hija es posible reescribir un método que ya estaba en la clase padre (esperar()).

8.- Java permite llamar a un método de la clase padre (aunque esté reescrito), a través de la palabra clave **super**, que se refiere a la *superclase*.

9.- La clase base se puede extender con nuevos métodos y/o miembros (lavar()).

Visibilidad

* Ahora que sabemos cómo funciona la herencia, y con la siguiente figura en mente, podemos hacernos una idea de qué es un objeto hijo.



* Vamos a explicar de nuevo los modificadores de visibilidad.
public: indica que cualquiera tiene acceso a ese miembro. En el caso anterior Miembro 3 es público.
protected: indica que las clases hijas tienen acceso a él. Miembro 2 es **protected**, lo cual quiere decir que Hija tiene acceso a él, pero que no es visible para nadie más.
private: Nadie tiene acceso (a no ser que también se indique **protected**).
-nada-: Sólo tienen acceso las clases hijas, y las clases del mismo paquete.

* Se empleará *composición* cuando se quiera crear una clase nueva, pero el usuario de dicha clase sólo deba ver la interfaz de la nueva clase, y no la del objeto componente. No así con la herencia.

Inicialización en herencia

* Para que el constructor de las clases hijas tenga sentido, es necesario que la clase base inicialice las «tripas» de aquello a lo que no tiene acceso la clase hija. Por ello, Java inserta automáticamente un llamada al constructor por defecto (el que no tiene parámetros) de la clase base, justo antes de hacer cualquier otra cosa en el constructor de la hija.

```
//: Cartoon.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Constructor calls during inheritance
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}
class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}
public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
```

Inicialización en herencia

* Si la clase padre no tiene constructor sin parámetros, lo primero que debe hacer el constructor de la clase hija es una llamada a uno de los constructores parametrizados de la clase padre. Para ello se hace uso de la palabra **super.**

```
//: Chess.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
}
```

Composición y herencia

* Predecir el resultado de:

```
//: PlaceSetting.java
// Copyright (c) Bruce Eckel, 1998
/* Source code file from the book "Thinking in
Java" */
// Combining composition and inheritance
class Plato {
  Plato(int i) {
    System.out.println("Plato constructor");
  }
}
class PlatoParaCenar extends Plato {
  PlatoParaCenar(int i) {
    super(i);
    System.out.println(
      "PlatoParaCenar constructor");
  }
}
class Utensilio {
  Utensilio(int i) {
    System.out.println("Utensilio constructor");
  }
}
class Cuchara extends Utensilio {
  Cuchara(int i) {
    super(i);
    System.out.println("Cuchara constructor");
  }
}
class Tenedor extends Utensilio {
  Tenedor(int i) {
    super(i);
    System.out.println("Tenedor constructor");
  }
}
class Cuchillo extends Utensilio {
  Cuchillo(int i) {
    super(i);
    System.out.println("Cuchillo constructor");
  }
}
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}
public class PlaceSetting extends Custom {
  Cuchara sp;
  Tenedor frk;
  Cuchillo kn;
  PlatoParaCenar pl;
  PlaceSetting(int i) {
    super(i + 1);
    sp = new Cuchara(i + 2);
    frk = new Tenedor(i + 3);
    kn = new Cuchillo(i + 4);
    pl = new PlatoParaCenar(i + 5);
    System.out.println(
      "PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
}
```

Upcasting

* Una de las características más importantes de la herencia, es que un objeto hijo se puede emplear en cualquier lugar donde se espere un objeto padre.

```
//: InstrumentoDeViento.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Inheritance & upcasting
import java.util.*;
class Instrumento {
  public void play() {}
  static void tune(Instrumento i) {
    // ...
    i.play();
  }
}
// InstrumentoDeViento objects are Instrumentos
// because they have the same interface:
class InstrumentoDeViento extends Instrumento {
  public static void main(String[] args) {
    InstrumentoDeViento flauta = new InstrumentoDeViento();
    Instrumento.tune(flauta); // Upcasting
  }
}
```

* Si tune() funciona para Instrumentos, también debe funcionar para cualquier cosa que herede de Instrumento. Al proceso automático que convierte de InstrumentoDeViento en Instrumento, se le llama *upcasting*.

* *Upcasting* es una acción segura, ya que las clases hijas siempre tienen que tener como mínimo todas las operaciones de la clase padre.

La palabra clave *final*

* **final** viene a decir: «Esto es inalterable». Esta palabra clave puede preceder a:

- Datos miembro. Es una forma de identificar a los valores primitivos constantes. A un dato miembro final se le puede asignar un valor en el momento de su declaración, o una sola vez en el cuerpo de cualquier método. Si el dato miembro es un manejador de objetos, quiere decir que ese manejador siempre apuntará al mismo objeto físico; sin embargo el objeto en sí, sí puede ser modificado.

  Que un dato miembro sea final no quiere decir que su valor se conozca en tiempo de compilación.

- Parámetros de función. Quiere decir que el parámetro es de solo lectura.

- Métodos. Evita que cualquier clase hija lo pueda reescribir. Además permite al compilador **javac** convertir las llamadas a dicho método en código *inline*.

- Clases. Quiere decir que nadie puede heredar de esta clase.

Polimorfismo

* El polimorfismo o vinculación dinámica permite distinguir entre objetos de clases distintas que heredan de una misma clase base.

```
//: Musica.java
// Copyright (c) Bruce Eckel, 1998
// Inheritance & upcasting
package c07;
class Nota {
  private int valor;
  private Nota(int val) { valor = val; }
  public static final Nota
    mediaFa = new Nota(0),
    FaAguda = new Nota(1),
    FaGrave = new Nota(2);
} // Etc.
class Instrumento {
  public void tocar(Nota n) {
    System.out.println("Instrumento.tocar()");
  }
}
// InstrumentoDeViento objects are Instrumentos because they have the same
interface:
class InstrumentoDeViento extends Instrumento {
  // Redefine interface method:
  public void tocar(Nota n) {
    System.out.println("InstrumentoDeViento.tocar()");
  }
}
// InstrumentoDeCuerda objects are Instrumentos because they have the same
interface:
class InstrumentoDeCuerda extends Instrumento {
  // Redefine interface method:
  public void tocar(Nota n) {
    System.out.println("InstrumentoDeCuerda.tocar()");
  }
}
public class Musica {
  public static void Afinar(Instrumento i) {
    i.tocar(Nota.mediaFa);
  }
  public static void main(String[] args) {
    InstrumentoDeViento flauta = new InstrumentoDeViento();
    InstrumentoDeCuerda piano = new InstrumentoDeCuerda();
    Instrumento maracas = new Instrumento();
    Instrumento guitarra = new InstrumentoDeCuerda();
```

Polimorfismo

```
    Afinar(flauta); // Upcasting
    Afinar(piano); // Upcasting
    Afinar(maracas); // Upcasting
    Afinar(guitarra); // Upcasting
  }
}
```

* Si observamos el resultado, vemos que en el momento en que se llama a afinar(), todos los objetos pasan a ser manejados por un manejador de Instrumento. Sin embargo, cuando en afinar() se llama a tocar(), lo que «suena» es cada instrumento concreto.

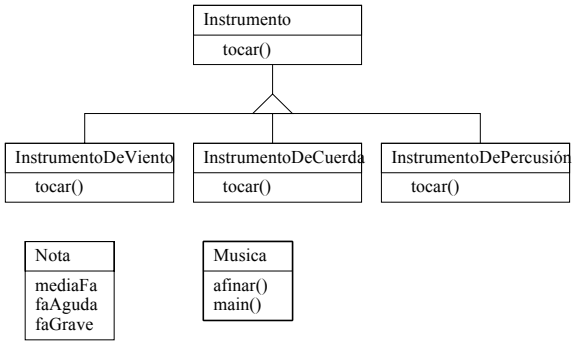
* En otros lenguajes de programación, para que esto funcione hay que hacer un enorme **switch** en el que se pregunta por todos y cada uno de los tipos posibles.

* El método de Java tiene la ventaja de que si se inserta un nuevo instrumento (p.ej., InstrumentoDePercusion), no es necesario modificar el código de las clases que ya están construidas.

* Aquí se emplea el *upcast* para «olvidar» intencionadamente el tipo concreto de un objeto, tratarlo como si fuera un Instrumento cualquiera (con la interfaz de Instrumento). De esta forma, el manejador de Instrumentos pasa a ser **polimórfico** en el sentido de que puede controlar no sólo Instrumentos sino cualquier objeto que herede de Instrumento.

* Cuando se llame a uno de sus métodos, se producirá lo que se llama **vinculación dinámica**, o sea, se llama al método de la clase a que pertenezca verdaderamente el objeto manejado.

Polimorfismo



* Cuando se llama a afinar(), se sabe que se tiene un Instrumento, pero no se sabe de qué tipo. afinar() llama a tocar(), y deja que el sistema vincule dinámicamente (en tiempo de ejecución) cada objeto con su verdadero método.
* Podemos ponérselo más complicado al sistema con el siguiente programa.

```
// Dificil.java
// No se sabe de qué tipo será cada Instrumento
package c07;
import java.util.*;
class InstrumentoDePercusion extends Instrumento {
  // Redefine interface method:
  public void tocar(Nota n) {
    System.out.println("InstrumentoDePercusion.tocar()");
  }
}
public class Dificil {
    public static Instrumento randInstrumento() {
          switch(new Random().nextInt() % 3) {
          case 0 : return new InstrumentoDeViento();
          case 1 : return new InstrumentoDeCuerda();
          default : return new InstrumentoDePercusion();
          }
    }
    public static void main(String[] args) {
          Instrumento[] ai = new Instrumento[10];
          for(int i=0; i< 10; i++)  ai[i] = randInstrumento(); Musica.afinar(ai[i]);
          }

    }
}
```

La palabra reservada *abstract*

* Una clase abstracta es aquella que posee al menos un método abstracto, anteponiendo **abstract** antes de la declaración del método.

* Un método abstracto es aquél que está declarado, pero que no posee implementación alguna. Está incompleto.

* Debido a esta carencia, es imposible declarar objetos de una clase abstracta. ¿Para qué sirve entonces una clase abstracta?

* Una clase abstracta sirve para establecer un marco de trabajo que deben cumplir todas las clases que hereden de ella. Las clases que heredan de ella sí deben dotar de una implementación a los métodos abstractos del padre. Si no lo hacen, automáticamente se convierten también en abstractas.

* También es posible definir una clase abstracta sin que posea métodos abstractos. Ello se hace anteponiendo **abstract** a la declaración de la clase. El efecto es el mismo.

* Ej.: Hacer que Instrumento sea abstracta, y ver los mensajes de error de compilación.

Interfaces

* Una interface es parecida a una clase abstracta excepto porque:

- En la interface todas las funciones miembro son implícitamente **abstract** y **public**.
- En la interface sólo pueden aparecer funciones miembro, y no datos miembro (excepto **static final**).
- Una clase no puede extender una interfaz (**extends**), sino que la implementa (**implements**).
- Una clase puede implementar tantas interfaces como quiera, pero sólo puede extender una clase.

* La interface hace que todas las clases que la implementan tengan un comportamiento semejante. Ej.:

```
// Prueba.java
// Ejemplo de uso de las interfaces
import java.io.*;
/**
@author Sergio Gálvez Rojas
*/
interface Comparable {
   boolean mayor(Object o);
   boolean menor(Object o);
   boolean igual(Object o);
}
class Caja implements Comparable {
   int lado;
   Caja(int l) {lado = l;}
   public boolean mayor(Object c) { return lado > ((Caja)c).lado; };
   public boolean menor(Object c) { return lado < ((Caja)c).lado; };
   public boolean igual(Object c) { return lado == ((Caja)c).lado; };
}
class Televisor implements Comparable {
   int pulgadas;
   boolean color;
   Televisor(int p, boolean c) {pulgadas = p; color = c;}
   public boolean mayor(Object t) { return pulgadas > ((Televisor)t).pulgadas; };
   public boolean menor(Object t) { return pulgadas < ((Televisor)t).pulgadas; };
   public boolean igual(Object t) { return pulgadas == ((Televisor)t).pulgadas
                                        && color == ((Televisor)t).color;
                                  }
}
```

Interfaces

```
public class Prueba {
   public static void main(String[] args) {
     Caja c1 = new Caja(5);  Televisor t1 = new Televisor(21, true);
     Caja c2 = new Caja(15); Televisor t2 = new Televisor(21, false);
     Caja c3 = new Caja(25); Televisor t3 = new Televisor(28, true);
     System.out.println("c1 > c2: " + c1.mayor(c2));
     System.out.println("c2 < c3: " + c2.menor(c3));
     System.out.println("c3 == c2: " + c3.igual(c2));
     System.out.println("c3 > c2: " + c3.mayor(c2));
     System.out.println("c1 == c1: " + c1.igual(c1));
     System.out.println("c1 > c1: " + c1.mayor(c1));
     System.out.println("t1 > t2: " + t1.mayor(t2));
     System.out.println("t2 == t1: " + t2.igual(t1));
     System.out.println("t2 > t1: " + t2.mayor(t1));
   }
}
```

* Ej. de implementación de múltiples interfaces:

```
//: Adventure.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Multiple interfaces
import java.util.*;
interface CanFight {
  void fight();
}
interface CanSwim {
  void swim();
}
interface CanFly {
  void fly();
}
class ActionCharacter {
  public void fight() {}
}
class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}
```

Interfaces

```
public class Adventure {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero i = new Hero();
    t(i); // Treat it as a CanFight
    u(i); // Treat it as a CanSwim
    v(i); // Treat it as a CanFly
    w(i); // Treat it as an ActionCharacter
  }
}
```

* Una interfaz puede heredar de otra interfaz, aunque con el único objetivo de incrementar el número de métodos miembro.

* Aunque no pueden crearse objetos de clases abstractas, y mucho menos de interfaces, sí es posible crear manejadores, a los que se puede «enchufar» cualquier objeto de verdad que extienda la clase abstracta, o implemente la interfaz.

Clases internas

* En Java es posible definir una clase dentro de otra. Dicha clase (llamada **clase interna**), tiene completa visibilidad sobre los miembros de su clase contenedora. Ej.:

```
//: Parcel1.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
import java.io.*;
public class Parcel1 {
  private int j = 23;
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    // Aquí accedemos a un miembro private de la clase contenedora.
    String readLabel() { return label + " " + j; }
  }
  // Using inner classes looks just like using any other class, within Parcel1:
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tanzania");
  }
}
```

Clases internas

* Se puede crear una clase interna anónima. Para ello, en el momento en que se crea el objeto se le dota de una nueva implementación. En el siguiente ejemplo se crea un objeto que hereda de una interface, y se le dota de implementación en el mismo momento de su creación.

* Para manejar dicho objeto, se utiliza un manejador de objetos que cumplen la interfaz Contents.

```
//: Parcel6.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// A method that returns an anonymous inner class
import java.io.*;
interface Contents {
  int value();
}
public class Parcel6 {
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
    System.out.println(c.value());
  }
}
```

Polimorfismo y constructores

* Probar el siguiente ejemplo:

```
//: PruebaFigura.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Constructors and polymorphism don't produce what you might expect.
abstract class Figura {
  abstract void draw();
  Figura() {
    System.out.println("Figura() before draw()");
    draw();
    System.out.println("Figura() after draw()");
  }
}
class Circulo extends Figura {
  int radio = 1;
  Circulo(int r) {
    radio = r;
    System.out.println("Circulo.Circulo(), radio = " + radio);
  }
  void draw() {
    System.out.println("Circulo.draw(), radio = " + radio);
  }
}
public class PruebaFigura {
  public static void main(String[] args) {
    new Circulo(5);
  }
}
```

* Esto se debe a que un objeto Círculo se inicializa así:

1.- Se ubica memoria para un Círculo y se pone toda a cero.

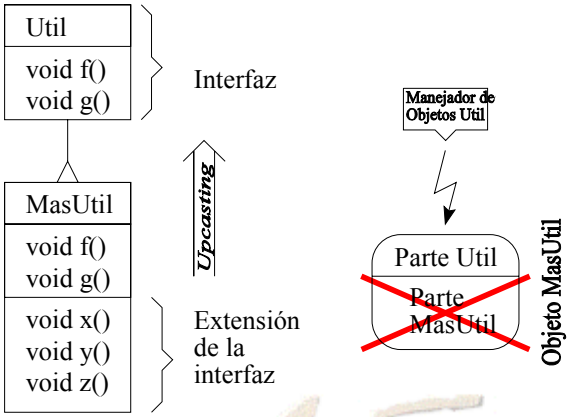
2.- Se llama al constructor de la clase base Figura. Aquí, se llama a **draw()**, que como está reescrito, por vinculación dinámica es el **draw()** de Círculo. ¡Pero ¿qué valor tiene radio en este momento?!

3.- Se inicializan los miembros de Círculo en el mismo orden en que están declarados.

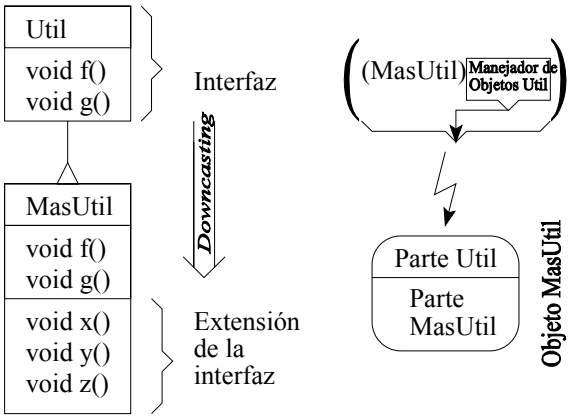
4.- Se llama al constructor de la clase hija.

Downcasting

* Hacer *upcasting* es una operación segura, porque se tiene asegurado que un objeto hijo implementa como mínimo todas las operaciones de uno padre. El único problema es que desde el manejador no se tiene acceso a las extensiones del objeto apuntado.



* Por *downcasting* se entiende especificar que un manejador de una clase padre está apuntando a un objeto de una clase hija. Se hace anteponiendo al nombre del manejador el nombre de la clase hija. Java testea en tiempo de compilación que los *downcasting* sean correctos. Si no lo es se genera la excepción **ClassCastException**.



Downcasting y *RTTI*

* El siguiente ejemplo ilustra las limitaciones del *upcasting*, y los peligros del *downcasting*.

```
//: RTTI.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Downcasting & Run-Time Type Identification (RTTI)
import java.util.*;
class Useful {
  public void f() {}
  public void g() {}
}
class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}
public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
}
```

Capítulo 5
Colecciones

* Cuando se desea tener una gran cantidad de objetos se usan los arrays.

* Si uno viola los límites de un array, se obtiene un **RuntimeException**.

* Cuando el número de elementos a almacenar es desconocido se emplean las colecciones, que son estructuras dinámicas de datos, tales como **Vector, Stack**, **Hashtable**, y **Bitset**. En la versión 1.2, se incrementan las posibilidades.

* Estas *colecciones* almacenan objetos de tipo **Object**. Como, tarde o temprano, cualquier clase hereda de Object, pues son capaces de almacenar cualquier cosa, excepto tipos primitivos, ya que no pertenecen a clase alguna. Esto presenta dos problemas:
- Aunque se pretenda guardar elementos homogéneos, p.ej. *Cuadrados*, dado que se puede guardar cualquier cosa que herede de **Object**, nada impide que, por descuido, se guarden también *Circulos*.
- Como la colección guarda manejadores de **Object**, cuando se extrae un elemento, lo que sale es un **Object**: se ha perdido toda la información referente al tipo. Para recuperarla, es necesario hacer un *cast*.

Vectores

```
//: CatsAndDogs.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Simple collection example (Vector)
import java.util.*;
class Cat {
  private int catNumber;
  Cat(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}
class Dog {
  private int dogNumber;
  Dog(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}
public class CatsAndDogs {
  public static void main(String[] args) {
    Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.elementAt(i)).print();
    // Dog is detected only at run-time
  }
}
```

- **addElement()**. Añade un elemento.
- **elementAt()**. Extrae el elemento de la posición indicada. Lo extrae como **Object**, luego hay que hacerle *casting*,
- **size()**. Devuelve el número de elementos en el **Vector**.

Java no tiene clases parametrizadas

* En Java no existe nada parecido a los *templates* de C++, ni a los genéricos de Ada.
* Si se quiere una colección de algo concreto, hay que construirlo explícitamente. Ej.:

```
//: VectorArdilla.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// A type-conscious Vector
import java.util.*;
class Ardilla {
  private int ArdillaNumber;
  Ardilla(int i) {
    ArdillaNumber = i;
  }
  void print(String msg) {
    if(msg != null) System.out.println(msg);
    System.out.println(
      "Ardilla number " + ArdillaNumber);
  }
}
class TrampaArdilla {
  static void pilladaYa(Ardilla g) {    g.print("Pillada one!");  }
}
class VectorArdilla {
  private Vector v = new Vector();
  public void addElement(Ardilla m) {
    v.addElement(m);
  }
  public Ardilla elementAt(int index) {
    return (Ardilla)v.elementAt(index);
  }
  public int size() { return v.size(); }
  public static void main(String[] args) {
    VectorArdilla Ardillas = new VectorArdilla();
    for(int i = 0; i < 3; i++)
      Ardillas.addElement(new Ardilla(i));
    for(int i = 0; i < Ardillas.size(); i++)
      TrampaArdilla.pilladaYa(Ardillas.elementAt(i));
  }
}
```

Enumerators

* Cada colección tiene una interfaz distinta.

* ¿Qué ocurre si empezamos nuestro proyecto utilizando *Vectores*, y luego decidimos, por cuestiones de eficiencia, utilizar *Listas*? Tendremos que modificar de cabo a rabo nuestro proyecto.

* Para evitar esto, surgen los iteradores, que en Java se llaman **Enumeration**, y que en la versión 1.2 se llaman **Iterator**.

* Un **Enumeration** es un objeto que nos permite movernos por una colección como si de una secuencia cualquiera se tratase y, por tanto, uniformizando el acceso, y haciéndolo independiente de la colección concreta de que se trate.

* Las operaciones principales que se permite hacer con un **Enumeration** son:

- Preparar a la colección para ser recorrida. Para ello, cualquier colección admite el método **elements()**, que devuelve un **Enumeration**.
- Recorrer los elementos del iterador. Para ello, los iteradores admiten el método **nextElement()**, que devuelve el siguiente objeto almacenado en la secuencia.
- Antes de recuperar un nuevo elemento, hay que cerciorarse de que quedan más objetos en la secuencia. Para ello se usa **hasMoreElements().**

Enumerators

```
//: CatsAndDogs2.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Simple collection with Enumeration
import java.util.*;
class Cat2 {
  private int catNumber;
  Cat2(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat number " +catNumber);
  }
}
class Dog2 {
  private int dogNumber;
  Dog2(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog number " +dogNumber);
  }
}
public class CatsAndDogs2 {
  public static void main(String[] args) {
    Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat2(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog2(7));
    Enumeration e = cats.elements();
    while(e.hasMoreElements())
      ((Cat2)e.nextElement()).print();
    // Dog is detected only at run-time
  }
}
```

Hashtables

* Una tabla *hash* permite establecer una colección de pares (**clave, dato**), donde tanto la clave como el dato son objetos).

* La interfaz básica es la siguiente:
- **size()**. Indica cuantos pares hay en la tabla.
- **put(clave, dato)**. Inserta un par en la tabla.
- **get(clave)**. Devuelve el dato asociado a la clave buscada.
- **remove(clave)**. Elimina el par que posee la clave indicada.
- **keys()**. Produce un **Enumeration** con todas las claves.
- **elements()**. Produce un **Enumeration** con todos los datos asociados.

* La clase **Object** tiene un método **hashCode()** que, para cada objeto, devuelve un código único de tipo **int**. Por defecto devuelve la dirección de memoria en que se encuentra el objeto.

* Cuando se producen conflictos en la tabla *hash*, la librería llama a **equals()** para comparar los objetos

* Para usar un objeto como clave en una tabla *hash*, hay que sobrecargar ambos métodos.

Hashtables

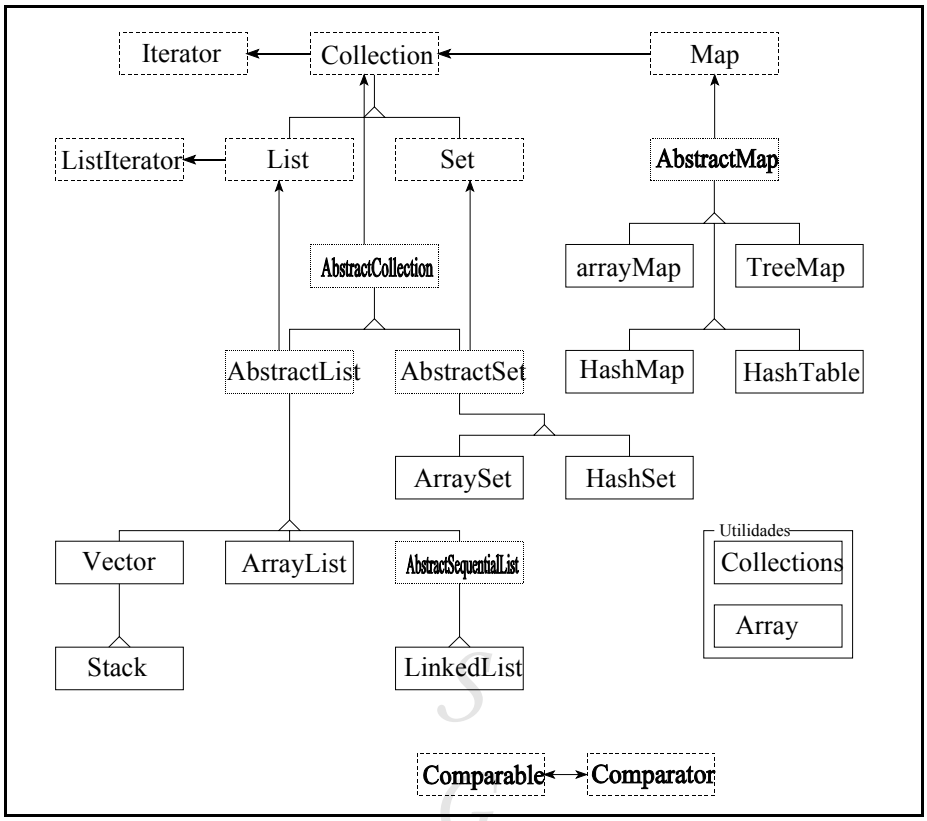
```
//: SpringDetector2.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// If you create a class that's used as a key in a Hashtable,
// you must override hashCode() and equals().
import java.util.*;

class Groundhog2 {
  int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    return (o != null) &&
      (o instanceof Groundhog2) &&
      (ghNumber == ((Groundhog2)o).ghNumber);
  }
}

public class SpringDetector2 {
  public static void main(String[] args) {
    Hashtable ht = new Hashtable();
    for(int i = 0; i < 10; i++)
      ht.put(new Groundhog2(i),new Prediction());
    System.out.println("ht = " + ht + "\n");
    System.out.println(
      "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
    if(ht.containsKey(gh))
      System.out.println((Prediction)ht.get(gh));
  }
}
```

Enumerators

* Ahora que conocemos más colecciones (Vector y tabla *hash*), queda más patente la utilidad homogeneizadora de la clase **Enumerator**.

```
//: Enumerators2.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Revisiting Enumerations
import java.util.*;

class PrintData {
  static void print(Enumeration e) {
    while(e.hasMoreElements())
      System.out.println(
        e.nextElement().toString());
  }
}

class Enumerators2 {
  public static void main(String[] args) {

    Vector v = new Vector();
    for(int i = 0; i < 5; i++)
      v.addElement(new Mouse(i));

    Hashtable h = new Hashtable();
    for(int i = 0; i < 5; i++)
      h.put(new Integer(i), new Hamster(i));

    System.out.println("Vector");
    PrintData.print(v.elements());
    System.out.println("Hashtable");
    PrintData.print(h.elements());
  }
}
```

Colecciones y Java 1.2



* Java versión 1.2 incorpora toda una nueva batería de colecciones, agrupadas en los conceptos de:
- **Collection**. Colección de objetos simples.
- **Map**. Colección de pares de objetos.
* Las clases rayadas son *interfaces*.
* Las clases punteadas son *clases abstractas*.
* Las líneas con flecha indican ***implements***.
* Todas las colecciones pueden producir un **Iterator**, a través del método **iterator()**, que es lo mismo que un **Enumeration**, excepto porque:
- Utiliza nombres de métodos más simples: **hasNext()** en vez de **hasMoreElements()**, **next()** en vez de **nextElement()**, etc.
- Incluye el método **remove()**, que elimina el último elemento producido. Puede llamarse a **remove()** por cada **next()** que se haga.

Test de List

```
//: ListPerformance.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Demonstrates performance differences in Lists
import java.util.*;

public class ListPerformance {
  private static final int REPS = 100;
  private abstract static class Tester {
    String name;
    int size; // Test quantity
    Tester(String name, int size) {
      this.name = name;
      this.size = size;
    }
    abstract void test(List a);
  }
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a) {
        for(int i = 0; i < REPS; i++) {
          for(int j = 0; j < a.size(); j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration", 300) {
      void test(List a) {
        for(int i = 0; i < REPS; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new Tester("insert", 1000) {
      void test(List a) {
        int half = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for(int i = 0; i < size * 10; i++)
          it.add(s);
      }
    },
    new Tester("remove", 5000) {
      void test(List a) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
          it.remove();
        }
      }
    },
  };
  public static void test(List a) {
    // A trick to print out the class name:
    System.out.println("Testing " +
      a.getClass().getName());
    for(int i = 0; i < tests.length; i++) {
      for(int j = 0; j < tests[i].size; j++)
        a.add(Integer.toString(i));
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(a);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void main(String[] args) {
    test(new ArrayList());
    test(new LinkedList());
  }
}
```

* Observar los .class generados por javac.

Capítulo 6
Tratamiento de excepciones

* El tratamiento de errores en lenguajes como C o Modula-2 supone una auténtica pesadilla, en la que hay que testear constantemente en *switches* externos a la lógica del programa.

* Java incorpora palabras reservadas para trabajar con errores y excepciones. Cuando se produce una excepción, el flujo de ejecución salta automáticamente a una zona especial del código. Si esta zona especial no existe, el método actual se cancela y se vuelve al método llamante, buscando en éste la susodicha zona de código especial. Así sucesivamente hasta que la excepción sea tratada.

* Cuando se produce un fallo que impide continuar la ejecución se «eleva» una excepción. Para ello se emplea la palabra reservada **throw**.

* Una excepción se considera un objeto que lleva asociada información sobre el fallo ocurrido. Hay distintos tipos de objetos para identificar distintos tipos de excepciones.
* Hay excepciones que «vienen de fábrica», pero además, el programador puede crear las suyas propias.
* Todos los fallos posibles en ejecución heredan de **Throwable**. Java suministra dos subclases fundamentales que derivan de **Throwable**:
- **Error**: Reservado para fallos internos del sistema: falta de memoria, etc.
- **Exception**: Son los fallos típicos de los programas: división por cero, punteros nulos, etc.

«Elevar» y «atrapar» una excepción

* Cuando se «eleva» o se «lanza» una excepción, lo primero que hay que hacer es crear el objeto *excepción*, con **new**.

* Dado que una excepción es un objeto más, puede contener toda la información que se quiera, puede tener distintos constructores, etc. Ej.:

```
if (t == null)
    throw new NullPointerException("t = null");
```

* Cuando se eleva una excepción, se supone que ésta va a ser atrapada en algún lugar del código. Para ello se emplea el concepto de «región protegida». Una región protegida tiene el formato:

```
try {
    // Código que puede generar excepciones.
} catch (TipoException1 t1) {
    // Controlador de excepciones del tipo 1
} catch (TipoException1 t2) {
    // Controlador de excepciones del tipo 2
} ....
} catch (TipoException1 tn) {
    // Controlador de excepciones del tipo n
} finally {
    // Trozo de código que se ejecutará siempre.
}
```

Captura y «relanzamiento». Palabra reservada *finally*

* Los ***catch*** establecen una especie de *switch* para atrapar las excepciones elevadas en el bloque ***try*.** Cuando en el *try* se eleva una excepción, se busca el primer catch que tenga como argumento una excepción del tipo elevado; en ese momento se ejecutará su cuerpo, y la excepción se da por gestionada.

* Es posible que el bloque **catch** no sea capaz de gestionar la excepción. En tal caso, puede volver a elevarla. Cuando un bloque **catch** eleva una excepción, el método en el que se encuentra finaliza de forma anómala. Se indica que un método puede acabar de forma anómala, mediante la palabra **throws** en la cabecera:

```
void f() throws tooBig, tooSmall, divZero {
....}
```

* Para capturar excepciones también se emplea el concepto de herencia. De esta forma es posible capturar cualquier tipo de excepción mediante un *catch* que captura un objeto de tipo **Exception**, del que heredan todos los demás.

* **Exception** posee:

- Constructor que admite un **String**: descripción del problema.
- **getMessage()**. Toma el **String** dado en el constructor.
- **toString()**. Convierte la excepción en una cadena.
- **printStackTrace()**. Visualiza la pila de llamadas a métodos.

* El trozo de código situado en la parte **finally** se ejecutará siempre, tanto se haya producido alguna excepción como si no. Este bloque tiene sentido para cuando hay que cerrar ficheros que se han abierto en la región protegida, o cosas parecidas.

Ejemplo

* Ejemplo de bloque o región protegida.

```
//: ExceptionMethods.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Demonstrating the Exception Methods
package c09;

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("Here's my Exception");
    } catch(Exception e) {
      System.out.println("Caught Exception");
      System.out.println("e.getMessage(): " + e.getMessage());
      System.out.println("e.toString(): " + e.toString());
      System.out.println("e.printStackTrace():");
      e.printStackTrace();
    }
  }
}
```

* Ejemplo de creación de excepciones propias.

```
//: Inheriting.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Inheriting your own exceptions
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}
public class Inheriting {
  public static void f() throws MyException {
    System.out.println(
      "Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println(
      "Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
  }
}
```

Detalles sobre excepciones

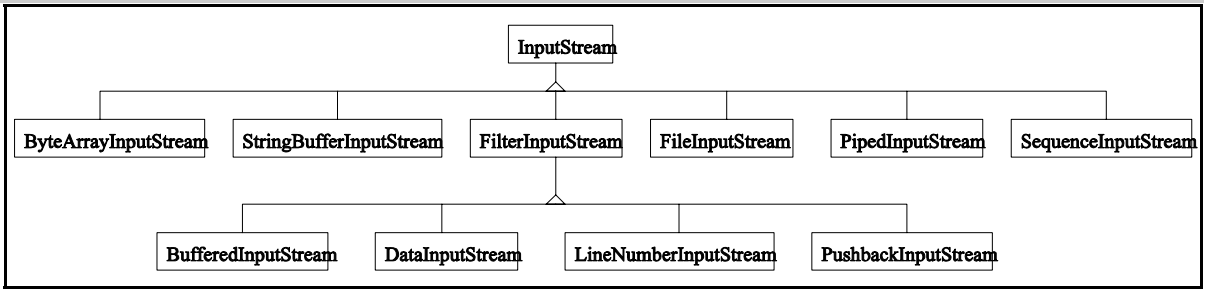
* De forma implícita, y sin necesidad de especificarlo, toda función puede lanzar **RuntimeException**.
* Es posible construir una excepción que no posea código alguno:

```
class ExcepcionSimple extends Exception {
}
```

* Cuando se reescribe un método sólo se pueden lanzar las excepciones que se describían en el método base.
* **finally** se ejecuta aunque la excepción no sea capturada.
* Hay que tener cuidado con la pérdida de excepciones.

```
//: LostMessage.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// How an exception can be lost
class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}
class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}
public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args)
      throws Exception {
    LostMessage lm = new LostMessage();
    try {
      lm.f();
    } finally {
      lm.dispose();
    }
  }
}
```

Capítulo 7
E/S con Java



* Dado que la entrada/salida depende de la platafroma en que Java se ejecute, el lenguaje Java no posee operaciones de E/S. En su lugar posee una compleja librería.

* Aunque se utiliza el concepto de canal (*stream*), éste no tiene nada ver con su homónimo de C++.

* Todas las clases que implementan canales en los que se puede escribir heredan de **OutputStream**, y todas aquellas de las que se puede leer heredan de **InputStream**.

* **InputStream** agrupa a todas las clases que representan canales de los que se puede leer. Por cada tipo de fuente de datos se tiene una subclase de **InputStream**. Las fuentes pueden ser:
- Un array de *bytes*.
- Un **String**.
- Un fichero en disco.
- Otro *stream*.
- Una secuencia de *streams*.
- Una conexión con Internet.

Estratificación

* Para cada una de las fuentes anteriores se tienen las siguientes subclases:

- **ByteArrayInputStream**: La entrada es un buffer de memoria.
- **StringBufferInputStream**: El fuente es un **String**.
- **FileInputstream**: El fuente es un fichero.
- **PipedInputStream**: La entrada es un *pipeline* de salida.
- **SequenceInputStream**: Convierte una secuencia de **InputStream** en uno solo.
- **FilterInputStream**: De aquí heredan todos los estratos.

* Java suministra una serie de clases que proporcionan distintas funcionalidades a los canales de entrada y de salida. Se pueden colocar en capas para suministrar más de una funcionalidad a la vez: es la estratificación. P.ej.:

- **DataInputStream**: Prepara al canal de entrada para recoger valores primitivos (int, char, long, etc.).
- **BufferedInputStream**: Evita que en cada operación de lectura se efectúe una lectura física. Crea un *buffer* interno.
- **LineNumberInputStream**: Lleva la cuenta del número de líneas que han sido leídas.
- **PushbackInputStream**: Permite reinsertar caracteres leídos en el canal de entrada.

Ejemplo: fichero entrada con buffer

```
try {
    // 1. Buffered input file
    DataInputStream in =
      new DataInputStream(
        new BufferedInputStream(
          new FileInputStream(new String("MiFichero.txt"))));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
} catch(FileNotFoundException e) {
    System.out.println(
      "File Not Found:" + args[0]);
} catch(IOException e) {
    System.out.println("IO Exception");
}
```

* Este ejemplo abre un fichero del disco, establece un *buffer* para él, y a continuación lo prepara para leer datos primitivos en formato estándar.

* A continuación lee su contenido línea por línea, y el fichero entero lo inserta en un objeto de tipo **String**.

* Por último cierra el fichero.

* Dado que el fichero que se abre puede no existir, y que pueden producirse excepciones de E/S, es necesario agrupar todo el proceso en una región protegida, en la que se capturen dichas excepciones.

Ejemplo: entrada desde una cadena

```
try
    // 2. Input from memory
    StringBufferInputStream in2 =
       new StringBufferInputStream(s2);
    int c;
    while((c = in2.read()) != -1)
      System.out.print((char)c);
} catch(IOException e) {
    System.out.println("IO Exception");
}
```

* En este caso, vamos leyendo carácter a carácter desde una cadena de caracteres. **read()** devuelve cada *byte* como un entero, luego hay que convertirlo a **char**.

* La propia fuente establece un buffer, luego no es necesario acoplarle ningún estrato más.

* De nuevo controlamos los posibles errores de E/S que aparezcan.

Ejemplo: cadena entrada con buffer

```
// 3. Formatted memory input
try {
  DataInputStream in3 =
    new DataInputStream(
      new StringBufferInputStream(s2));
  while(true)
    System.out.print((char)in3.readByte());
} catch(EOFException e) {
  System.out.println("End of stream encountered");
}
} catch(IOException e) {
  System.out.println("IO Exception");
}
```

* A la lectura desde una cadena se le ha añadido la posibilidad de leer datos primitivos, aunque en este caso, la lectura se ha efectuado igualmente en base a *bytes* posteriormente convertidos a caracteres.
* Con este método, incluso el <EOF> se considera un carácter válido, luego no hay forma de detectar el final del fichero. Aquí no se ha detectado, sino que se ha delegado en capturar la excepción **IOException.** Un método mejor es emplear **available()** que devuelve el número de *bytes* que quedan por leer.

```
// 3. Formatted memory input
try {
  DataInputStream in =
    new DataInputStream(
     new BufferedInputStream(
      new FileInputStream("TestEof.java")));
  while(in.available() != 0)
    System.out.print((char)in.readByte());
} catch (IOException e) {
  System.err.println("IOException");
}
```

Ejemplo: la entrada estándar

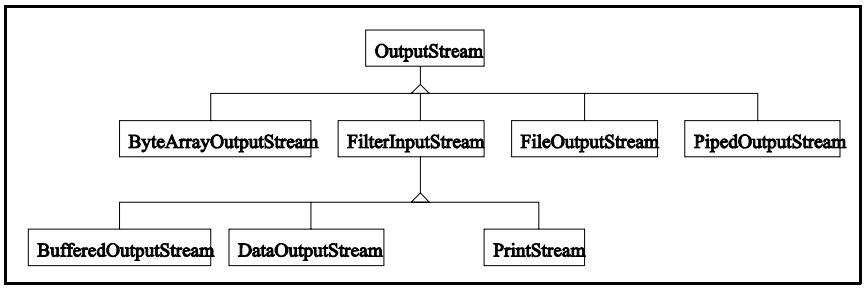
```
import java.io.*;
public class Prueba {
   public static void main(String[] args) {
        try {
           System.out.println(System.in.getClass().getName());
           DataInputStream d = new DataInputStream(System.in);
           System.out.println("Inserta una cadena, un carácter, " +
                              "un entero, otro carácter y un float.");
           String s = d.readLine();
           char c = d.readLine().charAt(0);
           int i = Integer.parseInt(d.readLine(), 10);
           char c2 = (char)(d.readByte());
           float f = (new Float(d.readLine())).floatValue();
           System.out.println(s + "\n" + i + "\n" + c + "\n" +
                              c2 + "\n" + f + "\n");

        } catch (IOException e) {
           System.out.println("Vaya, hay un error.");
        };
   }
}
```

* Este ejemplo ilustra como insertar cosas desde la entrada estándar. La entrada estándar queda representada por un objeto de tipo **BufferedInputStream**, cuyo nombre es ***System.in***.

* System.in permite leer sólo entrada binaria y caracteres. Sin embargo todas las entradas desde el teclado son secuencias de caracteres, consecuencia de las teclas pulsadas.

* Para leer correctamente es necesario leer una tira de caracteres, (para lo cualno hay ningún problema) y convertirla al formato que convenga.
* Es especialmente interesante la conversión a **int**, y el método que permite convertir a cualquier otro tipo (excepto **char**).

Salida de datos



* **OutputStream** agrupa a todas las clases que representan canales en los que se puede escribir. Por cada tipo de destino de datos se tiene una subclase de **OutputStream**. Los destinos pueden ser:
- Un array de *bytes*.
- Un fichero en disco.
- Otro *stream*.

* Para ello se tienen las siguientes subclases:
- **ByteArrayOutputStream**: Escribe en un buffer de memoria.
- **FileOutputStream**: Escribe en un fichero.
- **PipedOutputStream**: Enlaza con un **PipedInputStream**.
- **FilterOutputStream**: De aquí heredan todos los estratos de salida.

* **FilterOutputStream** suministra los siguientes estratos:
- **DataOutputStream**: Prepara al canal de salida para recoger valores primitivos (int, char, long, etc.).
- **PrintStream**: Para producir salida inteligible. **DataOutputStream** se encarga del almacenamiento, y **PrintStream** se encarga de la visualización.
- **BufferedOutputStream**: Para que todas las operaciones utilicen un *buffer* de memoria.

Ejemplo escritura en fichero de texto

```
// 4. Line numbering & file output
try {
  LineNumberInputStream li =
    new LineNumberInputStream(
      new StringBufferInputStream(s2));
  DataInputStream in4 =
    new DataInputStream(li);
  PrintStream out1 =
    new PrintStream(
      new BufferedOutputStream(
        new FileOutputStream(
          "IODemo.out")));
  while((s = in4.readLine()) != null )
    out1.println(
      "Line " + li.getLineNumber() + s);
  out1.close(); // finalize() not reliable!
} catch(EOFException e) {
  System.out.println(
    "End of stream encountered");
}
```

* Este ejemplo hace uso de **LineNumberInputStream.** Para sacar el número de línea que se está leyendo, es necesario tener un manejador a un objeto de este tipo, lo que se consigue construyendo el **DataInputStream** en dos pasos.

* El número de línea se obtiene con **getLineNumber()**.

* En este caso, lo que se lee por un lado se escribe en un fichero «IODemo.out», que se modifica para que tenga un *buffer*, y luego se convierte en **PrintStream**, lo que hace que lo que se va escribiendo lo haga en formato textual. De esta forma, el fichero IODemo.out es de tipo texto. En otro caso sería binario.

Ejemplo: guardar y recuperar datos

```
// 5. Storing & recovering data
try {
  DataOutputStream out2 =
    new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
  out2.writeBytes(
    "Here's the value of pi: \n");
  out2.writeDouble(3.14159);
  out2.close();
  DataInputStream in5 =
    new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
  System.out.println(in5.readLine());
  System.out.println(in5.readDouble());
} catch(EOFException e) {
  System.out.println(
    "End of stream encountered");
}
```

* Aquí se utiliza un **DataOutputStream** para guardar datos en formato binario, y **DataInputStream** para recuperarlos.

* Para que todo esto funcione bien, cuando se lee del fichero, hay que saber exactamente qué tipos de datos se han guardado, y en qué orden.

Ficheros de acceso directo

* La clase **RandomAccessFile** se utiliza para acceder a ficheros compuestos de registros de tamaño conocido, y posicionarse en cualquiera de ellos directamente, mediante **seek()**, para leerlo o modificarlo.
* **RandomAccessFile** viene a ser una mezcla entre **DataInputStream** y **DataOutputStream**, junto con los siguientes métodos fundamentales:
- **getFilePointer()**. Para saber en qué posición del fichero nos encontramos.
- **seek()**. Para irse a un punto concreto del fichero. El punto se especifica en bytes, motivo por el cual puede ser necesario saber la longitud de los distintos tipos de datos que se guardan.
- **length()**. Para saber el tamaño del fichero.
- Los constructores necesitan saber si el fichero se abre para sólo lectura "**r**", o en lectura/escritura "**rw**".

```
try {
    // 6. Reading/writing random access files
    RandomAccessFile rf =
      new RandomAccessFile("rtest.dat", "rw");
    for(int i = 0; i < 10; i++)
      rf.writeDouble(i*1.414);
    rf.close();

    rf =
      new RandomAccessFile("rtest.dat", "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();

    rf =
      new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
      System.out.println(
        "Value " + i + ": " +
        rf.readDouble());
    rf.close(); // Aquí vienen los catch....
```

La clase File

* La clase **File** sirve para guardar nombres de ficheros y de directorios. Admite el método **list()** que obtiene una lista con los nombres de los ficheros de dicho directorio.
* A **list()** se le puede pasar un parámetro que será un objeto que implementa la interfaz **FilenameFilter**. Todo objeto del tipo **FilenameFilter** debe tener un método llamado **accept()**, al que list() llama para cada uno de los nombres de fichero del directorio indicado. Si **accept()** devuelve **true** el fichero se admite; se rechaza en caso contrario.

```
// Displays directory listing
import java.io.*;
public class DirList {
  public static void main(final String[] args) {
    try {
      File path = new File(".");
      String[] list;
      if(args.length == 0)
          list = path.list();
      else
          list = path.list(new FilenameFilter() {
                                  String afn = args[0];
                                  public boolean accept(File dir, String name) {
                                    return name.indexOf(afn) != -1;
                                  }
                               }
                           );
      for(int i = 0; i < list.length; i++)
          System.out.println(list[i]);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

* Nótese que no se permite el uso de caracteres comodines.

StreamTokenizer

```
import java.io.*;
import java.util.*;
public class Token {
  public static void main(String[] args) {
          StreamTokenizer st = new
StreamTokenizer(System.in);
    st.ordinaryChar('.');
    st.ordinaryChar('-');
    try {
      while(st.nextToken() !=
StreamTokenizer.TT_EOF) {
        String s;
        double d;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            System.out.println("Fin de linea");
            break;
          case StreamTokenizer.TT_NUMBER:
            d = st.nval;
            System.out.println("Un número: " + d);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            System.out.println("Una palabra: " + s);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
            System.out.println("Un carácter: " + s);
        }
      }
    } catch(IOException e) {
      System.out.println(
        "st.nextToken() unsuccessful");
    }
  }
}
```

* En este ejemplo se utiliza una librería que posee Java para efectuar la partición de un texto de entrada en bloques: números y palabras fundamentalmente.
* Posee los siguientes elementos:
- **nextToken()**. Carga el siguiente token.
- **ttype**. Indica el tipo del token actual: TT_EOL, TT_NUMBER, TT_WORD, TT_EOF, o un carácter si no es ninguno de ellos.
- **nval**: Si el token es un número, posee el valor de dicho número como double.
- **sval**: Si el token es una palabra, contiene a dicha palabra.
* Es una alternativa para la lectura desde el teclado.

E/S con Java 1.1
Entrada

* Java versión 1.1 ha sustituído casi todas las clases que hemos visto por otras nuevas más eficientes.

* Para la entrada son:
Raíz principal:

Antes: *InputStream* Ahora: **Reader**

Descendientes directos:

Antes:	Ahora:
ByteArrayInputStream	**CharArrayReader**
StringBufferInputStream	**StringReader**
FileInputStream	**FileReader**
PipedInputStream	**PipedReader**
SequenceInputStream	**SequenceInputStream**
FilterInputStream	**FilterReader**

Estratos que heredan directamente de **Reader**:

Antes:	Ahora:
DataInputStream	**DataInputStream**
BufferedInputStream	**BufferedReader**
LineNumberInputStream	**LineNumberReader**
PushBackInputStream	**PushBackReader** *(es el único que hereda de FilterReader)*

* Se puede convertir un **InputStream** en un **Reader** mediante el estrato **InputStreamReader**.

E/S con Java 1.1
Salida

* Para la salida son:
Raíz principal:

Antes: *OutputStream* Ahora: **Writer**

Descendientes directos:

Antes:	Ahora:
ByteArrayOutputStream	**CharArrayWriter**
StringBufferOutputStream, que no existe	**StringWriter**
FileOutputStream	**FileWriter**
PipedOutputStream	**PipedWriter**
FilterOutputStream	**FilterWriter**

Estratos (ahora descienden directamente de **Writer**):

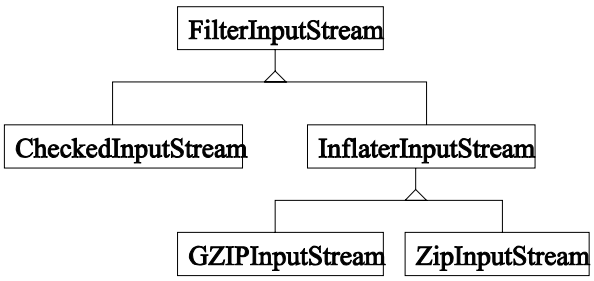
Antes:	Ahora:
BufferedInputStream	**BufferedWriter**
PrintStream	**PrintWriter**
DataOutputStream	**DataOutputStream**

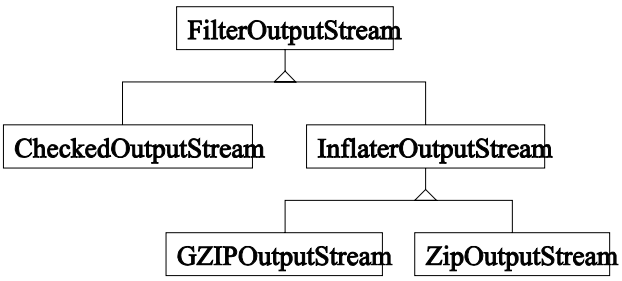
* El tratamiento sigue siendo igual que antes.

* Se puede convertir un **OutputStream** en un **Writer** mediante el estrato **OutputStreamWriter**.

Compresión

* Java 1.1 posee clases que permiten trabajar con ficheros comprimidos. Se encuentran en el paquete java.util.zip.





* La utilización es muy simple. El sistema se encarga de todo.

```
//: GZIPcompress.java Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Uses Java 1.1 GZIP compression to compress a file.
import java.io.*;
import java.util.zip.*;
public class GZIPcompress {
  public static void main(String[] args) {
    try {
      BufferedReader in =
        new BufferedReader(
          new FileReader(args[0]));
      BufferedOutputStream out =
        new BufferedOutputStream(
          new GZIPOutputStream(
            new FileOutputStream("test.gz")));
      System.out.println("Writing file");
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.close();
      System.out.println("Reading file");
      BufferedReader in2 =
        new BufferedReader(
          new InputStreamReader(
            new GZIPInputStream(
              new FileInputStream("test.gz"))));
      String s;
      while((s = in2.readLine()) != null)
        System.out.println(s);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

* El proceso de lectura escritura es el mismo de siempre.

Ficheros .jar

* Los ficheros .jar (Java Archive) son especialmente útiles cuando se trabaja con Internet. Permiten insertar clases, imágenes, sonidos, etc. en un sólo fichero, y tratarlo como un único bloque a través de la red.
*Además los .jar comprimen los ficheros que contiene, disminuyendo así el tráfico por la red.
* Formato: jar [opciones] destino [descripción] ficheros

Opción	Descripción
c	Crea un fichero nuevo o vacío.
t	Lista el contenido del fichero.
x	Extrae todos los ficheros.
x fichero	Extrae el fichero indicado.
f	Si no se pone, se usa la E/S estándar como fuente/destino, en lugar de los ficheros indicados
m	El fichero indicado contiene la descripción.
v	Modo *verbose.*
O	Sólo guarda los ficheros. No los comprime. Necesario cuando se usan en local.
M	No crea ninguna descripción automáticamente.

* Como ficheros, pueden incluirse directorios enteros.
* Si desea usarse en local un fichero .jar, debe usarse la opción *O*, e indicarse dicho fichero explícitamente en la variable de entorno CLASSPATH, o bien colocarlo como extensión del entorno Java en el directorio *jdk1.3\jre\lib\ext*.

Persistencia

* La persistencia es el proceso que permite guardar cualquier objeto en disco para ser recuperado posteriormente.

* En Java, la persistencia se hace vía ***Serialización***, que consiste en convertir el objeto en una secuencia de bytes, tal que pueda ser posteriormente guardada y recuperada desde un fichero.

* Para guardar un objeto es necesario que implemente la interfaz **Serializable**.

* Los pasos para guardar un objeto *serializable* son:
- Crear un objeto **OutputStream**.
- Ponerle un estrato del tipo **ObjectOutputStream**.
- Escribir los objetos mediante **writeObject().**

* Los pasos para recuperarlo son:
- Crear un objeto **InputStream**.
- Ponerle un estrato del tipo **ObjectInputStream**.
- Leer los objetos mediante **readObject()**, lo que devuelve un manejador a un **Object**, que deberá ser convertido *(downcasted)* a su verdadero tipo.

* Lo interesante a la hora de guardar un objeto es que se guarden no sólo sus datos primitivos, sino también todos los objetos que contiene, y así sucesivamente. Para conseguirlo, todos los componentes deben ser también serializables.

* Cuando se recupera un objeto, el sistema Java debe ser capaz de encontrar el .class a que corresponde cada objeto guardado.

Ejemplo de serialización

* En este ejemplo se guarda una objeto enorme formado a su vez por otros objetos. Probar el ejemplo, y probarlo de nuevo haciendo que Data no sea serializable.

```
import java.io.*;
class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}
public class Worm implements Serializable {
  // Generate a random int value:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
        new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0) next = new Worm(i, (char)(x + 1));
  }
  Worm() {
    System.out.println("Default constructor");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++) s += d[i].toString();
    s += ")";
    if(next != null) s += next.toString();
    return s;
  }
  public static void main(String[] args) {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    try {
      ObjectOutputStream out =
          new ObjectOutputStream(
            new FileOutputStream("worm.out"));
      out.writeObject("Worm storage");
      out.writeObject(w);
      out.close(); // Also flushes output
      ObjectInputStream in =
          new ObjectInputStream(
            new FileInputStream("worm.out"));
      String s = (String)in.readObject();
      Worm w2 = (Worm)in.readObject();
      System.out.println(s + ", w2 = " + w2);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

La interfaz Externalizable

* Para obtener un control total sobre qué elementos se guardan y cuáles no, se tiene la interfaz *Externalizable*, que extiende a la *Serializable*.

* La interfaz Externalizable añade dos métodos que deben ser reescritos por el implementador: son **writeExternal()** y **readExternal()**, que son automáticamente llamados por el sistema durante el proceso de serialización o deserialización respectivamente.

* En este caso, a diferencia de cuando se implementa **Serializable**, antes de cargar un objeto, se llama a su constructor, y posteriormente se llama automáticamente a **readExternal().**

* En **writeExternal()** y **readExternal()** hay que leer y excribir explícitamente todos los objetos y datos primitivos que se quiera.

* **writeExternal()** recibe como parámetro un objeto **ObjectOutput** (que hereda de **DataOutput**), en el que hay que escribir los objetos componentes que se quieran mediante **writeObject()**. Si lo que se quiere guardar son primitivas, emplearemos los métodos propios de **DataOutput**: **writeInt(), writeLong(), writeDouble()**, etc.

Ejemplo de Externalizable

```
//: Blip3.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Reconstructing an externalizable object
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
   // s & i initialized only in non-default constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip3.writeExternal");
    // You must do this:
    out.writeObject(s); out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException
  {
    System.out.println("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject(); i =in.readInt();
  }
  public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3.toString());
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Blip3.out"));
      System.out.println("Saving object:");
      o.writeObject(b3);
      o.close();
      // Now get it back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Blip3.out"));
      System.out.println("Recovering b3:");
      b3 = (Blip3)in.readObject();
      System.out.println(b3.toString());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

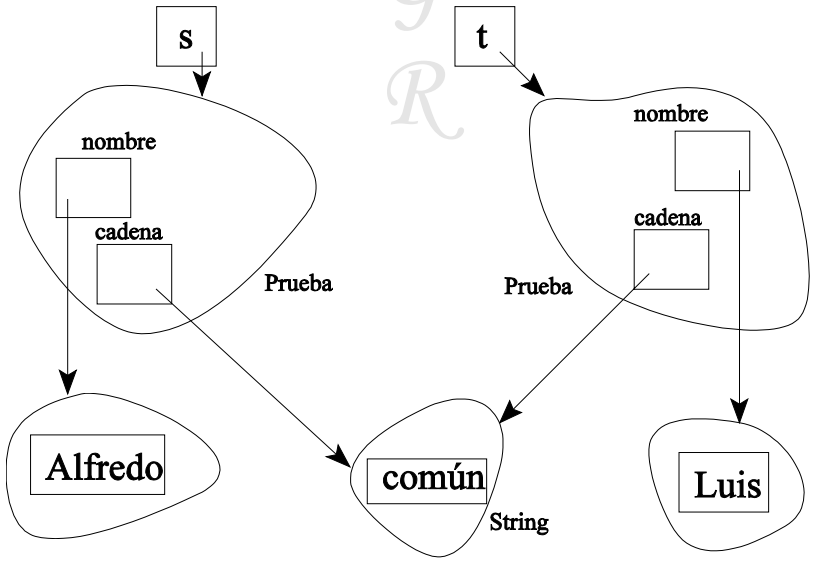
* Las líneas marcadas indican los lugares donde se hacen las llamadas para serializar y deserializar objetos y componentes.

Más sobre serialización

* Si no queremos implementar **Externalizable** para ahorrarnos escribir los métodos **writeObject()** y **readObject()**, pero aún así, hay campos que no queremos que se serialicen, podemos hacerlo precediendo la declaración de dichos campos del modificador **transient**.

* **transient** viene a indicar que el campo no forma parte del estado del objeto y que, por tanto, no hace falta guardarlo.

* Comprobar que ocurre cuando dos objetos que tienen un campo objeto en común se serializan, y luego se recuperan. ¿El sistema duplica el campo común o no?



* Para visualizar el valor de un manejador podemos sacar su valor hashCode(), que por defecto es la dirección de memoria en que se encuentra el objeto.

Ejemplo

```
import java.io.*;

public class Prueba implements Serializable {
   String nombre;
   String cadena;
   public Prueba() {}
   public Prueba(String a, String b)
        { nombre = a; cadena = b; }
   public String toString() {
        if (this == null) return "nada";
        else
            return nombre + nombre.hashCode() +
                    cadena + cadena.hashCode();
   }
   public static void main(String[] args) {
     String c = "común";
     Prueba s = new Prueba("Alfredo", c),
            t = new Prueba("Luis", c);
     try {
          ObjectOutputStream out =
             new ObjectOutputStream(
                  new FileOutputStream("out.dat"));
          System.out.println(s);
          System.out.println(t);
          out.writeObject(s);
          out.writeObject(t);
          out.close();
          s = t = null;
          System.out.println(s);
          System.out.println(t);
          ObjectInputStream in =
             new ObjectInputStream(
                  new FileInputStream("out.dat"));
          s = (Prueba)in.readObject();
          t = (Prueba)in.readObject();
            System.out.println(s);
            System.out.println(t);
          in.close();
     } catch (Exception e) {
          e.printStackTrace();
     }
   }
}
```

Capítulo 8
Metaclases

* Las metaclases permiten saber el tipo exacto de un objeto cuando lo único que se tiene es un manejador a su clase base.

* Java suministra un conjunto de metaclases con este objetivo: **Class, Field, Method, Constructor,** etc. Asímismo, la clase **Object** posee el método **getClass(),** que permite obtener un objeto de tipo **Class** con toda la información sobre el verdadero objeto apuntado.

* El objeto **.class** correspondiente a una clase cualquiera **Xxx** puede obtenerse sencillamente mediante la expresión **Xxx.class**.

* La verdad es que, durante la ejecución, por cada clase que hay cargada, Java mantiene un objeto oculto de tipo **Class** que describe a dicha clase.

* Ese objeto de tipo **Class** se utiliza en tiempo de ejecución para chequear las conversiones de tipo efectuadas por el programador.

* El operador **instanceof** nos dice si un objeto es instancia de una clase determinada o no.

Ejemplo

```
//: ToyTest.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Testing class Class
interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
  // Comment out the following default
  // constructor to see
  // NoSuchMethodError from (*1*)
  Toy() {}
  Toy(int i) {}
}
class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}
public class ToyTest {
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
    } catch(ClassNotFoundException e) {}
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requires default constructor:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {}
      catch(IllegalAccessException e) {}
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println(
      "Class name: " + cc.getName() +
      " is interface? [" +
      cc.isInterface() + "]");
  }
}
```

Capítulo 9
Copias

* Se denomina *alias* al efecto según el cual un mismo objeto puede ser gestionado a través de más de un manejador.

* Los *alias* tienen la particularidad de que producen efectos laterales en los objetos apuntados.

* Cuando se pasa un manejador como argumento a un método, en dicho método se está creando un *alias*, ya que el manejador se pasa por valor, y por ende, el objeto apuntado se pasa por referencia.

```
//: Alias2.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Method calls implicitly alias their
// arguments.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 handle) {
    handle.i++;
    System.out.println("handle: " + handle.j);
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
}
```

Copias locales

* Cuando pasan cosas así, hay que decidir si puede ser peligroso, si el que usa la clase modificadora no está al tanto de dicho mecanismo.

* Si se necesita modificar un argumento dentro de un método, pero no se quiere propagar dicha modificación al exterior, el método puede hacer una copia local para su gestión interna, y no tocar, así, el objeto de entrada.

* Para entender esto bien, hay que tener muy claros los siguientes 6 puntos:

- El alias es automático por el mero hecho de pasar un parámetro a un método.
- No existen objetos locales, sino manejadores locales.
- Hay ámbito para los manejadores, pero no para los objetos.
- La vida de un objeto no es problema nuestro. Para eso está el recolector de basura.
- No hay forma de evitar que un método modifique su parámetro de entrada.
- Los objetos no se pasan como parámetros, sino manejadores a objetos.

Copias locales

* Para hacer una copia local de un objeto, se hace uso del método **clone()** que está definido en **Object** como **private**, y que debemos reescribir como **public** en nuestra clase.

```
// Method calls explicitly clones their argument.

public class NoAlias2 implements Cloneable {
  int i;
  NoAlias2(int ii) { i = ii; }
  public Object clone()
          throws CloneNotSupportedException {
    Object interno;
    interno = super.clone();
    return interno;
  }
  static void f(NoAlias2 handle) {
    try {
          NoAlias2 copiaHandle = (NoAlias2)handle.clone();
    copiaHandle.i++;
    System.out.println("copiaHandle: " + copiaHandle.i);
    } catch (Exception e) {
          System.out.println("Mal está el asunto.");
    }
  }
  public static void main(String[] args) {
    NoAlias2 x = new NoAlias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
}
```

* La llamada a super.clone() es fundamental, y se encarga de:
- Ubicar la memoria necesaria.
- Hacer una copia bit a bit del objeto actual.

Ejemplo

* De lo anterior se deduce que no se hacen copias recursivas de un objeto. O sea, si un objeto A contiene a otro B, y se hace A' = A.clone(), se obtendrá que A y A' contienen manejadores distintos al mismo B: B posee aliases.
* Para que no ocurra esto, el método clone() de la clase de A debe:
  - Efectuar una llamada a super.clone().
  - Hacer un clone() para cada uno de los manejadores que contenga.

```
//: DeepCopy.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Cloning a composed object
class DepthReading implements Cloneable {
  private double depth;
  public DepthReading(double depth) {
    this.depth = depth;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return o;
  }
}
class TemperatureReading implements Cloneable
{
  private long time;
  private double temperature;
  public TemperatureReading(double temperature) {
    time = System.currentTimeMillis();
    this.temperature = temperature;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return o;
  }
}
class OceanReading implements Cloneable {
  private DepthReading depth;
  private TemperatureReading temperature;
  public OceanReading(double tdata, double ddata){
    temperature = new TemperatureReading(tdata);
    depth = new DepthReading(ddata);
  }
  public Object clone() {
    OceanReading o = null;
    try {
      o = (OceanReading)super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
    // Must clone handles:
    o.depth = (DepthReading)o.depth.clone();
    o.temperature =
      (TemperatureReading)o.temperature.clone();
    return o; // Upcasts back to Object
  }
}

public class DeepCopy {
  public static void main(String[] args) {
    OceanReading reading =
      new OceanReading(33.9, 100.5);
    // Now clone it:
    OceanReading r =
      (OceanReading)reading.clone();
  }
}
```

Capítulo 10
Concurrencia

* Java permite la multitarea, o sea, la existencia simultánea de distintos objetos ejecutando alguno de sus métodos a la vez.

A cada una de estas tareas, Java las llama *thread*.

* Para ello existen dos opciones fundamentales:

- Heredar de la clase ***Thread***.

Tiene la ventaja de que facilita la gestión del *thread*.

- Implementar la interfaz ***Runnable***.

Tiene la ventaja de que permite heredar de una clase propia, a la vez que se crea un *thread*.

* Cualquiera que sea la opción elegida, el objeto debe poseer un método llamado **run()** que comenzará a ejecutarse cuando se arranque el funcionamiento del objeto, mediante el mensaje **start()**.

```
// ThreadEscritor.java
//Thread que escribe un mensaje continuamente

public class ThreadEscritor extends Thread {
   String interno;
   public ThreadEscritor (String msg) {
      super();
      interno = msg + "\t";
   }
   public void run() {
      while (true) {
         System.out.println(interno);
         yield();
      }
   }

   public static void main(String[] args) {
         // Esto crea los threads, llamando a sus
constructores.
                  ThreadEscritor uno = new
ThreadEscritor("Uno");
                  ThreadEscritor dos = new
ThreadEscritor("Dos");
         // Esto empieza a ejecutar ambos threads.
         uno.start();
         dos.start();
        // Nos dormismos durante un rato, y mientras
         // los otros se siguen ejecutando.
         try {
            Thread.currentThread().sleep(10000);
         } catch (InterruptedException e) {
         } finally {
            System.exit(0);
         }
      }
   }
```

Concurrencia

* También es posible arrancar a ejecutar un objeto en el mismo momento de su creación. Para ello basta con llamar a **start()** justo cuando creamos el objeto.

```
// ThreadEscritor.java
//Thread que escribe un mensaje continuamente

public class ThreadEscritor extends Thread {
   String interno;
   public ThreadEscritor (String msg) {
      super();
      interno = msg + "\t";
      start();
   }
   public void run() {
      while (true) {
         System.out.println(interno);
         yield();
      }
   }

   public static void main(String[] args) {
         // Esto crea los threads, llamando a sus constructores.
                  ThreadEscritor uno = new ThreadEscritor("Uno");
                  ThreadEscritor dos = new ThreadEscritor("Dos");
         // Nos dormimos durante un rato, y mientras
         // los otros se siguen ejecutando.
         try {
            Thread.currentThread().sleep(10000);
         } catch (InterruptedException e) {
         } finally {
            System.exit(0);
         }
      }
}
```

* Nótese la utilización de System.exit(0) para acabar la ejecución de la aplicación Java completa. Si no se pone, acabará el programa main() (que se considera otro *thread*), pero como aún hay dos *threads* más ejecutándose, el sistema no da por acabada la ejecución del programa.

* Véase también la función estática **currentThread()** que devuelve un manejador al *thread* actual.

* Un *thread* se clasifica como daemon con **setDaemon(Boolean n)**. Así, el programa anterior acabaría si se quita el System.exit(0), y se hace t_x.setDaemon(true); antes de hacer t_x.start();

Concurrencia

* Que la clase que implementa el método **run()** tenga que heredar de **Thread** es un inconveniente, porque Java no permite herencia múltiple.
* Para solucionar el problema aparece la interfaz **Runnable**. **Runnable** obliga únicamente a que exista el método **run()**.
* Un objeto que herede de **Runnable** se puede comenzar a ejecutar pasándolo como parámetro a un constructor de **Thread**. P.ej., el ejemplo anterior quedaría:

```
// ThreadEscritor.java
//Thread que escribe un mensaje continuamente
public class ThreadEscritor implements Runnable
{
   String interno;
   public ThreadEscritor (String msg) {
      interno = msg + "\t";
   }
   public void run() {
      while (true) {
         System.out.println(interno);
         Thread.yield();
      }
   }

   public static void main(String[] args) {
         // Esto crea los objetos, llamando a sus
constructores.
                  ThreadEscritor uno = new
ThreadEscritor("Uno");
                  ThreadEscritor dos = new
ThreadEscritor("Dos");
         // Esto crea los threads en base a los objetos
         // que implementan Runnable.
         Thread t1 = new Thread(uno);
         Thread t2 = new Thread(dos);
         // Arrancamos los threads creados.
         t1.start();
         t2.start();
         // Nos dormimos durante un rato, y mientras
         // los otros se siguen ejecutando.
         try {
            Thread.currentThread().sleep(10000);
         } catch (InterruptedException e) {
         } finally {
            System.exit(0);
         }
      }
   }
```

* Nótese que ahora que ThreadEscritor no hereda de **Thread,** no puede llamar directamente a **yield(),** sino a través de de la propia clase (**yield()** es **static**).

Concurrencia

* Un programa Java acaba cuando acaban todos sus *threads* o cuando los únicos *threads* que quedan están clasificados como *daemons*, o sea, despachadores de servicios.

* En el siguiente ejemplo vemos la creación de un número indefinido de objetos que se convierten en *threads* y de ahí se comienza su ejecución. Aunque no guardamos ninguna referencia directa a esos objetos, el planificador de tareas de Java sí lo hace, por lo que no son recogidos como basura hasta que acaban su ejecución.

```
// Mogollon.java
// Esto crea 10 threads funcionando a la vez.

public class Mogollon implements Runnable {
   private long tiempoInicial = 0;
   private String interno;
   private int ejecuciones = 0;
   public Mogollon(String msg) {
      interno = msg;
   }
   public void run(){
      if (tiempoInicial == 0) tiempoInicial = System.currentTimeMillis();
      while (tiempoInicial + 5000 > System.currentTimeMillis()) {
         ejecuciones += 1;
         System.out.println(interno + " se ha ejecutado " +
                        ejecuciones + " veces.");
         Thread.yield();
      }
   }

   public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
         (new Thread(new Mogollon("Thread " + i + ": "))).start();
      }
      while (Thread.activeCount() >1) {}
      System.out.println("Esto es todo.");
   }
}
```

Concurrencia

* Un *thread* puede tener asociado un nombre, lo cual facilita las labores de depuración. Para ello, el nombre se le puede asignar en el mismo momento en que se crea el **Thread** o bien con **setName()**. Un *thread* puede saber su nombre con **getName()**.
* Los *threads* también pueden tener una prioridad, que se les asigna con **setPriority()** y que se puede saber con **getPriority()**.

```
// Mogollon.java
// Esto crea 10 threads funcionando a la vez.

public class Mogollon implements Runnable {
   final static int NUMERO_DE_THREADS = 10;
   private long tiempoInicial = 0;
   private String interno;
   private int numero;
   private static int[] ejecuciones = new int[NUMERO_DE_THREADS];
   public Mogollon(String msg, int i) {
      interno = msg;
      numero = i;
   }
   public void run(){
      if (tiempoInicial == 0) tiempoInicial = System.currentTimeMillis();
      while (tiempoInicial + 15000 > System.currentTimeMillis()) {
         ejecuciones[numero] += 1;
         System.out.println(interno + " se ha ejecutado " +
                        ejecuciones[numero] + " veces.");
         Thread.yield();
      }
   }
   public static void main(String[] args) {
      for (int i = 0; i < NUMERO_DE_THREADS; i++) {
         Thread t = new Thread(new Mogollon("Thread " + i + ": ", i));
         t.setPriority((i%3 == 0)? Thread.MAX_PRIORITY : Thread.MIN_PRIORITY);
         t.start();
      }
      while (Thread.activeCount() > 1) {}
      for (int i = 0; i < NUMERO_DE_THREADS; i++) {
         System.out.print("["+i+":"+ejecuciones[i]+"]");
      }
      System.out.println("Esto es todo.");
   }
}
```

La palabra *synchronized*

* Cuando hay muchos *threads* funcionando, a menudo comparten objetos, o incluso zonas de memoria. Suele ser necesario hacer que el acceso a esos objetos se haga de forma atómica.

* Para ello, cada objeto posee (de forma automática) una región crítica o **monitor**. Cuando se llama a un método *synchronized* de un objeto , el objeto se bloquea, y ya no acepta más mensajes *synchronized* por parte de nadie.

* Este mismo efecto lo produce *synchronized* cuando es aplicado a un método estático. En estas circunstancias, sólo uno de los objetos podrá entrar a ejecutar el método estático sincronizado.

- Probar a ejecutar el siguiente ejemplo.
- Quitarle el **synchronized.**
- Dejarle el **synchronized** y quitarle el **static**.

La palabra *synchronized*

```
// Mogollon.java
// Esto crea 10 threads funcionando a la vez sincronizadamente.
public class Mogollon implements Runnable {
   final static int NUMERO_DE_THREADS = 10;
   private static int[] ejecuciones = new int[NUMERO_DE_THREADS];
   private long tiempoInicial = 0;
   private String interno;
   private int numero;
   public Mogollon(String msg, int i) {
      interno = msg;
      numero = i;
   }
   synchronized public void run(){
         if (tiempoInicial == 0) tiempoInicial = System.currentTimeMillis();
         while (tiempoInicial + 5000 > System.currentTimeMillis()) {
            miRun(interno, numero);
         }
   }
    synchronized static int miRun(String s, int n) {
      for (int i = 0; i < 100; i++) {
         ejecuciones[n] += 1;
         System.out.println(s + " se ha ejecutado " +
                           ejecuciones[n] + " veces.");
      }
      return n;
   }
   public static void totales(){
      for (int i = 0; i < NUMERO_DE_THREADS; i++) {
         System.out.print("["+i+":"+ejecuciones[i]+"]");
      }
   }
   public static void main(String[] args) {
      for (int i = 0; i < NUMERO_DE_THREADS; i++) {
         Thread t = new Thread(new Mogollon("Thread " + i + ": ", i));
         t.start();
      }
      while (Thread.activeCount() > 1) {}
      totales();
      System.out.println("Esto es todo.");
      while(true){}
   }
}
```

Estados de un *thread*

* Un *thread* puede estar en cuatro estados:
- Nuevo: El *thread* ha sido creado, pero aún no se ha llamado a **start()**.
- Ejecutable: El *thread* está en disposición de ser ejecutado, o incluso se está ejecutando.
- Muerto: El thread ha finalizado su método **run()**.
- Bloqueado: Hay alguna circunstancia que evita que el *thread* esté en disposición de ejecutarse. Un *thread* puede estar bloqueado porque:
    - Ha ejecutado un **sleep(nº de milisegundos)**. El *thread* está dormido. Se despertará una vez transcurrido dicho tiempo.
    - Ha recibido el mensaje **suspend()**. Se quedará bloqueado hasta que reciba el mensaje **resume()**.
    - Ha ejecutado un **wait()**. Está a la espera de recibir un **notify()** o un **notifyAll()**.
    - Está esperando a completar alguna operación de E/S.
    - Está en espera de entrar en un método *synchronized* en ejecución por parte de otro objeto.

* Los pares **suspend()-resume()** y **wait()-notify()** varían en que, cuando son ejecutados en un método sincronizado, el primero no libera al monitor mientras está bloqueado, mientras que el segundo sí. El par **wait()-notify()** sólo puede aparecer en métodos sincronizados.

Ejemplo final

```
// Orden.java
// Esto ordena un array de 1000 enteros a través de threads.

import java.util.*;
public class Orden implements Runnable {
   final static int NUMERO_DE_ENTEROS = 100;
   static int[] a = new int[NUMERO_DE_ENTEROS];
   int extremoInicio, extremoFinal;
   byte enEspera;
   Orden padre;
   Orden() {}
   Orden(int i, int f, Orden p){
      extremoInicio = i;
      extremoFinal = f;
      padre = p;
   }

   synchronized public void meDuermo(byte n) throws InterruptedException {
      enEspera = n;
      wait();
   }

   synchronized public void despierta() {
      enEspera--;
      if (enEspera == 0)
         notify();
   }

   synchronized public void run() {
      /*System.out.println("En el run");*/
      if (extremoFinal - extremoInicio > 0) {
         int medio = a[(extremoFinal+extremoInicio)/2];
         int i = extremoInicio, j = extremoFinal;
         while (i <= j) {
            if (a[i] < medio) i++;
            else {
               while (a[j] > medio) j--;
               if (i <= j) {
                  int aux = a[i];
                  a[i] = a[j];
                  a[j] = aux;
                  i++; j--;
               }
            }
         }
```

Ejemplo final

```
            Thread o1 = new Thread(new Orden(extremoInicio, j, this));
            Thread o2 = new Thread(new Orden(i, extremoFinal, this));
            o1.start();
            o2.start();
            try {
               meDuermo((byte)2);
            } catch (InterruptedException e) {}
         }
         padre.despierta();
         /*System.out.println("Saliendo del run");*/
      }
      public static void main (String[] args) {
         // Rellena el array con números aleatorios.
         a[0] = Math.abs((new Random()).nextInt())% 93;
         for (int i = 1; i < NUMERO_DE_ENTEROS; i++) {
            a[i] = a[i - 1] * 53 % 93;
            System.out.print(a[i]+",");
         }
             Thread o = new Thread(new Orden(0, NUMERO_DE_ENTEROS-1, new
Orden()));
         o.start();
         // Esto no se debe hacer porque consume recursos.
         while (o.isAlive()) {}
         for (int i = 0; i < NUMERO_DE_ENTEROS; i++)
            System.out.print(a[i]+",");

         while (true) {}
      }
}
```

Capítulo 11
Prog. Windows con Java

* Programar bajo Windows implica utilizar una nueva disciplina de programación: la programación orientada a eventos.

* Un programa está formado por un conjunto de componentes, sobre cada uno de los cuales se pueden efectuar ciertas acciones.

* Estos componentes son **botones, ventanas, menús, paneles para visualización de cosas, imágenes,** etc.

* Al igual que en la vida real, cada uno de estos componentes acepta un conjunto de operaciones: ser pulsado, ser arrastrado, ser escrito, aceptar una tecla, etc. Cuando el usuario efectúa una de estas operaciones el sistema Windows se da cuenta, y la recoge, creando un evento que debe ser tratado por el componente correspondiente.

* Windows envía el evento al componente, y éste lo trata como sea: grabando un fichero, emitiendo un mensaje, copiando algo al portapapeles, etc.

* Los componentes se distribuyen de forma jerárquica, dependiendo siempre, en última instancia, de una ventana en la que se integra.

* Se dice que un componente ***tiene el foco*** cuando es el componente actualmente seleccionado por el usuario.

Applets y aplicaciones independientes

* Java permite la creación de aplicaciones Windows independientes, y de aplicaciones para uso en Internet, a través de un navegador: lo que se llaman *applets*.

* Un *applet* tiene las siguientes restricciones:
- Necesita de un navegador para ser ejecutado, y de una página .html que lo incluya.
- No tiene acceso al disco duro del ordenador cliente.
- No tiene acceso a ningún ordenador de Internet, más que al servidor del que proviene.
- No puede contener menúes (sí con Java Swing).

* Todo *applet* debe heredar de una clase denominada **Applet**; además debe poseer una función **paint()** en la que se dibuje la información necesaria.

* **paint()** recibe como parámetro un objeto de tipo **Graphics** que es el lugar que el navegador reserva para realizar el dibujo.

* Un objeto de tipo **Graphics** posee operaciones para dibujar distintas figuras, incluído texto con **drawString()**.

El primer *applet*

```
// Primer.java

import java.applet.Applet;
import java.awt.Graphics;

public class Primer extends Applet {
   public void paint(Graphics g){
        g.drawString("¡El primer applet ya.!", 20, 25);
   }
}
```

* El fichero .html que lo utiliza es:

```
<html>
<head>
<title>El primer applet </title>
</head>
<body>
<hr>
<applet code=Primer.class width=300 height=80>
Applets no soportados.
</applet>
<hr>
</body>
</html>
```

La etiqueta *applet*

* La etiqueta APPLET contiene tres elementos, que son:
code: Nombre de la clase a ejecutar.
width: Anchuro reservada para el *applet*.
height: Altura reservada para el applet.
y puede contener:
archive: Nombre del fichero .jar que contiene a la clase a ejecutar. El navegador se traerá al fichero .jar entero.
codebase: Indica la URL en la que podemos encontrar el fichero .class que contiene el *applet* a ejecutar.
alt: Texto que visualizarán los navegadores que no tienen Java, pero que saben interpretar la etiqueta *applet*.
name: Nombre que puede darse al *applet*. Este nombre es el que hay que mandar al método **getApplet()** para saber si dicho *applet* está en ejecución o no.
align: Tiene el mismo sentido que en la etiqueta <IMG>.
vspace: Espacio en pixels que el navegador dejará por encima y por debajo del *applet*.
hspace: Espacio en pixels que el navegador dejará a ambos lados del *applet*.

* Entre <APPLET> y >/APPLET> pueden aparecer cuantas etiquetas PARAM se quiera, constituidas por dos elementos:
name: Nombre del parámetro.
value: Valor de dicho parámetro.

Fuentes y Colores

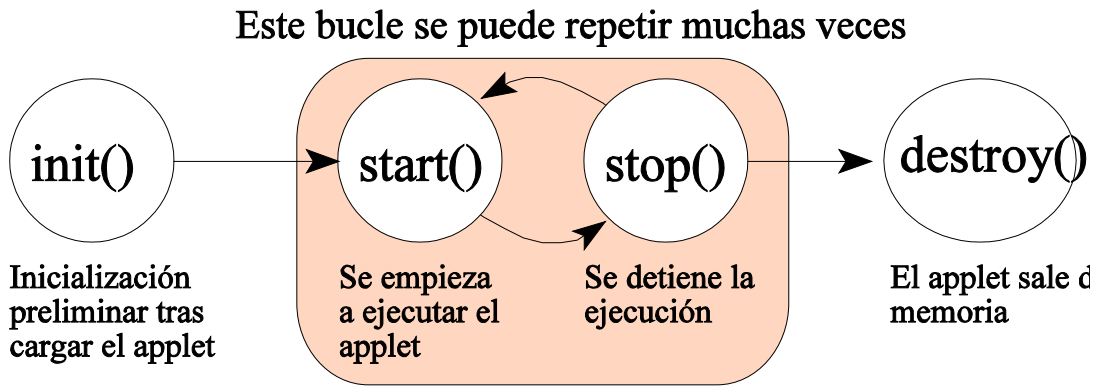
* Para ilustrar las posibilidades de esto, en cuanto a fuentes y colores, se tiene el siguiente ejemplo:

```
// Primer.java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

public class Primer extends Applet {
   public void paint(Graphics g){
   // Rectángulo amarillo.
   g.setColor(Color.yellow);
   g.fillRect(50, 25, 600, 300);
   // αvalo rojo.
   g.setColor(Color.red);
   g.fillOval(50, 25, 600, 300);
   // Círculo azul.
   g.setColor(Color.red);
   g.fillOval(200, 25, 300, 300);
   // Letras en blanco.
   g.setColor(Color.white);
   g.setFont(new Font("Helvetica", Font.BOLD, 25));
   g.drawString("Viva la feria y la playa.", 180, 175);
   }
}
```

Ciclo de vida de un *applet*

* Un *applet* cuando es llamado por el navegador, sufre la siguiente secuencia de mensajes:

1.- Se carga la clase que se haya indicado en la etiqueta **code**.
2.- Se llama al método **init()**, que se encarga de realizar las operaciones que sólo se vayan a realizar una vez: cargar imágenes por Internet, etc.
3.- Se llama al método **start()** para indicar al *applet* que comience su ejecución. **start()** será invocado cada vez que se quiera que se reinicie la ejecución. Normalmente **start()** se encarga de lanzar uno o más *threads*.
4.- El navegador irá invocando **paint()** y otros métodos similares para actualizar la información en pantalla, mientras el *applet* esté activo.
5.- Si el usuario se va a otra página, el navegador llamará a **stop()** para parar el *applet*. Si **start()** lanza *threads*, **stop()** deberá pararlos.
6.- A pesar de que el usuario cambie de página y se mueva por Internet, el navegador mantiene al *applet* en memoria, por lo que los puntos 3, 4 y 5 pueden repetirse muchas veces.
7.- Cuando el navegador ya no necesita más el código del *applet*, llama a **destroy()** y lo quita de memoria.



Paso de parámetros en un *applet*

* Como hemos visto, el fichero .html pasa los parámetros a través de la etiqueta <PARAM>.

* Los parámetros se recogen desde el *applet* a través de **getParameter()**, que admite como parámetro una cadena con el nombre de parámetro a recuperar.

* **getParameter()** recoge siempre una cadena, por lo que si el parámetro es numérico habrá que transformarlo convenientemente.

* La clase Applet incorpora el método **getParameterInfo()** que devuelve un array con información sobre cada uno de los parámteros. Sobre cada parámtero se devuelve su nombre, su tipo, y su descripción.

```
<html>
<head>
<title>El primer applet </title>
</head>
<body>
<hr>
<applet code=Primer.class width=800 height=400>
<param name=UNO value="Un texto de ejemplo.">
<param name=TRES value=102.56>
<param name=DOS value=100>
Applets no soportados.
</applet>
<hr>
</body>
</html>
```

Recuperación de parámetros desde un *applet*

* El programa Java correspondiente es:

```
// Primer.java
import java.applet.Applet;
import java.awt.Graphics;
public class Primer extends Applet {
   private static final String PARAM_UNO  = "UNO";
   private static final String PARAM_DOS  = "DOS";
   private static final String PARAM_TRES = "TRES";
   private static String uno;
   private static int dos;
   private static float tres;
   public String[][] getParameterInfo() {
        String[][] retorno = {
           { PARAM_UNO, "Texto", "Parámetro de prueba tipo textual." },
           { PARAM_DOS, "Entero", "Parámetro de prueba tipo entero." },
           { PARAM_TRES, "Float", "Parámetro de prueba tipo float." }       };
        return retorno;
   }
   public void init() {
        // Inicializa la clase padre.
        super.init();
        // Captura de parámetros.
        uno = getParameter(PARAM_UNO);
        if (uno == null) uno = new String("Valor por defecto");
        try {
           dos = Integer.parseInt(getParameter(PARAM_DOS));
        } catch (NumberFormatException e) {
           dos = 0;
        }
        try {
           tres = (Double.valueOf(getParameter(PARAM_TRES))).floatValue();
        } catch (NumberFormatException e) {
           tres = (float)0.0;
        }
   }
   public void paint(Graphics g){
   g.drawString("Estos son los valores de los parámetros:", 180, 175);
   g.drawString(uno + " " + dos + " " + tres, 180, 200);
   }
}
```

Comunicación con el navegador

* Es de especial importancia en el ámbito de los *applets* el método **getAppletContext()**, que devuelve un objeto que implementa la interfaz **AppletContext**. Este objeto es construído por el propio navegador, y posee, entre otros, los siguientes métodos:

- **Applet getApplet(String)**. Mira a ver si el *applet* especificado está o no en ejecución.

- **Enumeration getApplets()**. Devuelve un iterador con todos los *applets* en ejecucuión.

- **AudioClip getAudioClip(URL)**. Obtiene un fichero de sonido. Véase la clase **AudioClip**.

- **Image getImage()**. Obtiene una imagen. Véase la clase **Image**.

- **void showDocument(URL)**. Indica al navegador que visualice la página especificada.

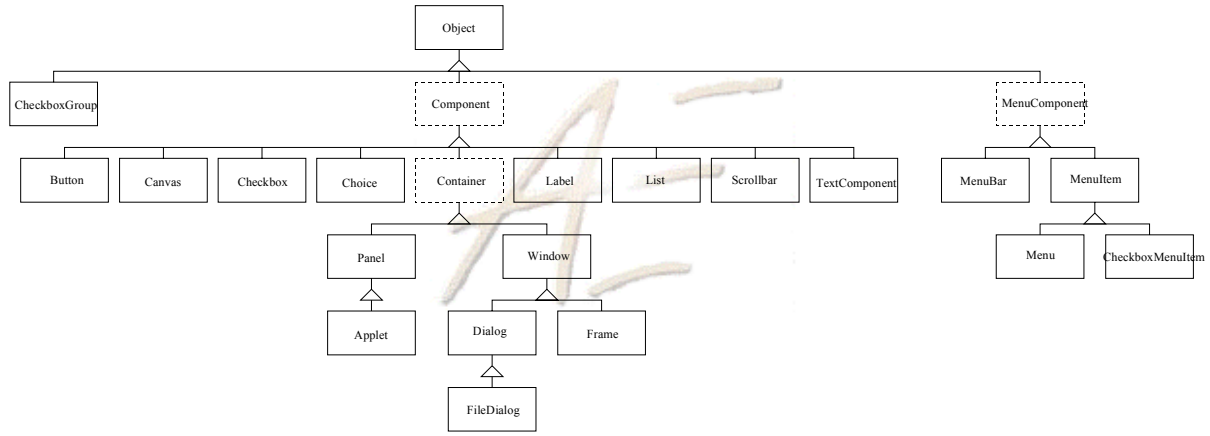
- **void showStatus(String)**. Saca un mensaje por la barra de estado del navegador.

Ventanas

* Más interesante, si cabe, que los *applets* son las aplicaciones independientes, ya que no tienen las restricciones de los *applets*.

* La librería AWT (Abstract Windows Toolkit) suministra todos los componentes necesarios para aplicaciones Windows. Esto componentes han sido extendidos en la librería Swing de la JFC (Java Fundation Classes).

* La jerarquía básica de AWT es la siguiente:



* Toda aplicación Windows suele comenzar en una clase que hereda de **Frame** (o bien activa un **Frame**), a la que se van añadiendo los componentes que queremos que integren nuestra aplicación.

Primer ejemplo de ventana

```
// Primer.java

import java.awt.*;
public class Primer extends Frame {
   Button b1 = new Button("Aceptar"),
          b2 = new Button("Cancelar");
   public Primer() {
        setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
        add(b1);
        add(b2);
   }

   public static void main(String[] args) {
   Primer f = new Primer();
   f.resize(200, 300);
   f.show();
   }
}
```

* La ventana (Frame) posee dos botones que se le añaden con **add()** en el constructor.

* Antes de añadir ningún componente, hay que decidir cómo queremos que se distribuyan los componentes que se van añadiendo. Esto se consigue con **setLayout()**.

* La aplicación arranca desde un **main()**, al igual que cualquier otra aplicación independiente.

* Tras crear el *Frame*, se le dota de un tamaño, y se visualiza.

* **Nota:** Es imposible cerrar la ventana, y además los botones no hacen nada.

Distribución de componentes: *Layout.*

* La forma en que los componentes se distribuyen por la ventana viene dada por un **distribuidor** (*Layout*).

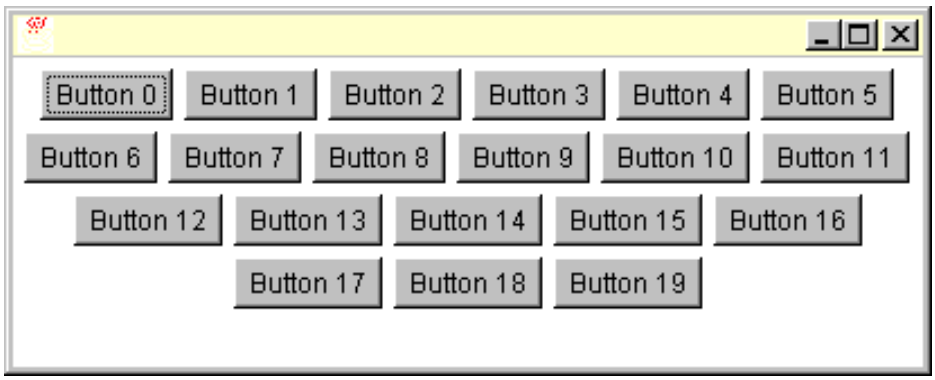
* Los distribuidores se adaptan a las dimensiones de la ventana, de manera que si ésta se redimensiona, el distribuidor redistribuye los componentes de acuerdo a unas reglas.

* Cualquier **Container** tiene operaciones para indicar cual es su distribuidor, y para añadir componentes: **setLayout()** y **add()** respectivamente.

* Los tipos de *Layout* que hay son los siguientes:
- **FlowLayout.**
- **BorderLayout**.
- **GridLayout**.
- **CardLayout**.
- **GridBagLayout.** Este no lo vamos a estudiar.

FlowLayout

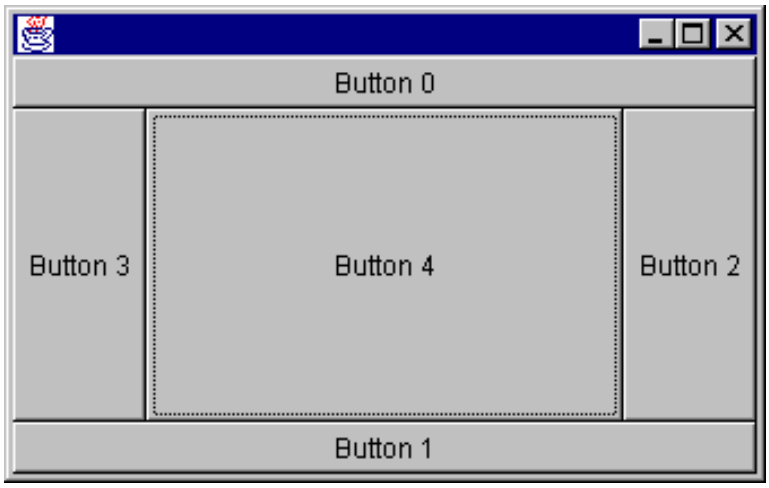
* Este distribuidor va colocando los componentes de izquierda a derecha, y de arriba abajo.




```
//: FlowLayout1.java
// Demonstrating the FlowLayout
import java.awt.*;

public class FlowLayout1 extends Frame {
  public FlowLayout1() {
    setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      add(new Button("Button " + i));
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
              System.out.println("Frame eliminado.");
              System.exit(0);
          }
          return super.handleEvent(e);
    }
  public static void main(String[] ars) {
      FlowLayout1 f = new FlowLayout1();
      f.setSize(300, 400);
      f.show();
  }
}
```

BorderLayout

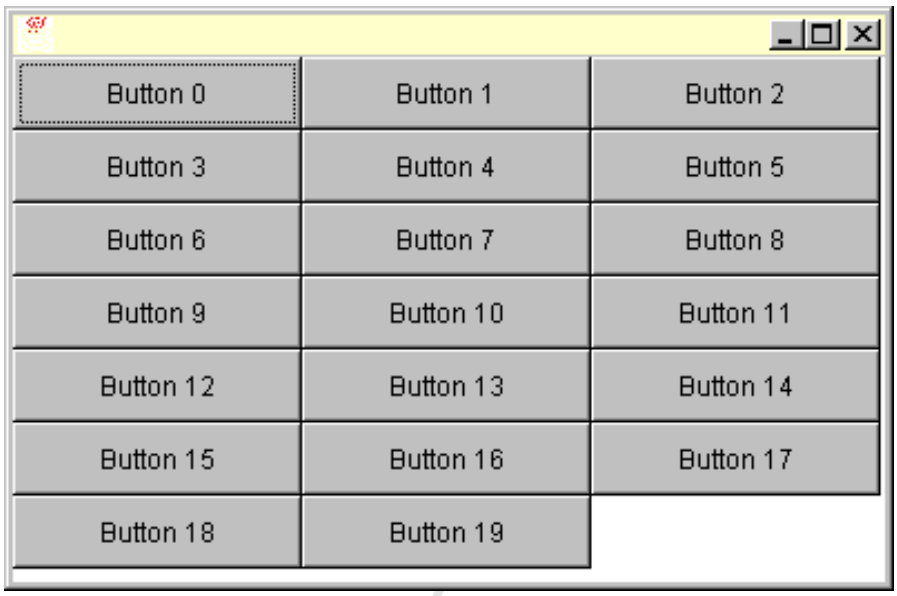
* Este distribuidor coloca las cosas distribuídas a alguna de los lados, o en el centro.




```
//: BorderLayout1.java
// Demonstrating the BorderLayout
import java.awt.*;
public class BorderLayout1 extends Frame {
  public BorderLayout1() {
    int i = 0;
    setLayout(new BorderLayout());
    add("North", new Button("Button " + i++));
    add("South", new Button("Button " + i++));
    add("East", new Button("Button " + i++));
    add("West", new Button("Button " + i++));
    add("Center", new Button("Button " + i++));
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
              System.out.println("Frame eliminado.");
              System.exit(0);
          }
          return super.handleEvent(e);
    }
  public static void main(String[] args) {
      BorderLayout1 f = new BorderLayout1();
      f.setSize(300, 200);
      f.show();
  }
}
```

GridLayout

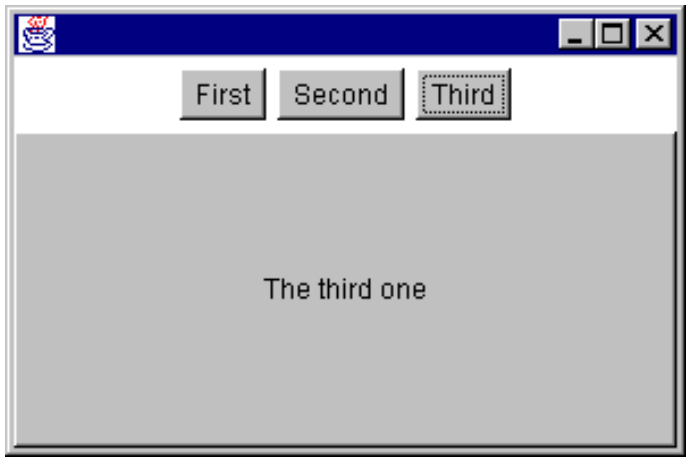
* Este distribuidor va colocando los componentes de izquierda a derecha, y de arriba abajo, según una rejilla cuyo tamaño se especifica en el constructor.




```
//: GridLayout1.java
// Demonstrating the GridLayout
import java.awt.*;

public class GridLayout1 extends Frame {
  public GridLayout1() {
    setLayout(new GridLayout(7, 3));
    for(int i = 0; i < 20; i++)
      add(new Button("Button " + i));
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
              System.out.println("Frame eliminado.");
              System.exit(0);
          }
          return super.handleEvent(e);
    }
  public static void main(String[] ars) {
      GridLayout1 f = new GridLayout1();
      f.setSize(300, 400);
      f.show();
  }
}
```

CardLayout

* Este distribuidor va colocando los componentes unos encima de otros, de manera que pueden ser seleccionados alternativamente mediante botones o cualquier otro sistema similar.




```
//: CardLayout1.java
// Demonstrating the CardLayout
import java.awt.*;
import java.applet.Applet;

class ButtonPanel extends Panel {
  ButtonPanel(String id) {
    setLayout(new BorderLayout());
    add("Center", new Button(id));
  }
}

public class CardLayout1 extends Frame {
  Button
    first = new Button("First"),
    second = new Button("Second"),
    third = new Button("Third");
  Panel cards = new Panel();
  CardLayout cl = new CardLayout();
  public CardLayout1() {
    setLayout(new BorderLayout());
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    p.add(first);
    p.add(second);
    p.add(third);
    add("North", p);
    cards.setLayout(cl);
    cards.add("First card",
      new ButtonPanel("The first one"));
    cards.add("Second card",
      new ButtonPanel("The second one"));
    cards.add("Third card",
      new ButtonPanel("The third one"));
    add("Center", cards);
  }
    public boolean handleEvent(Event e) {
      if (e.id == Event.WINDOW_DESTROY &&
          e.target == this) {
        System.out.println("Frame eliminado.");
        System.exit(0);
      }
      return super.handleEvent(e);
    }
  public boolean action(Event evt, Object arg) {
    if (evt.target.equals(first)) {
      cl.first(cards);
    }
    else if (evt.target.equals(second)) {
      cl.first(cards);
      cl.next(cards);
    }
    else if (evt.target.equals(third)) {
      cl.last(cards);
    }
    else
      return super.action(evt, arg);
    return true;
  }
  public static void main(String[] args) {
      CardLayout1 f = new CardLayout1();
      f.setSize(300, 200);
      f.show();
    }
}
```

Manejo de eventos

* Hay dos formas básicas de manejar un evento, como pueda ser la pulsación de un botón, o el intento por parte del usuario de cerrar la ventana.

* La primera de ellas es horrorosa, pero hay que aprenderla porque es la que poseen muchos navegadores aún no adaptados al nuevo sistema.

* La segunda es mucho más cómoda, y es la que se emplea en la versión 1.1 de Java.

Versión 1.0 de Java

* Crear un enorme **if** en cascada para detectar el componente al que va dirigido el evento.

* Este **if** va dentro del método **action()**, que se llama automáticamente cuando se produce un evento, y al que se le pasan dos parámetros: uno de tipo **Event** que posee toda la información del evento. El segundo lo ignoraremos.

* Un objeto de tipo **Event** tiene un campo **target** que contiene al objeto que ha recibido el evento.

* **action()** debe gestionar el evento y retornar *true*. Si el evento no se puede manejar, haremos que lo genere nuestro padre.

Ejemplo con *action()*

* En el siguiente ejemplo:

```
// Primer.java

import java.awt.*;
public class Primer extends Frame {
   Button b1 = new Button("Aceptar"),
          b2 = new Button("Cancelar");
   public Primer() {
        setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
        add(b1);
        add(b2);
   }
   public boolean action(Event e, Object o) {
        if (e.target.equals(b1))
           System.out.println("Botón uno pulsado.");
        else if (e.target.equals(b2))
           System.out.println("Botón dos pulsado.");
        else if (e.target.equals(this))
           System.out.println("Frame seleccionado.");
        else
           return super.action(e, o);
        return true;
   }

   public static void main(String[] args) {
   Primer f = new Primer();
   f.resize(200, 300);
   f.show();
   }
}
```

no se detecta nada sobre la ventana original.

* Esto se debe a que action() sólo recoge los eventos básicos sobre cada componente, pero no los más complejos.

* Para recoger todo tipo de eventos se emplea **handleEvent()**, que sólo tiene como argumento el evento producido.

* **handleEvent()** se convierte en una cascada de *if* en los que no sólo hay que controlar quién recibe el evento sino qué evento se ha producido.

Ejemplo con *handleEvent()*

* El tipo de evento producido aparece en el campo **id**, y puede ser comparado con una delas numerosas constantes especificadores de tipo de evento que posee **Event**.
* Nada impide tratar los eventos más complejos con **handleEvent()** y los más simples con **action()**. Ej.:

```
// Primer.java

import java.awt.*;
public class Primer extends Frame {
   Button b1 = new Button("Aceptar"),
          b2 = new Button("Cancelar");
   public Primer() {
        setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
        add(b1);
        add(b2);
   }
   public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY && e.target == this) {
           System.out.println("Frame eliminado.");
           System.exit(0);
        }
        return super.handleEvent(e);
   }
   public boolean action(Event e, Object o) {
        if (e.target.equals(b1))
           System.out.println("Botón uno pulsado.");
        else if (e.target.equals(b2))
           System.out.println("Botón dos pulsado.");
        else
           return super.action(e, o);
        return true;
   }

   public static void main(String[] args) {
   Primer f = new Primer();
   f.resize(200, 300);
   f.show();
   }
}
```

TextField

* TextField es un área de texto de una sola línea.
* Hereda de **TextComponent**, que permite poner o tomar texto, seleccionar texto y cogerlo, e indicar si el texto se puede editar o no. Permite operaciones básicas con el portapapeles.
* El constructor permite indicar el texto inicial, así como la longitud de la línea en caracteres.

```
//: TextField1.java
// Using the text field control
import java.awt.*;

public class TextField1 extends Frame {
  Button b1 = new Button("Get Text"),
         b2 = new Button("Set Text");
  TextField t = new TextField("Starting text", 30);
  String s = new String();
  public TextField1() {
    setLayout(new FlowLayout(
                      FlowLayout.CENTER, 20, 20));
    add(b1);
    add(b2);
    add(t);
  }
  public boolean handleEvent(Event e) {
     if (e.id == Event.WINDOW_DESTROY &&
          e.target == this) {
           System.out.println("Frame eliminado.");
           System.exit(0);
     }
     return super.handleEvent(e);
  }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(b1)) {
      System.out.println(t.getText());
      s = t.getSelectedText();
      if(s.length() == 0) s = t.getText();
      t.setEditable(true);
    }
    else if(evt.target.equals(b2)) {
      t.setText("Inserted by Button 2: " + s);
      t.setEditable(false);
    }
    // Let the base class handle it:
    else
      return super.action(evt, arg);
    return true; // We've handled it here
  }
  public static void main(String[] args) {
      TextField1 f = new TextField1();
      f.setSize(300, 200);
      f.show();
  }
}
```

TextArea

* Es igual que TextField, excepto porque permite varias líneas, y además permite añadir, reemplazar e insertar texto por enmedio. A partir de ahora un componente de este puede sustituir a la salida estándar.

```
//: TextArea1.java
// Using the text area control
import java.awt.*;

public class TextArea1 extends Frame {
  Button b1 = new Button("Text Area 1"),
         b2 = new Button("Text Area 2"),
         b3 = new Button("Replace Text"),
         b4 = new Button("Insert Text");
  TextArea t1 = new TextArea("t1", 1, 30),
           t2 = new TextArea("t2", 4, 30);
  public TextArea1() {
    setLayout(new FlowLayout(
                  FlowLayout.CENTER, 20, 20));
    add(b1);
    add(t1);
    add(b2);
    add(t2);
    add(b3);
    add(b4);
  }
    public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY &&
            e.target == this) {
            System.out.println("Frame eliminado.");
            System.exit(0);
        }
        return super.handleEvent(e);
    }
    public boolean action (Event evt, Object arg) {
      if(evt.target.equals(b1))
        t1.setText("Boton 1");
      else if(evt.target.equals(b2)) {
        t2.setText("Boton 2");
        t2.append(":-" + t1.getText());
      }
      else if(evt.target.equals(b3)) {
        String s = "-Replacement-";
        t2.replaceRange(s, 3, 3 + s.length());
      }
      else if(evt.target.equals(b4))
        t2.insert("-Inserted-", 10);
      // Let the base class handle it:
      else
        return super.action(evt, arg);
      return true; // We've handled it here
    }
    public static void main(String[] args) {
        TextArea1 f = new TextArea1();
        f.setSize(300, 200);
        f.show();
    }
}
```

* **TextArea** posee un constructor con el que no sólo se indica el contenido y dimensiones iniciales de este componente, sino con el que también es posible indicar si se quieren barrars de desplazamiento, y a qué lados.

Label

* Permite colocar etiquetas, por ejemplo para identificar un **TextField**.
* En el momento en que se crea la etiqueta se le da un tamaño. Sepuede cambiar el título del la etiqueta con **setText()**, y recuperarlo con **getText()**.
* También se puede hacer que el texto de la etiqueta esté alineado a izquierda, derecha o centro.

```
//: Label1.java
// Using labels
import java.awt.*;

public class Label1 extends Frame {
  TextField t1 = new TextField("t1", 10);
  Label labl1 = new Label("TextField t1");
  Label labl2 = new Label("                    ");
  Label labl3 = new Label("                    ",
    Label.RIGHT);
  Button b1 = new Button("Test 1");
  Button b2 = new Button("Test 2");
  public Label1() {
    setLayout(new FlowLayout(
                  FlowLayout.CENTER, 20, 20));
    add(labl1); add(t1);
    add(b1); add(labl2);
    add(b2); add(labl3);
  }
    public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY &&
            e.target == this) {
            System.out.println("Frame eliminado.");
            System.exit(0);
        }
            return super.handleEvent(e);
    }
    public boolean action (Event evt, Object arg) {
      if(evt.target.equals(b1))
        labl2.setText("Text set into Label");
      else if(evt.target.equals(b2)) {
        if(labl3.getText().trim().length() == 0)
            labl3.setText("labl3");
        if(labl3.getAlignment() == Label.LEFT)
            labl3.setAlignment(Label.CENTER);
        else if(labl3.getAlignment()==Label.CENTER)
            labl3.setAlignment(Label.RIGHT);
        else if(labl3.getAlignment() == Label.RIGHT)
            labl3.setAlignment(Label.LEFT);
      }
      else
        return super.action(evt, arg);
      return true;
    }
    public static void main(String[] args) {
      Label1 f = new Label1();
      f.setSize(300, 200);
      f.show();
    }
}
```

Checkbox

* Checkbox sirve para hacer un selección binaria.

* Cada vez que un Checkbox se activa o desactiva, ocurre un evento como si de un botón se tratase.

* Se puede saber el estado de un Checkbox mediante **getState()**.

```
//: Checkbox1.java
// Using check boxes
import java.awt.*;

public class Checkbox1 extends Frame {
  TextArea t = new TextArea(6, 20);
  Checkbox cb1 = new Checkbox("Check Box 1");
  Checkbox cb2 = new Checkbox("Check Box 2");
  Checkbox cb3 = new Checkbox("Check Box 3");
  public Checkbox1() {
    setLayout(new FlowLayout(
                    FlowLayout.CENTER, 20, 20));
    add(t); add(cb1); add(cb2); add(cb3);
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
              System.out.println("Frame eliminado.");
              System.exit(0);
          }
          return super.handleEvent(e);
    }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(cb1))
      trace("1", cb1.getState());
    else if(evt.target.equals(cb2))
      trace("2", cb2.getState());
    else if(evt.target.equals(cb3))
      trace("3", cb3.getState());
    else
      return super.action(evt, arg);
    return true;
  }
  void trace(String b, boolean state) {
    if(state)
      t.appendText("Box " + b + " Set\n");
    else
      t.appendText("Box " + b + " Cleared\n");
  }
  public static void main(String[] args) {
      Checkbox1 f = new Checkbox1();
      f.setSize(300, 200);
      f.show();
  }
}
```

CheckboxGroup

* **CheckboxGroup** implementa los llamados *radio buttons* que son una secuencia de **Checkbox** en los que sólo puede haber activo uno cada vez.

* Antes de visualizar un **CheckboxGroup** debe haber uno y sólo un **Checkbox** activado.

* Para incluir un **Checkbox** en un **CheckboxGroup** hay que hacer uso de un constructor de **Checkbox** en el que se dice que se lo quiere incluir en un **CheckboxGroup** y en cual.

```
//: RadioButton1.java
// Using radio buttons
import java.awt.*;

public class RadioButton1 extends Frame {
  TextField t =
    new TextField("Radio button 2", 30);
  CheckboxGroup g = new CheckboxGroup();
  Checkbox
    cb1 = new Checkbox("one", g, false),
    cb2 = new Checkbox("two", g, true),
    cb3 = new Checkbox("three", g, false);
  public RadioButton1() {
    setLayout(new FlowLayout(
                  FlowLayout.CENTER, 20, 20));
    t.setEditable(false);
    add(t);
    add(cb1); add(cb2); add(cb3);
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
              System.out.println("Frame eliminado.");
              System.exit(0);
          }
          return super.handleEvent(e);
    }
    public boolean action (Event evt, Object arg) {
      if(evt.target.equals(cb1))
        t.setText("Radio button 1");
      else if(evt.target.equals(cb2))
        t.setText("Radio button 2");
      else if(evt.target.equals(cb3))
        t.setText("Radio button 3");
      else
        return super.action(evt, arg);
      return true;
    }
    public static void main(String[] args) {
        RadioButton1 f = new RadioButton1();
        f.setSize(300, 200);
        f.show();
    }
}
```

Choice

* **Choice** no es como el **Combo box** de Windows. Con **Choice** sólo se puede escoger un único elemento de la lista, aunque es posible añadir nuevos valores dinámicamente.

* Las opciones se añaden a un **Choice** con **addItem()**. La posición del elemento seleccionado se averigua con **getSelectedIndex()**. También puede usarse **getSelectedItem()** para recoger la cadena seleccionada, y **remove()** para eliminar opciones.

```
//: Choice1.java
// Using drop-down lists
import java.awt.*;

public class Choice1 extends Frame {
  String[] description = { "Hirviente", "Obtuso",
    "Recalcitrante", "Brillante", "Amuermante",
    "Primoroso", "Florido", "Putrefacto" };
  TextField t = new TextField(30);
  Choice c = new Choice();
  Button b = new Button("Add items");
  int count = 0;
  public Choice1() {
    setLayout(new FlowLayout(
                    FlowLayout.CENTER, 20, 20));
    t.setEditable(false);
    // Sólo se añaden las 4 primeras.
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    add(t);
    add(c);
    add(b);
  }
    public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY &&
            e.target == this) {
            System.out.println("Frame eliminado.");
            System.exit(0);
        }
        return super.handleEvent(e);
    }
  public boolean action (Event evt, Object arg) {
    if(evt.target.equals(c))
      t.setText("index: " +  c.getSelectedIndex()
          + "   " + (String)arg);
    else if(evt.target.equals(b)) {
      if(count < description.length)
          c.addItem(description[count++]);
    }
    else
      return super.action(evt, arg);
    return true;
  }
  public static void main(String[] args) {
      Choice1 f = new Choice1();
      f.setSize(300, 200);
      f.show();
  }
}
```

List

* Una lista ocupa un número concreto de líneas en pantalla, y permite selección múltiple. Los elementos también se añaden con **addItem()**. Los elementos seleccionados se devuelven en una array mediante **getSelectedItems()**.

```
//: List1.java
// Using lists with action()
import java.awt.*;

public class List1 extends Frame {
  String[] flavors = { "Chocolate", "Fresas",
    "Vainilla", "Chirimoya",
    "Pasas", "Chumbos",
    "Mango", "Damasquillos" };
  // Show 6 items, allow multiple selection:
  List lst = new List(6, true);
  TextArea t = new TextArea(flavors.length, 30);
  Button b = new Button("test");
  int count = 0;
  public List1() {
    setLayout(new FlowLayout(
                    FlowLayout.CENTER, 20, 20));
    t.setEditable(false);
    for(int i = 0; i < 4; i++)
      lst.addItem(flavors[count++]);
    add(t);
    add(lst);
    add(b);
  }
    public boolean handleEvent(Event e) {
          if (e.id == Event.WINDOW_DESTROY &&
              e.target == this) {
                System.out.println("Frame eliminado.");
                System.exit(0);
          }
          return super.handleEvent(e);
    }
    public boolean action (Event evt, Object arg) {
      if(evt.target.equals(lst)) {
        t.setText("");
        String[] items = lst.getSelectedItems();
        for(int i = 0; i < items.length; i++)
            t.appendText(items[i] + "\n");
      }
      else if(evt.target.equals(b)) {
        if(count < flavors.length)
            lst.addItem(flavors[count++], 0);
      }
      else
        return super.action(evt, arg);
      return true;
    }
    public static void main(String[] args) {
        List1 f = new List1();
        f.setSize(300, 200);
        f.show();
    }
}
```

MenuComponent

* Esta clase es abstracta, y de ella heredan los elementos que componen un menú y que son los siguientes:
- **MenuBar**. La barra de menús.
- **Menu**. Cada menú desplegable de la barra de menús.
- **MenuItem**. Cada elemento de un menú desplegable.
- **CheckboxMenuItem**. Igual que **MenuItem**, pero que le pone al lado una marca si dicha opción está seleccionada.

* La forma de trabajar es muy simple:
  - Los **MenuITem** se añaden a los **Menu** con **add()**. Lo mismo con los **CheckboxMenuItem**.
  - Es posible añadir un **Menu** a otro **Menu**.
  - Esposible incluir separadores en un **Menu** con **addSeparator()**.
  - Los **Menu** se añaden a los **MenuBar** con **add()**.
  - Un **MenuBar** se coloca en un **Frame** con **setMenuBar()**.
  - El estado de un **CheckboxMenuItem** se sabe con **getState()**.
  - Con **setEnabled()** podemos habilitar o inhabilitar cualquier opción.

```
//: Menu1.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Menus work only with Frames.
// Shows submenus, checkbox menu items
// and swapping menus.
import java.awt.*;

public class Menu1 extends Frame {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  TextField t = new TextField("No flavor", 30);
  MenuBar mb1 = new MenuBar();
  Menu f = new Menu("File");
  Menu m = new Menu("Flavors");
  Menu s = new Menu("Safety");
  // Alternative approach:
  CheckboxMenuItem[] safety = {
    new CheckboxMenuItem("Guard"),
    new CheckboxMenuItem("Hide")
  };
```

MenuComponent

```
MenuItem[] file = {
  new MenuItem("Open"),
  new MenuItem("Exit")
};
// A second menu bar to swap to:
MenuBar mb2 = new MenuBar();
Menu fooBar = new Menu("fooBar");
MenuItem[] other = {
  new MenuItem("Foo"),
  new MenuItem("Bar"),
  new MenuItem("Baz"),
};
Button b = new Button("Swap Menus");
public Menu1() {
  for(int i = 0; i < flavors.length; i++) {
    m.add(new MenuItem(flavors[i]));
    // Add separators at intervals:
    if((i+1) % 3 == 0)
      m.addSeparator();
  }
  for(int i = 0; i < safety.length; i++)
    s.add(safety[i]);
  f.add(s);
  for(int i = 0; i < file.length; i++)
    f.add(file[i]);
  mb1.add(f);
  mb1.add(m);
  setMenuBar(mb1);
  t.setEditable(false);
  add("Center", t);
  // Set up the system for swapping menus:
  add("North", b);
  for(int i = 0; i < other.length; i++)
    fooBar.add(other[i]);
  mb2.add(fooBar);
}
public boolean handleEvent(Event evt) {
  if(evt.id == Event.WINDOW_DESTROY)
    System.exit(0);
  else
    return super.handleEvent(evt);
  return true;
}
public boolean action(Event evt, Object arg) {
  if(evt.target.equals(b)) {
    MenuBar m = getMenuBar();
    if(m == mb1) setMenuBar(mb2);
    else if (m == mb2) setMenuBar(mb1);
  }
  else if(evt.target instanceof MenuItem) {
    if(arg.equals("Open")) {
      String s = t.getText();
      boolean chosen = false;
      for(int i = 0; i < flavors.length; i++)
        if(s.equals(flavors[i])) chosen = true;
      if(!chosen)
        t.setText("Choose a flavor first!");
      else
        t.setText("Opening "+ s +". Mmm, mm!");
    }
    else if(evt.target.equals(file[1]))
      System.exit(0);
    // CheckboxMenuItems cannot use String
    // matching; you must match the target:
    else if(evt.target.equals(safety[0]))
      t.setText("Guard the Ice Cream! " +
        "Guarding is " + safety[0].getState());
    else if(evt.target.equals(safety[1]))
      t.setText("Hide the Ice Cream! " +
        "Is it cold? " + safety[1].getState());
    else
      t.setText(arg.toString());
  }
  else
    return super.action(evt, arg);
  return true;
}
public static void main(String[] args) {
  Menu1 f = new Menu1();
  f.resize(300,200);
  f.show();
}
}
```

MenuITem setActionCommand() MenuShortCut

* Como puede observarse en el ejemplo anterior, un componente de un menú (una opción) queda identificado por el propio nombre que aparece en pantalla.

* Esto implica un grave problema de internacionalización, ya que cambiar el título a un menú implica retocar las entrañas del programa.

* Para evitar esto, podemos asociar un identificar independiente a cada opción, con **setActionCommand()**.

* Cuando se quiera preguntar por la opción que se ha escogido, cuando sepamos que se trata de una opción de menú, haremos uso de **getActionCommand()** para saber cuál es.

* De esta forma podemos cambiar los titulos sin preocuparnos de retocar el código.

* Por otro lado, el constructor de MenuItem permite especificar una tecla paraacceder rápidamente a esa opción del menú. Para ello se emplea la clase **MenuShortCut**.

Eventos de teclado, ratón y foco. Canvas

* Además de **action()** que recibe los eventos usuales, se tienen los siguientes métodos, definidos en la clase base **Component**:

-keyDown() Se ha pulsado una tecla y se mantiene pulsada.
- keyUp() Se ha liberado una tecla pulsada.

----··----

- lostFocus() El componente deja de tener el foco.
- gotFocus() El foco pasa a otro componente dentro del mismo contenedor.

----··----

- mouseDown() Se ha pulsado un botón del ratón.
- mouseUp() Se ha liberado un botón pulsado del ratón.
- mouseMove() El ratón se ha movido sobre el componente.
- mouseDrag() El ratón se está moviendo con un botón pulsado. Este mensaje se envía al componente sobre el que ocurrió el evento **mouseDown()**.
- mouseEnter() El ratón ha pasado a estar sobre el componente.
- mouseExit() El ratón estaba sobre el componente y ahora no.

* En el siguiente ejemplo, heredaremos de **Canvas** que es un área de pantalla rectangular en la que se puede dibujar, y que captura eventos. Normalmente debe especializarse y hay que reescribir su método **paint()** para que haga algo útil.

Ejemplo: foco, teclado, ratón. Canvas

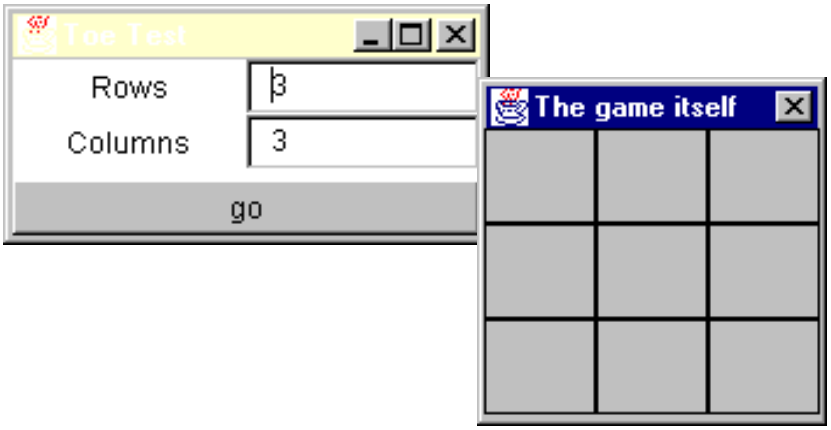
```
//: AutoEvent.java
// Alternatives to action()
import java.awt.*;
import java.util.*;

class MyButton extends Canvas {
  AutoEvent parent;
  Color color;
  String label;
  MyButton(AutoEvent parent,
           Color color, String label) {
    this.label = label;
    this.parent = parent;
    this.color = color;
  }
  public void paint(Graphics  g) {
    g.setColor(color);
    int rnd = 30;
    g.fillRoundRect(0, 0, size().width,
                    size().height, rnd, rnd);
    g.setColor(Color.black);
    g.drawRoundRect(0, 0, size().width,
                    size().height, rnd, rnd);
    FontMetrics fm = g.getFontMetrics();
    int width = fm.stringWidth(label);
    int height = fm.getHeight();
    int ascent = fm.getAscent();
    int leading = fm.getLeading();
    int horizMargin = (size().width - width)/2;
    int verMargin = (size().height - height)/2;
    g.setColor(Color.white);
    g.drawString(label, horizMargin,
                 verMargin + ascent + leading);
  }
  public boolean keyDown(Event evt, int key) {
    TextField t =
      (TextField)parent.h.get("keyDown");
    t.setText(evt.toString());
    return true;
  }
  public boolean keyUp(Event evt, int key) {
    TextField t =
      (TextField)parent.h.get("keyUp");
    t.setText(evt.toString());
    return true;
  }
  public boolean lostFocus(Event evt, Object w) {
    TextField t =
      (TextField)parent.h.get("lostFocus");
    t.setText(evt.toString());
    return true;
  }
  public boolean gotFocus(Event evt, Object w) {
    TextField t =
      (TextField)parent.h.get("gotFocus");
    t.setText(evt.toString());
    return true;
  }
  public boolean
  mouseDown(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseDown");
    t.setText(evt.toString());
    super.requestFocus();
    return true;
  }
  public boolean
  mouseDrag(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseDrag");
    t.setText(evt.toString());
    return true;
  }
  public boolean
  mouseEnter(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseEnter");
    t.setText(evt.toString());
    return true;
  }
  public boolean
  mouseExit(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseExit");
    t.setText(evt.toString());
    return true;
```

```
  }
  public boolean
  mouseMove(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseMove");
    t.setText(evt.toString());
    return true;
  }
  public boolean mouseUp(Event evt,int x,int y) {
    TextField t =
      (TextField)parent.h.get("mouseUp");
    t.setText(evt.toString());
    return true;
  }
}

public class AutoEvent extends Frame {
  Hashtable h = new Hashtable();
  String[] event = {
    "keyDown", "keyUp", "lostFocus",
    "gotFocus", "mouseDown", "mouseUp",
    "mouseMove", "mouseDrag", "mouseEnter",
    "mouseExit"
  };
  MyButton
    b1 = new MyButton(this, Color.blue, "test1"),
    b2 = new MyButton(this, Color.red, "test2");
  public AutoEvent() {
    setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
      TextField t = new TextField();
      t.setEditable(false);
      add(new Label(event[i], Label.CENTER));
      add(t);
      h.put(event[i], t);
    }
    add(b1);
    add(b2);
  }
    public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY &&
            e.target == this) {
            System.out.println("Frame eliminado.");
            System.exit(0);
        }
        return super.handleEvent(e);
    }
    public static void main(String[] args) {
      AutoEvent f = new AutoEvent();
      f.setSize(300, 200);
      f.show();
    }
}
```

* En este ejemplo se observa lo fácil que es crear nuestros propios componentes a partir de un **Canvas**.

Dialog

* Un diálogo es lo mismo que una ventana, pero que aparece como elemento idependiente de la principal.
* La diferencia estriba en que no puede tener menú, y, evidentemente, cuando se la cierra, no se sale del programa, sino que se llama a **dispose()** para que libere los recursos.
* Además, un **Dialog** no se añade al **Frame** actual, sino que se visualiza directamente con **show()**.




```
//: ToeTest.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in
Java"
// Demonstration of dialog boxes
// and creating your own components
import java.awt.*;

class ToeButton extends Canvas {
  int state = ToeDialog.BLANK;
  ToeDialog parent;
  ToeButton(ToeDialog parent) {
    this.parent = parent;
  }
  public void paint(Graphics  g) {
    int x1 = 0;
    int y1 = 0;
    int x2 = size().width - 1;
    int y2 = size().height - 1;
    g.drawRect(x1, y1, x2, y2);
    x1 = x2/4;
    y1 = y2/4;
    int wide = x2/2;
    int high = y2/2;
    if(state == ToeDialog.XX) {
      g.drawLine(x1, y1, x1 + wide, y1 + high);
      g.drawLine(x1, y1 + high, x1 + wide, y1);
    }
    if(state == ToeDialog.OO) {
      g.drawOval(x1, y1, x1+wide/2, y1+high/2);
    }
  }
  public boolean
  mouseDown(Event evt, int x, int y) {
    if(state == ToeDialog.BLANK) {
      state = parent.turn;
      parent.turn= (parent.turn == ToeDialog.XX ?
        ToeDialog.OO : ToeDialog.XX);
    }
    else
```

```
      state = (state == ToeDialog.XX ?
        ToeDialog.OO : ToeDialog.XX);
    repaint();
    return true;
  }
}

class ToeDialog extends Dialog {
  // w = number of cells wide
  // h = number of cells high
  static final int BLANK = 0;
  static final int XX = 1;
  static final int OO = 2;
  int turn = XX; // Start with x's turn
  public ToeDialog(Frame parent, int w, int h) {
    super(parent, "The game itself", false);
    setLayout(new GridLayout(w, h));
    for(int i = 0; i < w * h; i++)
      add(new ToeButton(this));
    resize(w * 50, h * 50);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      dispose();
    else
      return super.handleEvent(evt);
    return true;
  }
}

public class ToeTest extends Frame {
  TextField rows = new TextField("3");
  TextField cols = new TextField("3");
  public ToeTest() {
    setTitle("Toe Test");
    Panel p = new Panel();
    p.setLayout(new GridLayout(2,2));
    p.add(new Label("Rows", Label.CENTER));
    p.add(rows);
    p.add(new Label("Columns", Label.CENTER));
    p.add(cols);
    add("North", p);
    add("South", new Button("go"));
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    else
      return super.handleEvent(evt);
    return true;
  }
  public boolean action(Event evt, Object arg) {
    if(arg.equals("go")) {
      Dialog d = new ToeDialog(
        this,
        Integer.parseInt(rows.getText()),
        Integer.parseInt(cols.getText()));
      d.show();
    }
    else
      return super.action(evt, arg);
    return true;
  }
  public static void main(String[] args) {
    Frame f = new ToeTest();
    f.resize(200,100);
    f.show();
  }
}
```

FileDialog

* Por su enorme utilidad hay un tipo especial de diálogo que sirve para abrir y guardar ficheros. El siguiente ejemplo ilustra su utilización.

```
//: FileDialogTest.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in
Java"
// Demonstration of File dialog boxes
import java.awt.*;

public class FileDialogTest extends Frame {
  TextField filename = new TextField();
  TextField directory = new TextField();
  Button open = new Button("Open");
  Button save = new Button("Save");
  public FileDialogTest() {
    setTitle("File Dialog Test");
    Panel p = new Panel();
    p.setLayout(new FlowLayout());
    p.add(open);
    p.add(save);
    add("South", p);
    directory.setEditable(false);
    filename.setEditable(false);
    p = new Panel();
    p.setLayout(new GridLayout(2,1));
    p.add(filename);
    p.add(directory);
    add("North", p);
  }
  public boolean handleEvent(Event evt) {
    if(evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    else
      return super.handleEvent(evt);
    return true;
  }
  public boolean action(Event evt, Object arg) {
    if(evt.target.equals(open)) {
      // Two arguments, defaults to open file:
      FileDialog d = new FileDialog(this,
        "What file do you want to open?");
      d.setFile("*.java"); // Filename filter
      d.setDirectory("."); // Current directory
      d.show();
      String openFile;
      if((openFile = d.getFile()) != null) {
        filename.setText(openFile);
        directory.setText(d.getDirectory());
      } else {
        filename.setText("You pressed cancel");
        directory.setText("");
      }
    }
    else if(evt.target.equals(save)) {
      FileDialog d = new FileDialog(this,
        "What file do you want to save?",
        FileDialog.SAVE);
      d.setFile("*.java");
      d.setDirectory(".");
      d.show();
      String saveFile;
      if((saveFile = d.getFile()) != null) {
        filename.setText(saveFile);
        directory.setText(d.getDirectory());
      } else {
        filename.setText("You pressed cancel");
        directory.setText("");
      }
    }
    else
      return super.action(evt, arg);
    return true;
  }
  public static void main(String[] args) {
    Frame f = new FileDialogTest();
    f.resize(250,110);
    f.show();
  }
}
```

Java 1.1

* Java 1.1 cambia radicalmente la forma de capturar los eventos.
* En Java 1.1 cada componente se mete en una lista de objetos a la escucha de un tipo de evento. Cuando un evento tal se produce sobre dicho componente se llama a otro objeto asociado que tiene un método para cada evento concreto del tipo que queremos controlar.
* Todas las listas de tipos de eventos acaban en **Listener**, y empiezan por el tipo de evento que quiere controlar:

Tipo evento	Situaciones que acepta
Action	Las que acepta **action()**.
Adjustment	Las que se hacen sobre barras de desplazamiento.
Component	Ocultación y redimensionamiento de componentes.
Container	Adición y eliminación de componentes.
Focus	Toma y pérdida del foco.
Key	Eventos sobre teclas.
Mouse	Todos los eventos de ratón excepto arrastrar y mover.
MouseMotion	Arrastrar y mover el ratón.
Window	Todos los eventos relacionados con ventanas, excepto los de **Component**.
Item	Todo lo que tenga que ver con selección de elementos de una lista, o del tipo sí/no.
Text	Si se ha cambiado un texto o no.

Eventos en Java 1.1

* Un componente se añade a una lista de oyentes de eventos con **addXXXListener()**, donde XXX es una de las listas anteriores.

* Un componente se elimina de una lista de oyentes con **removeXXXListener()**.

* Cuando se añade un componente a una lista de oyentes, se debe indicar un objeto que empezará a ejecutarse cuando se produzca es eevento sobre el objeto oyente.

* Ese objeto ejecutor, debe implementar la interfaz **XXXListener**, y debe poseer métodos que se ejecutan cuando se produce cada evento concreto.

* Cada método recibe siempre un evento del tipo **XXXEvent**, cuyas características hay que mirar en la documentación.

* Los eventos que controla exactamente cada **XXXListener** son:

interfaz **Listener**	Métodos de que dispone
ActionListener	**actionPerformed()**
AdjustmentListener	**adjustmentValueChange()**
ComponentListener	**componentHidden()** **componentShown()** **componentMoved()** **componentResized()**
ContainerListener	**componentAdded()** **componentRemoved()**

interfaz **Listener**	Métodos de que dispone
FocusListener	**focusGained()** **focusLost()**
KeyListener	**keyPressed()** **keyReleased()** **keyTyped()**
MouseListener	**mouseClicked()** **mouseEntered()** **mouseExited()** **mousePressed()** **mouseReleased()**
mouseMotionListener	**mouseDragged()** **mouseMoved()**
WindowListener	**windowOpened()** **windowClosing()** **windowClosed()** **windowActivated()** **windowDeactivated()** **windowIconified()** **windowDeiconified()**
ItemListener	**ItemStateChanged()**
TextListener	**textValueChanged()**

* Como puede observarse, hay interfaces que poseen más de un método, con lo que los objetos ejecutores deben implementar todos esos métodos.

* Para evitarlo, cada **Listener** con más de un método tiene una clase asociada con todos esos métodos ya escritos por defecto. Así, en lugar de implementar la interfaz, se puede extender el **Adapter** asociado y reescribir sólo los métodos que interesen.
* Se dispone de los siguientes **Adapter**: **ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter** y **WindowAdapter.**

Eventos soportados

* Los tipos de eventos soportados por cada componente son los siguientes:

Componente	Tipos de eventos soportados
Adjustable	AdjustmentEvent
Applet	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Button	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Componente	Tipos de eventos soportados
Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
ScrollBar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

* Para que estos nuevos tipos de eventos sean reconocidos, es necesario hacer un **import** de **java.awt.event.*.**

Ejemplo de un botón

```
//: Button2New.java
// Capturing button presses
import java.awt.*;
import java.awt.event.*; // Must add this

public class Button2New extends Frame {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  TextField  t  = new TextField(15);
  public Button2New() {
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
    add(b1);
    add(b2);
    add(t);
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Boton 1.");
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Boton 2.");
    }
  }
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  public static void main(String[] args) {
      Button2New f = new Button2New();
      f.addWindowListener(new WL());
      f.setSize(300, 200);
      f.show();
  }
}
```

* Nótese que la clase WL es estática para que pueda ser usada dentro del **main()**, que es un método estático.

Ejemplo de un botón

* El ejemplo anterior se puede simplificar haciendo uso de clases internas.

```
//: Button2New.java
// Capturing button presses
import java.awt.*;
import java.awt.event.*; // Must add this

public class Button2New extends Frame {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  TextField  t  = new TextField(15);
  public Button2New() {
    b1.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                    t.setText("Boton 1.");
                                }
                               }
                            );
    b2.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                    t.setText("Boton 2.");
                                }
                               }
                            );
    setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
    add(b1);
    add(b2);
    add(t);
  }
  public static void main(String[] args) {
      Button2New f = new Button2New();
      f.addWindowListener(new WindowAdapter() {
                                    public void windowClosing(WindowEvent e) {
                                      System.exit(0);
                                    }
                                  }
                                );
      f.setSize(300, 200);
      f.show();
  }
}
```

Aplicaciones y *applets* todo en uno

* Para hacer que una misma aplicación pueda funcionar tanto como *applet* como como aplicación independiente, basta con:

- Crear el *applet* como siempre.
- Añadirle un **main()** que lo único que hace es crear un **Frame** al que le añade el *applet*.

```
//: Button2NewB.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// An application and an applet
import java.awt.*;
import java.awt.event.*; // Must add this
import java.applet.*;
public class Button2NewB extends Applet {
  Button
    b1 = new Button("Button 1"),
    b2 = new Button("Button 2");
  TextField t = new TextField(20);
  public void init() {
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
    add(t);
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button 1");
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Button 2");
    }
  }
  // To close the application:
  static class WL extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
  // A main() for the application:
  public static void main(String[] args) {
    Button2NewB applet = new Button2NewB();
    Frame aFrame = new Frame("Button2NewB");
    aFrame.addWindowListener(new WL());
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300,200);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
}
```

Java Swing

* Swing es una nueva librería que aparece con Java 1.2. Swing posee la práctica totalidad de componentes que se echan en falta hasta ahora.

* Swing utiliza sólo el formato de eventos de Java 1.1, y no el de Java 1.0.

* Swing es una enorme y compleja librería, que pretende reemplazar a la AWT utilizada hasta ahora.

* Como introducción a Swing podemos decir que posee todos los elementos que hemos visto hasta ahora, con la diferencia de que a los nombres de las clases les antepone la letra **J** para evitar confusiones.

* Vamos a echar un vistazo general a algunas de las cosas nuevas que incorpora Swing.

* Una característica de Swing es la homogeneidad en los nombres que emplea. Para asignar una propiedad V al objeto O, se hace **o.setV(valor)**. Para averiguar el valor de dicha propiedad se hace **o.getV()**.

* La más simple de todas ellas, es la posibilidad de que nos salga una etiquetita cuando el ratón está un cierto tiempo sobre un componente. Cualquier cosa que herede de **Jcomponent** tiene el método **setToolTipText(String),** al que se le pasa la etiquetita a visualizar.

Bordes

* **Jcomponent** tiene un método llamado **setBorder()** que permite colocar bordes a cualquier componente. Los bordes heredan de AbstractBorder.

* Los bordes disponibles son BevelBorder, EmptyBorder, EtchedBorder, LineBorder, MatteBorder, SoftBevelBorder y TitledBorder. Es posible mezclar varios bordes con CompoundBorder.

```
//: Borders.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Different Swing borders
package c13.swing;
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.border.*;

public class Borders extends JPanel {
  static JPanel showBorder(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public Borders() {
    setLayout(new GridLayout(2,4));
    add(showBorder(new TitledBorder("Title")));
    add(showBorder(new EtchedBorder()));
    add(showBorder(new LineBorder(Color.blue)));
    add(showBorder(
      new MatteBorder(5,5,30,30,Color.green)));
    add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    add(showBorder(      new
       SoftBevelBorder(BevelBorder.LOWERED)));
    add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
  }
  public static void main(String args[]) {
    Show.inFrame(new Borders(), 500, 300);
  }
}
```

* En esta y sucesivas clases, vamos a hacer uso de una clase **Show**, que posee un método estático al que se le pasa un Jpanel y se encarga de visualizarlo dentro de un **Frame**.

Botones

* Swing dispone de una enorme cantidad de nuevos botones, como son:
- BasicArrowButton. Son botones que tienen una flechita dibujada. Se les puede poner la flechita en dirección a cualquier punto cardinal.
- JtoggleButton. Botón que se queda pulsado.
- JcheckBox. Igual que Checkbox.
- JradioButton. Igual que CheckboxGroup con un solo botón.
- Jbutton. Igual que Button.

* Una diferencia entre estos botones y los que ya hemos visto es, entre otras, que heredan de **Jcomponent,** y por tanto, pueden poseer bordes.

* En el ejemplo que sigue también se ilustra el uso de **Spinner** y **StringSpinner**, que son contadores rotatorios en formato numérico, y en formato cadena respectivamente. En el ejemplo puede verse su utilización.

Botones

```
//: Buttons.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Various Swing buttons
package c13.swing;
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.plaf.basic.*;
import java.awt.swing.border.*;

public class Buttons extends JPanel {
  JButton jb = new JButton("JButton");
  BasicArrowButton
    up = new BasicArrowButton(
      BasicArrowButton.NORTH),
    down = new BasicArrowButton(
      BasicArrowButton.SOUTH),
    right = new BasicArrowButton(
      BasicArrowButton.EAST),
    left = new BasicArrowButton(
      BasicArrowButton.WEST);
  Spinner spin = new Spinner(47, "");
  StringSpinner stringSpin =
    new StringSpinner(3, "",
      new String[] {
        "red", "green", "blue", "yellow" });
  public Buttons() {
    add(jb);
    add(new JToggleButton("JToggleButton"));
    add(new JCheckBox("JCheckBox"));
    add(new JRadioButton("JRadioButton"));
    up.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        spin.setValue(spin.getValue() + 1);
      }
    });
    down.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        spin.setValue(spin.getValue() - 1);
      }
    });
    JPanel jp = new JPanel();
    jp.add(spin);
    jp.add(up);
    jp.add(down);
    jp.setBorder(new TitledBorder("Spinner"));
    add(jp);
    left.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        stringSpin.setValue(
          stringSpin.getValue() + 1);
      }
    });
    right.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        stringSpin.setValue(
          stringSpin.getValue() - 1);
      }
    });
    jp = new JPanel();
    jp.add(stringSpin);
    jp.add(left);
    jp.add(right);
    jp.setBorder(
      new TitledBorder("StringSpinner"));
    add(jp);
  }
  public static void main(String args[]) {
    Show.inFrame(new Buttons(), 300, 200);
  }
}
```

Iconos

* Es posible incluir un icono dentro cualquier cosa que herede de **AbstractButton**. Para ello se emplean las clases **Icon** e **ImageIcon**.

* Para ello, se crea un **ImageIcon** pasándole como parámetro el nombre del fichero .gif a recoger, e **ImageIcon** devuelve un **Icon** que ya podemos manejar.

* Se pueden tener distintos iconos para un mismo componente. Se usará uno u otro icono dependiendo de la acción que se haga sobre el botón asociado. Las acciones son:
- No se hace nada con el botón.
- El ratón se desplaza sobre el botón.
- El botón está presionado.
- El botón está desactivado.

* Es posible hacer que los iconos salgan en distintas posiciones del botón en el que se integra, con **setVerticalAligment()** y **setHorizontalAlignment()**.

Iconos

```
//: Faces.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Icon behavior in JButtons
package c13.swing;
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;

public class Faces extends JPanel {
  static Icon[] faces = {
    new ImageIcon("face0.gif"),
    new ImageIcon("face1.gif"),
    new ImageIcon("face2.gif"),
    new ImageIcon("face3.gif"),
    new ImageIcon("face4.gif"),
  };
  JButton
    jb = new JButton("JButton", faces[3]),
    jb2 = new JButton("Disable");
  boolean mad = false;
  public Faces() {
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(mad) {
          jb.setIcon(faces[3]);
          mad = false;
        } else {
          jb.setIcon(faces[0]);
          mad = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Yow!");
    add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Enable");
        } else {
          jb.setEnabled(true);
          jb2.setText("Disable");
        }
      }
    });
    add(jb2);
  }
  public static void main(String args[]) {
    Show.inFrame(new Faces(), 300, 200);
  }
}
```

Visualización de ejemplos

* Por último, la clase Show que se pedía para visualizar todos estos ejemplos, puede ser algo así:

```
//: Show.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Tool for displaying Swing demos
package c13.swing;
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;

public class Show {
  public static void
  inFrame(JPanel jp, int width, int height) {
    String title = jp.getClass().toString();
    // Remove the word "class":
    if(title.indexOf("class") != -1)
      title = title.substring(6);
    JFrame frame = new JFrame(title);
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        System.exit(0);
      }
    });
    frame.getContentPane().add(jp, BorderLayout.CENTER);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
}
```

* Cuando se trabaja con Swing, no es posible hacer **frame.setLayout()**; en su lugar hay que trabajar con la parte del **Frame** destinada a contener cosas, lo cual se consigue con **frame.getContentPane().setLayout()**.

JDBC

* Una de las características más versátiles de Java, es la facilidad con que trabaja con los recursos de Internet, y en concreto con las bases de datos a disposición pública con ODBC.

* Para ello, Java dispone de JDBC (Java DataBase Connectivity), así como del estándar SQL-92 para efectuar los accesos a la base de datos.

* De esta forma, JDBC es independiente de la plataforma concreta en que se encuentra la base de datos.

* Para permitir la independencia, JDBC proporciona un ***gestor de drivers***, que mantiene en todo momento los objetos *drivers* que el programa necesite para funcionar. Así, si nos conectamos a tres bases de datos de distintos fabricantes, tendremos un objeto *driver* de cada uno de ellos.

* Para abrir una base de datos hay que crear un URL de base de datos (URL: Uniform Resource Locator), que indique:
- Que estamos usando JDBC: "jdbc".
- El subprotocolo, o nombre del mecanismo de conexión que se está utilizando: "odbc".
- El identificador de la base de datos, tal como se haya declarado en el administrador de ODBC de la máquina en la que ejecuta el programa Java.

En nuestro caso emplearemos **jdbc:odbc:prueba1**.

JDBC

* Una vez que disponemos del URL en cuestión, los pasos a seguir para efectuar la conexión son los siguientes:

1.- Cargar el driver que permite la conexión con bases de datos de ODBC:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

2.- Configurar la base de datos en DSN del sistema en ODBC. Esto es porpio de cada plataforma. En Windows se hace a través de administrador de ODBC.

3.- Efectuar la conexión:

Connection c = DriverManager.getConnection(dbUrl, user, password);

que retorna un objeto **Connection**, que representa a la base de datos.

4.- Crear la sentencia SQL:

Statement s = c.createStatement();

Además hay que meter en un **String** el texto de la consulta SQL.

5.- Ejecutar la sentencia SQL y guardar los resultados en un **ResultSet**:

ResultSet r =s.executeQuery(textoDeLaSentencia);

6.- Recorrer el **ResultSet** para inspeccionar todas las tuplas seleccionadas:

```
while(r.next()) {
        /* Las tuplas se recogen con:
                r.getString(nombreDeCampo)
                r.getBoolean(nombreDeCampo)
                r.getDate(nombreDeCampo)
                r.getDouble(nombreDeCampo)
                r.getFloat(nombreDeCampo)
                r.getInt(nombreDeCampo)
                r.getLong(nombreDeCampo)
                r.getShort(nombreDeCampo)
                r.getTime(nombreDeCampo)
        */
    }
    s.close(); // Also closes ResultSet
```

Ejemplo de JDBC

```
//: Lookup.java

// Looks up email addresses in a
// local database using JDBC
import java.sql.*;

public class Lookup {
  public static void main(String[] args) {
    String dbUrl = "jdbc:odbc:prueba1";
    String user = "";
    String password = "";
    try {
      // Load the driver (registers itself)
    Class.forName(
          "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
          dbUrl, user, password);
      Statement s = c.createStatement();
      // SQL code:
      ResultSet r =
          s.executeQuery(
            "SELECT Nombre, Apellidos, NIF " +
            "FROM Clientes " +
            "WHERE " +
            "(NIF > '3000') " +
            "ORDER BY Apellidos");
      while(r.next()) {
          // Capitalization doesn't matter:
          System.out.println(
            r.getString("Nombre") + ", "
            + r.getString("aPELLIDOS")
            + ": " + r.getString("NIF") );
      }
      s.close(); // Also closes ResultSet
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Comunicación vía Sockets

* Vamos a ver cómo dos ordenadores pueden comunicarse vía *sockets*.

* Se llama *socket* al objeto que cada interlocutor crear para gestionar la conexión con el otro.

* A este respecto hay dos clases fundamentales que intervienen aquí.
- **ServerSocket**. Es una clase que se dedica a escuchar por la red (con **accept()**) a ver si alguien se quiere poner en contacto con ella.
- **Socket**. Es el gestor de la conexión.

+ El servidor crea un **ServerSocket** para escuchar por la red.
+ El cliente crea un **Socket** para ponerse en contacto con un servidor.
+ Cuando el **ServerSocket** se da cuenta de que alguien se quiere comunicar con él, crea un **Socket** en el servidor de manera que ya puede empezar la comunicación.
+ Una vez que cada uno tiene su **Socket**, se utiliza **getInputStream** y **getOutputStream** para establecer la comunicación.

* Dado que unamisma máquina puede tener muchos programas servidores, cada **ServerSocket** tiene asignado un puerto, de forma que cuando alguien se quiere conectar a nosotros, no sólo especifica nuestra máquina, sino también el puerto.

Servidor

```
//: JabberServer.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
            ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {
        System.out.println(
         "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(

socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
           System.out.println("Echoing: " +
str);
          out.println(str);
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
}
```

* Cuando el **ServerSocket** hace el **accept()** se queda dormido hasta que recibe una petición de conexión, comento en el que devuelve el **Socket** correspondiente.
* No debemos olvidar nunca cerra tanto el **Socket** como el **ServerSocket**.
* Se ha hecho uso del estrato **PrintWriter**, que vacía el buffer automáticamente (*flush*) tras cada retorno de carro.

Cliente

```
//: JabberClient.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in
Java"
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
}
```

* Para efectuar la conexión, basta con crear un **Socket**, en cuya construcción se indican la máquina y el puerto de conexión. Para especificar la máquina de conexión se utiliza un objeto de tipo **InetAddress**.

* Para coger una máquina basta con llamar al método estático **getByName()**, al que se le pasa la dirección internet como **String**.

Servir a muchos clientes

```
//: MultiJabberServer.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in
Java"
// A server that uses multithreading to handle
// any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
      }
      System.out.println("closing...");
    } catch (IOException e) {
    } finally {
      try {
          socket.close();
      } catch(IOException e) {}
    }
  }
}

public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
          // Blocks until a connection occurs:
          Socket socket = s.accept();
          try {
            new ServeOneJabber(socket);
          } catch(IOException e) {
            // If it fails, close the socket,
            // otherwise the thread will close it:
            socket.close();
          }
      }
    } finally {
      s.close();
    }
  }
}
```

* Cuando se cierra la conexión el *thread* desaparece.
* Los **try-catch** sirven para asegurarse de que los *sockets* se cierran.

Acceso por Internet a Bases de Datos. *Applets* y *threads*

* Un *applet* puede comunicarse con el ordenador-servidor del que procede. Es una forma de conectar un ordenador cliente con una base de datos situada en el servidor.

* El servidor puede conectarse a una base de datos y suministrar información al *applet*.

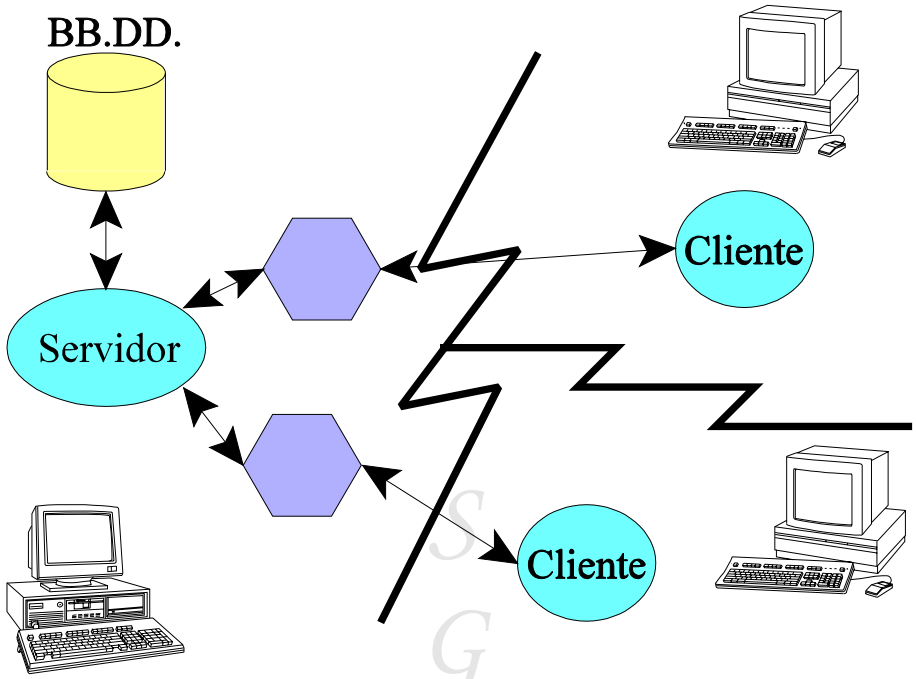
* El servidor debe crear un *thread* por cada petición que reciba. Cada *thread* debe servir a un cliente, y destruirse cuando acabe su servicio.

* El servicio del *thread* consiste en enviar los datos requeridos al *applet* cliente.

* Los datos enviados pueden ser:
    - Datos textuales que el *applet* directamente.
    - El nombre de una página **.html** que el *thread* crea dinámicamente, y en la que coloca los datos en forma de tabla.

Internet, BB.DD. y *Applets*

* La comunicación sigue el siguiente esquema:



* Los pasos que se siguen son:
  - El ***applet*** se conecta al **servidor**.
  - El **servidor** crea un ***thread*** de atención.
  - El ***applet*** solicita los datos al ***thread***.
  - El ***thread*** crea una página **.html** dinámicamente, con los datos solicitados.
  - El ***thread*** le dice al ***applet*** el nombre de la página.
  - El ***thread*** desaparece.
  - El ***applet*** carga la página en el navegador.

* Cada cierto tiempo hay que eliminar las páginas **.html** creadas dinámicamente.

Creación de tablas en HTML

* Las tablas en HTML siguen una estructura muy sencilla:
- Empiezan por <TABLE> y acaban con </TABLE>.
- Cada línea comienza con <TR>.
- Cada cabecera de cada columna comienza por <TH>.
- Cada dato de cada columna comienza por <TD>.

* El siguiente fichero HTML:

```
<TABLE>
<TR><TH> Col1 <TH> Col2 <TH> Col3
<TR><TD> Dato11 <TD> Dato12 <TD> Dato13
<TR><TD> Dato21 <TD> Dato22 <TD> Dato23
<TR><TD> Dato31 <TD> Dato32 <TD> Dato33
</TABLE>
```

produce la tabla:

Col1	Col2	Col3
Dato11	Dato12	Dato13
Dato21	Dato22	Dato23
Dato31	Dato32	Dato33

* A una tabla se le pueden especificar distintos tipos de bordes.

Internet y BB.DD. El *applet*

* El *applet* quedaría como sigue:

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

public class ClientDB extends Applet{
    TextField t1 = new TextField(10);
    TextField t2 = new TextField(10);
    TextArea a = new TextArea();
    Button b = new Button("Enviar");
    public void init(){
        setLayout(new FlowLayout());
        add(t1); add(t2); add(a);
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e){alfa();}
                                }
        );
        add(b);
    }

    public void alfa(){
        PrintWriter out;
        BufferedReader in;
        try{
            Socket s = new Socket(InetAddress.getByName("altamira.es"), 8080);
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));
            out = new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                s.getOutputStream())), true);
            out.println(t1.getText());
            out.println(t2.getText());
            out.println("END"); // Forma parte del protocolo de comunicaciones.
            out.println("SELECT "+ t1.getText() + ", " + t2.getText() + " " + a.getText());
            String nombreFichero = in.readLine();
            if ((nombreFichero == null) || (nombreFichero.equals("ERROR")))
                getAppletContext().showStatus(nombreFichero);
            else
                getAppletContext().showDocument(new URL("http://altamira.es/" +
nombreFichero));
            s.close();
        } catch (IOException e){
            System.out.println("Hay algún error con la red");
        }
    }
}
```

Internet y BB.DD. Servidor

* El servidor sería:

```
import java.net.*;
import java.io.*;
import java.awt.*;
import java.sql.*;

public class ServerDB{
    public static void main(String args[]){
      Connection c;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            c = DriverManager.getConnection("jdbc:odbc:cursjava", "", "");
            ServerSocket ss = new ServerSocket(8080);
            while(true){
                System.out.println("A la espera");
                Socket s = ss.accept();
                System.out.println("Procesando...");
                new Acceso(s, c).start();
            };
            // c.close();
        } catch (IOException e){
            System.out.println("Hay algún error con la red.");
        } catch (Exception e){
            System.out.println("Error con la base de datos o con la clase del ODBC.");
        }
    }
}
class Acceso extends Thread{
    BufferedReader in; PrintWriter out;
    Connection c;
    Socket s;
    public Acceso(Socket k, Connection h){
        c = h; s = k;
        try{
            in = new BufferedReader(
                        new InputStreamReader(
                            s.getInputStream()));
            out = new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                s.getOutputStream())), true);
        } catch (IOException e){
            System.out.println("Hay algún error con la red al abrir el socket");
        }
    }
```

```
    public void run(){
        String[] campos = new String[11]; // Sólo 10 campos. El 11-avo es para "END".
        int cont = 0;
        String nombreFichero = new String("file" + System.currentTimeMillis() + ".html");
        try{
            if ((in == null) || (out == null))
                System.out.println("No se inicia comunicación. Cancelado.");
            else{
                do{
                    campos[cont++] = in.readLine();
                } while ((! campos[cont - 1].equals("END")) && (cont <= 11));
                cont--;
                if (! campos[cont].equals("END")){
                    System.out.println("Demasiados campos. Cancelado.");
                    return;
                }
                String sentencia = in.readLine();
                Statement s = c.createStatement();
                ResultSet r = s.executeQuery(sentencia);
                // Aquí construiremos una página html nueva.
                // Paso 1. Abrir el fichero plantilla.
                DataInputStream fileIn =    new DataInputStream(
                                                new BufferedInputStream(
                                                    new FileInputStream("plantilla.txt")));
                // Paso 2. Copiar el fichero plantilla en el fichero resultado.
                PrintWriter fileOut =      new PrintWriter(
                                               new BufferedOutputStream(
                                                   new FileOutputStream(nombreFichero)));
                String entrada;
                while ((entrada = fileIn.readLine()) != null)
                    fileOut.println(entrada);
                fileIn.close();
                // Paso 3. Copiar en el fichero resultado los datos obtenidos.
                fileOut.println("<Table Border=\"0\"><tr>");
                // Paso 3.1. Escribir la cabecera.
                for (int i = 0; i < cont; i++)
                    fileOut.println("<th> " + campos[i]);
                // Paso 3.2. Escribir los datos.
                while (r.next()){
                    fileOut.println("<tr>");
                    for (int i = 0; i < cont; i++)
                        fileOut.println("<td> " + r.getString(campos[i]));
                }
                fileOut.println("</Table>");
                fileOut.println("</body>");
                fileOut.println("</html>");
                fileOut.close();
            }
            out.println(nombreFichero);
            s.close();
```

```
        } catch (SQLException e){
            System.out.println("Error en la sentencia SQL.");
            e.printStackTrace();
            try{
                out.println("ERROR");
            } catch (Exception ex){}
        }   catch (IOException ex){
            System.out.println("Error de E/S.");
        } catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

Servlets

* A menudo conviene establecer comunicación entre una página **.html** y el servidor, pero sin necesidad de tener un *applet* ejecutándose en el cliente. Java 1.2 admite esta posibilidad.

* El ejemplo del punto anterior también se puede desarrollar mediante una página **.html** y un *servlet* conectado a dicha página.

* La página **.html** es un formulario que el navegante rellena en la máquina cliente. Cuando el navegante pulsa el botón "Enviar", el navegador envía el contenido del formulario a un programa ubicado en la máquina servidora, llamado *servlet*.

* El *servlet* crea dinámicamente una página **.html** (sin necesidad de guardarla en un fichero), y se la envía al navegador que, entonces, la muestra al navegante como resultado de su consulta a través del formulario.

* Para que todo esto funcione, en la máquina servidora debe ejecutarse un servidor de *servlets*. Usaremos el JSWDK 1.0 (Java Servlets Web Development Kit versión 1.0), colocaremos los servlets en el directorio"jswdk-1.0\webpages\WEB-INF\servlets\".

* El servidor de servlets se arranca con **startserver**. Para que funcione correctamente es necesario colocar en el CLASSPATH al fichero **servlet.jar**.

* ES NECESARIO USAR UN SERVIDOR WEB.

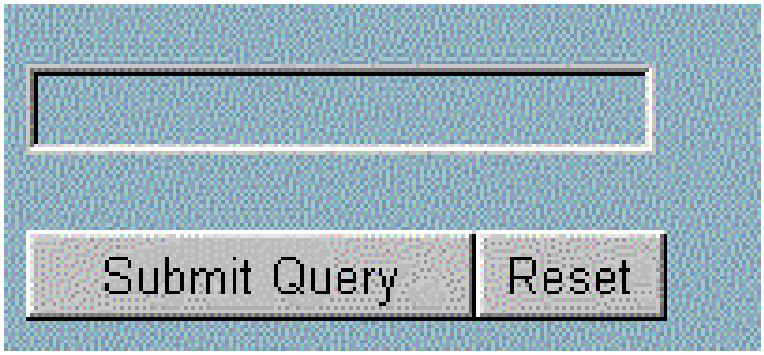
HTML y formularios

* Los formularios en HTML constan de las siguientes etiquetas:
- Una etiqueta de comienzo <FORM> y otra de final </FORM>.
- La etiqueta <INPUT> para definir la estructura del formulario. Éstas pueden ser de varios tipos (**type**):
    + **submit**: Crea el botón para enviar el formulario.
    + **reset**: Crea el botón para borrar el contenido.
    + **text**: Crea un campo de texto a ser relleno por el usuario navegante. Al campo se le da un nombre con **name**.
- La etiqueta <FORM> lleva asociada la acción (**action**) que hay que realizar cuando se pulsa el botón **submit**. Esta acción es el nombre del *servlet* a ejecutar, incluyendo la URL en que se encuentra.

* El siguiente fichero HTML:

```
<form action=http://betelgeuse.lcc.uma.es:8080/servlet/MiPrimerServlet
method=”POST”>
<BR>
<BR>Introduzca un texto en el cuadro y pulse "Submit"<BR>
<BR>
<input type=text name=TEXTO>
<BR>
<BR><input type=submit><input type=reset></form>
```

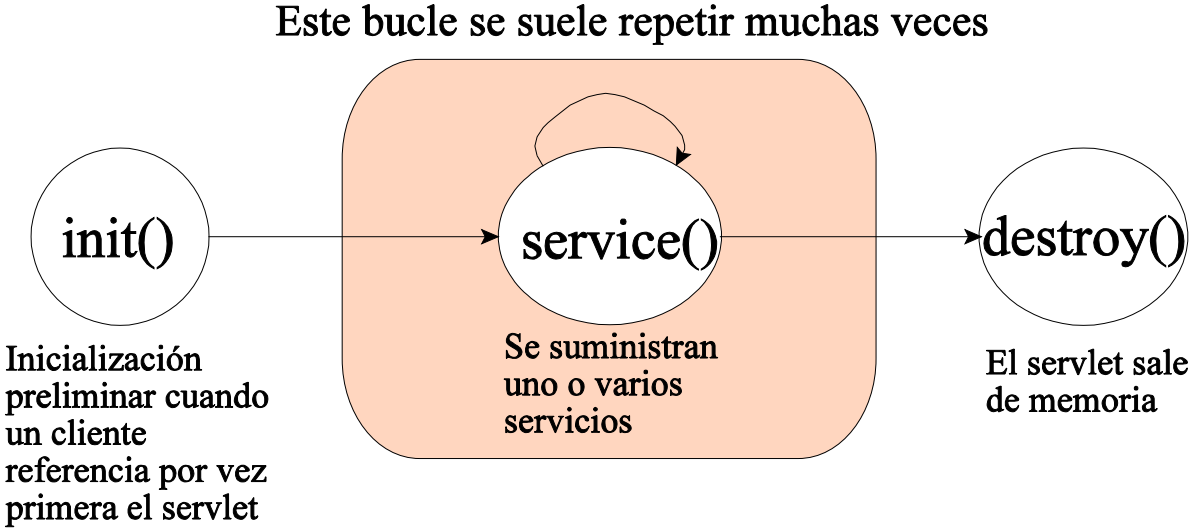
produce el formulario:



Ciclo de vida de un *Servlet*

* Un *servlet* tiene un ciclo de vida similar al de un *applet*:
  - El servidor de *servlets* carga los *servlets* bajo demanda.
  - La primera vez que carga un *servlet* crea un objeto cuyo tipo es ese *servlet* e invoca a su **init()**.
  - **init()** debe inicializar todos aquellos componentes que tengan vigencia mientras dicho objeto siga vivo: abrir bases de datos, crear *threads*, etc.
  - Para cada solicitud de servicio por parte de un cliente, el servidor invoca a la función **service** del objeto anteriormente creado.
  - Cuando hace mucho que el objeto de ese *servlet* no recibe peticiones, entonces queda en desuso y debe desaparecer de memoria: se invoca a **destroy()** quien debe: cerrar bases de datos, eliminar *threads*, etc.

* ¡Cuidado! Los *servlets* son *reentrantes* lo que quiere decir que si se reciben varias peticiones a un mismo *servlet*, el servidor creará varios *threads* que acceden simultáneamente a un mismo objeto *servlet*: usar **synchronized** cuando sea necesario.



Clases para *Servlets*

* Todo *servlet* debe heredar de **HttpServlet**.

* Cuando el servidor invoca al *servlet* llama a su método **service**, quien a su vez llama al método **doXxx**, donde **Xxx** representa el método de comunicación usado en la página **.html**, y en el que habrá que colocar el código de lo que se quiere hacer.

* Las funciones **doXxx()** toman dos parámetros, que permiten la comunicación con el formulario enviado: uno del tipo **HttpServletRequest** y otro del tipo **HttpServletResponse**.

* **HttpServletResponse** admite el mensaje **getOutputStream()** que devuelve un objeto del tipo **PrintStream**. A través de este canal escribiremos la página **.html**, que aparecerá automáticamente en el navegador del cliente.

* Antes de enviar datos por el canal de salida, hay que especificarle al navegador la naturaleza de los datos que se van a transmitir. Ello se hace a través del mensaje **setContentType(String)**. Para las páginas **.html**, el valor de la cadena debe ser "text/html".

* Para tomar cada campo del formulario origen, se invoca el mensaje **getParameter(String)** de **HttpServletRequest**, al que se le pasa como parámetro el nombre del campo de texto cuyo contenido se desea conocer.

Ejemplo de *Servlet*

* Un *servlet* que recoja los datos del formulario visto anteriormente sería:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class MiPrimerServlet extends HttpServlet
{

  public void doGet(HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException
  {
    PrintStream out = new PrintStream(res.getOutputStream());
    res.setContentType("text/html");
    String TEXTO    = req.getParameter("TEXTO");
    out.println("<p>Usted ha escrito : "+TEXTO+"</p>");
  }
}
```

* La clase **HttpServletRequest** posee diversos métodos:
  - **getRemoteAddr()** devuelve la dirección TCP/IP del cliente.
  - **getRemoteHost()** devuelve el nombre del cliente.
  - **getParameterNames()** devuelve un **Enumeration** con los parámetros pasados por el formulario.

* La utilización de *cookies* es relativamente fácil a través de la clase **Cookie**.

Capítulo 12: RMI

* RMI (*Remote Method Invocation*).

* Esto permite tener objetos Java en otros ordenadores de Internet, que reciban mensajes, y que los ejecuten.

* El objeto remoto es el servidor, y el que manda el mensaje el cliente.

* En la máquina cliente se debe disponer de la interfaz que suministra el objeto remoto.

* En la máquina remota, el objeto debe estar vivo permanentemente, a la espera de recibir mensajes.

* Para conseguir esto, en el cliente se siguen los siguientes pasos:
- Crear la interfaz como pública.
- Dicha interfaz debe extender a **java.rmi.Remote**.
- Cada método de la interfaz puede lanzar la excepción **java.rmi.Remote.Exception**.
- Si un objeto remoto se pasa como argumento, su tipo debe ser el de la interfaz que declaramos.

```
//: PerfectTimeI.java
// The PerfectTime remote interface
package c15.ptime;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
}
```

RMI. El servidor

* La clase correspondiente en el servidor debe extender a la clase **UnicastRemoteObject**, e implementar la interfaz remota.

* Debe poseer obligatoriamente un constructor, que puede elevar la excepción **RemoteException**.

* El programa **main()** debe instalar al gestor de seguridad, de la forma:
System.setSecurityManager(new RMISecurityManager());

* Crear cuantas instancias se necesiten del objeto remoto, y darles un nombre con: **Naming.bind(maquina, objeto**), donde **maquina** es una cadena con la dirección IP y el nombre del objeto, y objeto es un manejador al objeto remoto. Máquina puede poseer un puerto por el que se establecerá la comunicación. Para hacer que un objeto deje de estar disponible, se hace **unbind()**.

* Para que lo anterior funciones, el servidor debe tener funcionando en *background* **rmiregistry**, lo que se consigue con:
rmiregistry *puerto*
desde MS-DOS. **rmiregistry** debe ser capaz de encontrar las clases de los objetos que sirve.

* Cualquier parámetro que se le pase en un mensaje a un objeto remoto, o que éste devuelva, debe implementar **Serializable**.

La comunicación

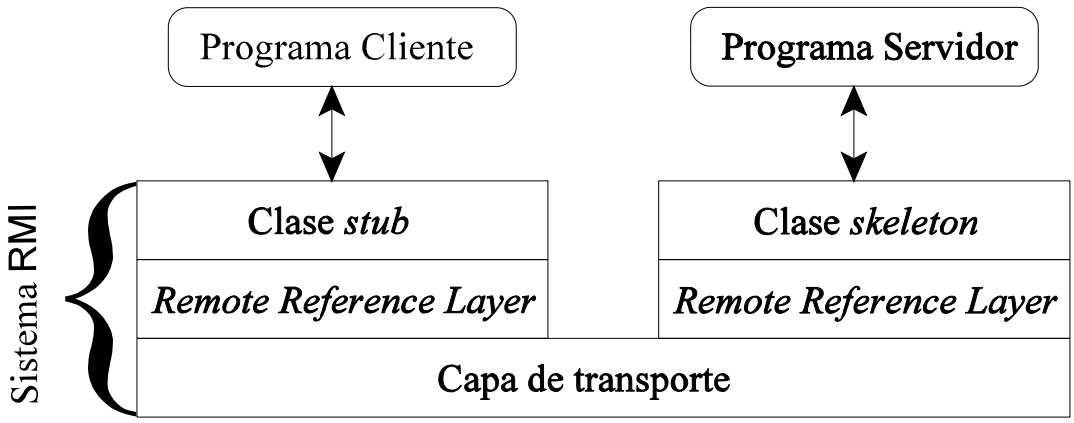
* Aunque el objeto remoto resida en una máquina diferente, para que el cliente lo pueda utilizar necesita un representante de dicho objeto: es lo que se llama *stub* o *proxy*.

* Para comunicar el *stub* con el objeto remoto, el servidor necesita un *skeleton*. (en la versión JDK 1.2 no es necesario).

* La comunicación es como sigue:
  - El cliente se comunica con el *stub*.
  - El *stub* se comunica con el *skeleton* a través de la red (protocolo de transporte).
  - El *skeleton* se comunica con el objeto remoto.

* Para generar el *stub* y el *skeleton* de la clase del objeto remoto, hay que ejecutar desde MS-DOS **rmic *nombreClase***, lo que nos generará dos nuevos ficheros necesarios para la comunicación: **nombreClase_Stub.class** y **nombreClase_Skeleton.class**

* Los *stub* y *skeleton* también se encargan de serializar los parámetros.



RMI. El servidor

```
//: PerfectTime.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// The implementation of the PerfectTime remote object
package c15.ptime;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving:
  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      PerfectTime pt = new PerfectTime();
      Naming.bind("//colossus:2005/PerfectTime", pt);
      System.out.println("Ready to do time");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

* Para que todo funcione bien (en JDK 1.2), hay que modificar la política de seguridad, establecida por el fichero *java.policy* del directorio *jre/lib/security/*, añadiendo:

```
grant {
        permission java.security.AllPermission;
};
```

RMI. El cliente

* El cliente trata los objetos temotos como si de cualquier otro objeto se tratase, con la única diferencia de que en vez de crearlo, tiene que buscarlo en el servidor con:
Naming.lookup(maquina_y_nombreObjeto);
lo que devuelve un **Object** que habrá que convertir a la interfaz que se tiene disponible en el cliente.

```
//: DisplayPerfectTime.java
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// Uses remote object PerfectTime
package c15.ptime;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[] args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTimeI t =
        (PerfectTimeI)Naming.lookup(
          "//colossus:2005/PerfectTime");
      for(int i = 0; i < 10; i++)
        System.out.println("Perfect time = " +
          t.getPerfectTime());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Parámetros

* El servidor debe ejecutar **Naming.bind(...)** para registrar el objeto en el servidor de nombres de objetos **rmiregistry**.

* Esto permite que un cliente acceda a un objeto remoto a través de su nombre con **Naming.lookup(...)**.

* Pero no todo objeto remoto tiene porqué ser registrado con un nombre. Es posible pasar un objeto remoto como parámetro en una llamada remota a otro objeto remoto.

* Lo que verdaderamente convierte a un objeto en remoto es:
  - Que herede de **UnicastRemoteObject**.
  - Que existan los necesarios *stub* y *skeleton* para la comunicación.

* Si la clase del objeto que se quiere hacer remoto ya hereda de otra clase, no puede heredar de **UnicastRemoteObject**. Contra este problema existe una solución alternativa: un objeto cualquiera se puede hacer remoto con el mensaje:

UnicastRemoteObject.exportObject(objeto **);**

* **rmiregistry** puede registrar objetos generados por diferentes programas java (diferentes *JVM*) ubicados en un mismo servidor.

* Si el ordenador servidor sólo va a tener un programa (*JVM*) que registre objetos remotos, no es necesario ejecutar **rmiregistry** aparte, sino que el propio programa puede arrancar una instancia de **rmiregistry** dedicada a él sólo. Esto se hace con: **LocateRegistry.createRegistry(PUERTO);**

Retrollamadas

* RMI permite que cliente y servidor puedan intercambiar sus papeles.

```
// Copyright MageLang Institute;
import java.net.*;
import java.io.*;
import java.util.Date;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;

public class RMIServer implements Remote, TimeServer {
  private static final int    PORT        = 10005;
  private static final String HOST_NAME   = "betelgeuse";
  private static RMIServer  rmi;
  public static void main ( String[] args ) {
    // We need to set the security manager to the RMISecurityManager
    System.setSecurityManager( new RMISecurityManager() );
    try {
      rmi = new RMIServer();
      LocateRegistry.createRegistry( PORT );
      System.out.println( "Registry created" );
      UnicastRemoteObject.exportObject( ((TimeServer)rmi) );
      Naming.rebind( "//" + HOST_NAME + ":" + PORT + "/" + "TimeServer", rmi );
      System.out.println( "Bindings Finished" );
      System.out.println( "Waiting for Client requests" );
    } catch ( java.rmi.UnknownHostException uhe ) {
            System.out.println( "The host computer name you have specified, " +
HOST_NAME + " does not match your real computer name." );
    } catch ( RemoteException re ) {
      System.out.println( "Error starting service" );
      System.out.println( "" + re );
    } catch ( MalformedURLException mURLe ) {
      System.out.println( "Internal error" + mURLe );
    }
  }  // main
  public void registerTimeMonitor( TimeMonitor tm ) {
    System.out.println( "Client requesting a connection" );
    TimeTicker tt;
    tt = new TimeTicker( tm );
    tt.start();
    System.out.println( "Timer Started" );
  }
}  // class RMIServer
```

```
class TimeTicker extends Thread {
  private TimeMonitor tm;
  TimeTicker( TimeMonitor tm ) {
    this.tm = tm;
  }
  public void run() {
    while ( true ) {
      try {
        sleep( 2000 );
        tm.tellMeTheTime( new Date() );
      } catch ( Exception e ) {
        stop();
      }
    }
  }
}

// Copyright MageLang Institute;
import java.rmi.*;
public interface TimeServer extends java.rmi.Remote {
  public void registerTimeMonitor( TimeMonitor tm) throws RemoteException;
}
```

* En este ejercicio, un *applet* va a invocar a un objeto remoto para que le diga la hora cada dos segundos. Para dicho objeto remoto el *applet* es, a su vez, otro objeto remoto.

Retrollamadas

```
// Copyright MageLang Institute;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.Date;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;

public class Applet1 extends Applet implements TimeMonitor {
  private final static String HOST_NAME = "betelgeuse";
  private TimeServer ts;
  public void init() {
    super.init();
    uiInit();
    try {
      System.out.println( "Exporting the Applet" );
      UnicastRemoteObject.exportObject( this );
      URL     base = getDocumentBase();
      String hostName = base.getHost();
      if ( 0 == hostName.length() ) {
        hostName = HOST_NAME;
      }
       String  serverName = "rmi://" + hostName + ":" + getParameter( "registryPort" )
+ "/TimeServer" ;
      System.out.println( "Looking up TimeService at: " + serverName );
      try {
        ts = (TimeServer)Naming.lookup( serverName );
      } catch ( Exception e ) {
        System.out.println( "" + e );
      }
      ts.registerTimeMonitor( this );
      System.out.println( "We have been registered!" );
    } catch ( RemoteException re ) {
      System.out.println( "" + re );
    }
        }

  public void tellMeTheTime( Date d ) {
    textArea1.appendText( d.toString() + "\n" );
  }
```

```
  public void uiInit() {
    setLayout(null);
    resize(456,266);
    textArea1 = new java.awt.TextArea();
    textArea1.reshape(36,24,252,170);
    add(textArea1);
    button1 = new java.awt.Button("Clear");
    button1.reshape(324,36,72,24);
    add(button1);
    button1.addActionListener (
      new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          textArea1.setText("");
        }
      }
    );
  }
  java.awt.TextArea textArea1;
  java.awt.Button button1;
}

// Copyright MageLang Institute;
import java.rmi.*;
import java.util.Date;
public interface TimeMonitor extends java.rmi.Remote {
  public void tellMeTheTime( Date d ) throws RemoteException;
}
```

* El *applet* necesita un parámetro que le diga por qué puerto debe comunicarse. Este parámetro se le pasa en el correspondiente fichero .html.

Descarga de clases

* Un .class se puede cargar remotamente a través de la clase **RMIClassLoader**, mediante la función estática:

Class loadClass(url, nombreClase)

* La clase **Class** admite el método **newInstance()**, que devuelve un objeto de la clase representada. Dado que devuelve un **Object** es necesario realizar un *downcasting*.

* Es posible indicarle a Java que debe buscar los .class en una máquina diferente. Esto se hace a través de la propiedad **java.rmi.server.codebase**, que se hace igual a una URL que será la máquina servidora que contiene los .class.

* La máquina servidora debe servir HTTP, FTP o similar. Sun suministra la clase **ClassFileServer** como servidora.

* La propiedad **java.rmi.server.useCodebaseOnly**, cuando se coloca a *true* hace que las clases remotas sólo se puedan cargar desde **java.rmi.server.codebase**.

* La propiedad **java.rmi.server.logCalls** hace que se visualicen por el dispositivo **stderr** los mensajes de conexión.

Las propiedades se pueden especificar en el momento de ejecutar Java, con: **java -Dnombre.propiedad=valor NombreClase**

* Tras crear todas las clases y tener el servidor de clases, el siguiente programa se ejecuta con:

java -Djava.rmi.server.codebase=http://betelgeuse:2002/ RMIClientLoader

Descarga de clases

```
 // Copyright MageLang Institute;
import java.net.*;
import java.io.*;
import java.util.Properties;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;
public class RMIClientLoader {
  private static final int    PORT        = 10009;
  private static final String HOST_NAME   = "betelgeuse";
  private static RMIClientLoader rcl;
  private Class    clientClass;
  private URL        url;
  private Properties p;
  private String clientName;
  public RMIClientLoader() throws MalformedURLException,
ClassNotFoundException,
      InstantiationException, IllegalAccessException {
    p = System.getProperties();
    url = new URL(p.getProperty("java.rmi.server.codebase"));
    System.out.println("" + url);
    // El cliente que descargamos puede incluso acceder a un objeto remoto
    // situado en cualquier lugar. La siguiente propiedad le puede decir tal lugar.
    p.put("java.rmi.server.rminode", "rmi://" + HOST_NAME + ":" + PORT + "/");
    clientName = "RMIClient";
    System.out.println("Asking for: " + url + " and " + clientName);
    clientClass = RMIClassLoader.loadClass(url, clientName);
    System.out.println("After loading Client Class");
    ((Runnable)clientClass.newInstance()).run;
  }
  public static void main (String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      rcl = new RMIClientLoader();
    } catch (MalformedURLException mURLe) {
      System.out.println("URL not specified correctly for the Client class: " + mURLe);
    } catch (ClassNotFoundException cnfe) {
      System.out.println("RMIClientLoader, class not found: " + cnfe);
    } catch (InstantiationException ie) {
      System.out.println("RMIClientLoader, class could not be instantiated" + ie);
    } catch (IllegalAccessException iae) {
      System.out.println("Internal error" + iae);
    }
  }  // main
}  // class RMIClientLoader
```

Política de seguridad

* La política de seguridad se puede cambiar de varias maneras:
    - Mediante la propiedad **java.security.policy**.
    - Mediante el fichero *java.policy*.
    - Creando un nuevo *SecurityManager*

```
// Copyright MageLang Institute;
/**
 * Esta clase define una política de seguridad para aplicaciones
 * RMI cargadas desde un servidor. La relajación suministrada por
 * esta clase es la mínima necesaria para que todo funcione.
 *
 * Los cambios con respecto al RMISecurityManager por defecto son:
 *
 * Security Check                This Policy   RMISecurityManager
 * ----------------------------  -----------   ------------------
 * Access to Thread Groups           SÍ              NO
 * Access to Threads                 SÍ              NO
 * Create Class Loader               SÍ              NO
 * System Properties Access          SÍ              NO
 * Connections                       SÍ              NO
 *
 */

public class RMIClientBootstrapSecurityManager
    extends RMISecurityManager {
// Loaded classes are allowed to create class loaders.
  public synchronized void checkCreateClassLoader() {}
// Connections to other machines are allowed
  public synchronized void checkConnect(String host, int port) {}
// Loaded classes are allowed to manipulate threads.
  public synchronized void checkAccess(Thread t) {}
// Loaded classes are allowed to manipulate thread groups.
  public synchronized void checkAccess(ThreadGroup g) {}
// Loaded classes are allowed to access the system properties list.
  public synchronized void checkPropertiesAccess() {}
}
```

* Para más información, consultar la clase **RMISecurityManager**.

Activación remota de objetos

* Hasta ahora, para acceder un objeto remoto, el servidor lo creaba, lo registraba y *el servidor seguía funcionando todo el tiempo*.

* Con JDK 1.2 es posible crear un objeto remoto que se active automáticamente cuando sea necesario, y *sin que el servidor funcione todo el tiempo*.

* Para ello, el *stub* del cliente posee un **identificador de activación** y una **referencia viva** al objeto, que estará a **null** si el objeto aún no ha sido activado.

* Cuando se envía un mensaje a un objeto *dormido*, hay que activarlo en el servidor. en este proceso se ven implicados:

- Una **referencia de fallo** (el mensaje no se ha podido ejecutar bien porque el objeto estaba dormido).
- Un **activador**.
- Un **grupo de activación** en una JVM.

* El **activador** (uno por servidor) mantiene información entre los **identificadores de activación** y la información interna necesaria para la activación.

* Un **grupo de activación** (uno por JVM) recibe la petición de activar un objeto en su JVM y le devuelve al **activador** el objeto activado.

Protocolo de activación

* El protocolo de activación:

- Una **referencia de fallo** usa un **identificador de activación** y llama al **activador** (clase interna de RMI) para activar al objeto asociado.
- El **activador** busca el **descriptor de activación** (que ha debido ser previamente registrado), y que contiene:
  > El **identificador de grupo** (indica la JVM en que se ejecutará el objeto).
  > El nombre de la clase del objeto.
  > La URL donde encontrar su .class.
  > Datos de inicialización en formato plano (**marshalled**).

* Para poder utilizar el sistema de activación automático, hay que ejecutar en *background* el dispensador de activaciones **rmid** (RMI *daemon*). **rmiregistry** debe estar activado también.

* Para todo esto se utilizan las clases:
ActivationGroup
ActivationGroupID
ActivationGroupDesc
ActivationDesc
MarshalledObject
Activatable
ActivationGroupDesc.CommandEnvironment

Ejemplo de activación

```
// Copyright (c) 1998, 1999, 2000 Sun Microsystems, Inc. All Rights Reserved.
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
public class Setup {
   // This class registers information about the ActivatableImplementation
   // class with rmid and the rmiregistry
   public static void main(String[] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        // Because of the Java 2 security model, a security policy should
        // be specified for the ActivationGroup VM.
        Properties props = new Properties();
      props.put("java.security.policy", "\\jdk1.2\\jre\\lib\\security\\java.policy");
      ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props, ace);
        // Once the ActivationGroupDesc has been created, register it
        // with the activation system to obtain its ID
      ActivationGroupID agi =
        ActivationGroup.getSystem().registerGroup(exampleGroup);
        // Now explicitly create the group
        ActivationGroup.createGroup(agi, exampleGroup, 0);
        // The "location" String specifies a URL from where the class
        // definition will come when this object is requested (activated).
      // Don't forget the trailing slash at the end of the URL or classes won't be found.
        String location = "file:/Sergio/temp/c15/activo/";
        // Create the rest of the parameters that will be passed to
        // the ActivationDesc constructor
        MarshalledObject data = null;
        // The second argument to the ActivationDesc constructor will be used
        // to uniquely identify this class; it's location is relative to the
        // URL-formatted String, location.
        ActivationDesc desc = new ActivationDesc
                ("ActivatableImplementation", location, data);
        // Register with rmid
        MyRemoteInterface mri = (MyRemoteInterface)Activatable.register(desc);
        System.out.println("Got the stub for the ActivatableImplementation");
        // Bind the stub to a name in the registry running on 1099
        Naming.rebind("ActivatableImplementation", mri);
        System.out.println("Exported ActivatableImplementation");
        System.exit(0);
   }
}
```

Ejemplo de activación

```
// Copyright (c) 1998, 1999, 2000 Sun Microsystems, Inc. All Rights Reserved.
import java.rmi.*;
public interface MyRemoteInterface extends Remote {
        public Object callMeRemotely() throws RemoteException;
}


import java.rmi.*;
import java.rmi.activation.*;
public class ActivatableImplementation extends Activatable
    implements MyRemoteInterface {
   // The constructor for activation and export; this constructor is
   // called by the method ActivationInstantiator.newInstance during
   // activation, to construct the object.
   public ActivatableImplementation(ActivationID id, MarshalledObject data)
        throws RemoteException {
        // Register the object with the activation system
        // then export it on an anonymous port
        super(id, 0);
   }
   public Object callMeRemotely() throws RemoteException {
        return "Perfecto";
   }
}
```

* El cliente que hace uso del objeto, no distingue si es **rmi** normal o **rmi activable**.

* Un objeto se puede exportar de dos formas: a) heredando de **Activatable** b) llamando a **exportObject(...)**, de la forma:

Activatable.exportObject(this, id, 0);

donde **id** y **0** tienen el mismo sentido que la llamada al constructor de la clase padre en caso de que sí heredase.

Ejemplo de activación

* La clase **CommandEnvironment** permite cambiar las propiedades por defecto del sistema para un **ActivationGroup** concreto.

* Son necesarios un **CommandEnvironment** junto con unas **Properties** para crear un **descriptor de grupo de activación**.

* Es necesario registrar el **descriptor** para obtener un **identificador de grupo de activación**.

* A continuación se crea el grupo con **createGroup(...)**.

* El **descriptor de activación** se crea con una llamada de la forma: new ActivationDesc (ClaseActivable, ubicacionDeClass, datosMarshalled);, donde los datosMarshalled suelen ser null.

* El descriptor se registra con una llamada de la forma: Activatable.register(descriptor);, lo que devuelve un objeto que habrá que registrar posteriormente de la manera tradicional, con **Naming.bind(...)**.

* Nótese como en ningún momento se hace un **new** del objeto que se pretende exportar, sino que se le pueden dar datos por defecto mediante un objeto **MarshalledObject**.

Capítulo 13: Ejecución de programas no Java. JNI

* Algunas veces es necesario hacer cosas muy dependientes de la máquina donde se va a ejecutar el programa Java.

* Dado que Java es independiente de la máquina hay que recurrir a otros lenguajes de programación, como C.

* La ejecución y comunicación con programas escritos en otro lenguaje es muy fácil. Se utiliza la clase **Process**, y el método estático **Runtime.getRuntime().exec()**, que admite como parámetro una cadena con el nombre del fichero .exe a ejecutar.

* La comunicación con el fichero .exe es muy simple: en Java creamos *streams* de entrada y de salida, que en el .exe pasan a ser su salida estándar y su entrada estándar.

* No hay que olvidar tanto en el .exe como en el programa Java, vaciar los *bufferes* cuando escribimos algo, para asegurarnos de que el otro interlocutor lo recibe en ese mismo momento.

* Ej. en C:

```
#include <stdio.h>
void main() {
        int i, j;
        scanf("%d %d", &i, &j);
        printf("%d\n", i*j);
        fflush(stdout);
}
```

Ejecución de programas no Java

```
// Conexion.java
import java.util.*;
import java.io.*;

public class Conexion {
   public static void main(String[] args) {
         Process p;
         PrintStream escritor;
         DataInputStream lector;
         try {
            p = Runtime.getRuntime().exec("Mult.exe");
            escritor = new PrintStream(
                              new BufferedOutputStream(
                                    p.getOutputStream()
                                 )
                           );
            lector = new DataInputStream(
                              new BufferedInputStream(
                                    p.getInputStream()
                                 )
                           );
            System.out.println("Mando cosas al .exe.");
            escritor.println("34 56"); escritor.flush();
            System.out.println("Leo cosas del .exe.");
            System.out.println(lector.readLine());
         } catch (IOException e) {
            System.err.println("Imposible iniciar el ejecutable.");
            System.exit(1);
         }

   }
}
```

JNI

* JNI: *Java Native Interface*.

* JNI permite que una misma clase pueda tener unos métodos escritos en Java y otros métodos escritos en C.

* A los métodos escritos en C se les llama **nativos**, puesto que están compilados en código máquina y son nativos de un µp concreto.

* El código de los métodos nativos no se coloca en el .class, sino en una librería dinámica (archivo .dll en el caso de Windows)

* Desde el punto de vista de la clase Java, sólo hay que hacer tres cosas:

  - Declarar las funciones C como **native**, y no dotarlas de cuerpo.
  - En la inicialización de la clase hay que cargar la librería que contendrá el cuerpo escrito en C (*nombre.dll*) de la forma: **System.loadLibrary("nombre");**
  - Generar un fichero de cabecera .h asociado a la clase de Java, mediante la utilidad **javah**.

* Ej.:

```
class HolaMundo {
public native void displayHolaMundo();
        static {
                System.loadLibrary("hello");
        }
        public static void main(String[] args) {
                new HolaMundo().displayHolaMundo();
        }
}
```

JNI

* La utilidad **javah** genera un fichero de cabecera .h en lenguaje C, que debe ser incluido en el fichero C que implementa la función nativa.

* La función nativa en el programa en C no tiene exactamente el mismo nombre con el que se ha declarado en Java. Esto es así porque diferentes clases de Java pueden tener funciones nativas con el mismo nombre. Además se permite sobrecarga de funciones nativas.

* Por tanto, es necesario "decorar" el nombre de la función. Aunque hay ciertos criterios bien definidos para hacer esta decoración, el fichero cabecera que genera **javah** ya posee la declaración completa.

* El programa .C quedaría:

```
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj) {
        printf("Hello world!\n");
        return;
}
```

* La cabecera de la función ha sido copiada tal cual del fichero .C generado por **javah**.

Compilación del programa C

* El programa C debe dar lugar a una librería dinámica (.dll), y **nunca** a un programa .exe.

* La cabecera incluye el fichero *jni.h* que está en *C:\jdk1.2\include*, y al fichero *jni_md.h* ubicado en *C:\jdk1.2\include\win32*.

* Una libería .dll suele tener un punto de entrada (**LibMain()**) con varios parámetros. Suele ser necesario cambiar este punto de entrada a una función cualquiera inventada a tal efecto, y de cuerpo vacío.

* Algunas versiones del JDK 1.2 pueden dar problemas debido a la definición de la macro **JNIEXPORT**. En tales circunstancias basta con eliminar dicha palabra en la declaración de las funciones:

    JNI_GetDefaultJavaVMInitArgs(...)
    JNI_CreateJavaVM(...)
    JNI_GetCreatedJavaVMs(...)
    JNI_OnLoad(...)
    JNI_OnUnload(...)

dentro de la cabecera *jni.h*.

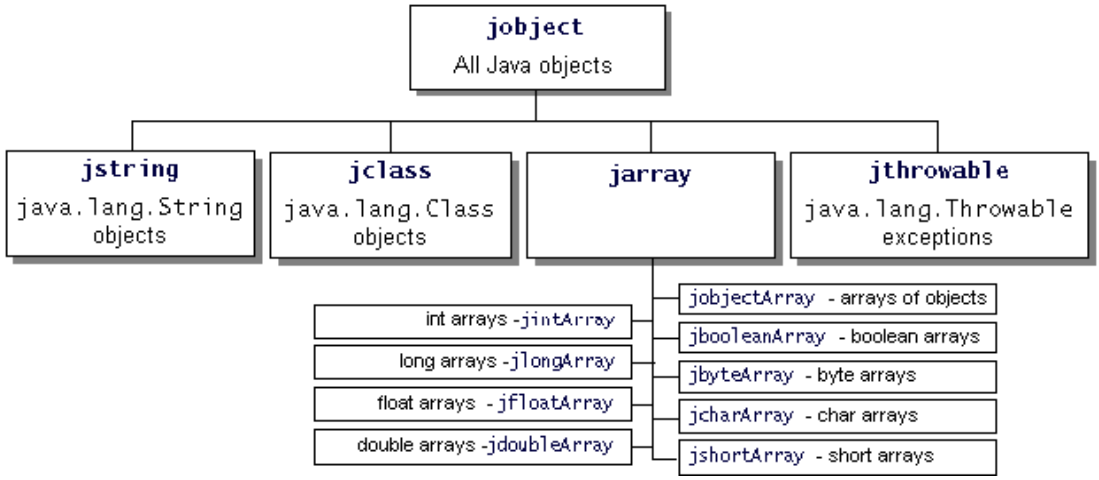
* El fichero .dll generado ha de tener el mismo nombre que se usó en **System.loadLibrary(...)**.

Paso de parámetros

* Los tipos primitivos de Java son directamente accesibles desde C a través de los tipos predefinidos en *jni.h*:

Tipo en Java	Tipo en C	Tamaño en bits
boolean	jboolean	8 (sin signo)
byte	jbyte	8
char	jchar	16 (sin signo)
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	--

* Los objetos de Java se pasan por referencia a los métodos nativos, y todos heredan de **jobject**. Además:



jstring

* El tipo de C **jstring** se usa para acceder a parámetros Java de tipo **String**.

* El fichero *jni.h* define muchas funciones que son accedidas a través de los parámetros **JNIEnv *env, jobject obj** que se pasan de forma automática a toda función nativa:

- **env** es un puntero que nos permite acceder al entorno Java (***env****ironment*), desde dentro de una función nativa.
- **obj** es el objeto Java que recibe el mensaje nativo.

* El tipo **jstring** no es como el *array* de **char** de C, puesto que un **jstring** trabaja con Unicode y no con ASCII puro.

* La siguiente función ilustra como hacer uso de un **jstring**:

```
JNIEXPORT jstring JNICALL
    Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt) {
        char buf[128];
        const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
        printf("%s", str);
        (*env)->ReleaseStringUTFChars(env, prompt, str);
        ...
}
```

* Algunas funciones para manejar **jstring** son:

- **GetStringUTFChars(...)** crea un *array* de **chars** en ASCII.
- **ReleaseStringUTFChars(...)** libera dicho *array*.
- **GetStringChars** retorna un puntero a un *array* de Unicodes.
- **ReleaseStringChars** libera el puntero anterior.
- **NewString** construye un **String** Java a partir de Unicodes.
- **GetStringLength** da la longitud de un *array* de Unicodes.
- **GetStringUTFLength** da la longitud de un *array* de ASCII.

Arrays nativos

* El tipo de C **j<*tipo*>Array** permite acceder a los *arrays* previa conversión, al igual que sucede con **jstring**.

```
NIEXPORT jint JNICALL
	Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {
	int i, sum = 0;
	jsize len = (*env)->GetArrayLength(env, arr);
	jint *body = (*env)->GetIntArrayElements(env, arr, 0);
	for (i=0; i<len; i++) {
		sum += body[i];
	}
	(*env)->ReleaseIntArrayElements(env, arr, body, 0);
	return sum;
}
```

* Las funciones que se usan son:
  - **GetArrayLength(...)**.
  - **Get<*tipo*>ArrayElements(...)**.
  - **Release<*tipo*>ArrayElements(...)**.
  - **Get<*tipo*>ArrayRegion(...)** permite obtener sólo un trozo del *array*, en lugar del *array* entero. Se suele emplear cuando el *array* es muy grande y consume mucho espacio.

* El programa Java sería tan sencillo como:

```
class IntArray {
	private native int sumArray(int arr[]);
	static {
		System.loadLibrary("MyImpOfIntArray");
	}
	public static void main(String args[]) {
		IntArray p = new IntArray();
		int arr[] = new int [10];
		for (int i = 0; i < 10; i++)
		arr[i] = i;
		System.out.println("sum = " + p.sumArray(arr));
	}
}
```

Acceso al objeto principal. Campos

* Desde una función nativa es posible acceder a los campos miembro y de clase del objeto **this** que recibe el mensaje.

* El acceso a un campo tal se produce en dos fases:

1.- Se obtiene un identificador del campo al que se quiere acceder (tipo **jfieldID**), mediante **GetStaticFieldID** o **GetFieldID**. Estas funciones toman como parámetro (además de **env**):

a) La clase del objeto principal, en formato **jclass**. Ello se consigue a través de la función **GetObjectClass(env, obj)**.

b) dos cadenas de caracteres que contienen el nombre del campo y su tipo "decorado" respectivamente.

2.- A partir del **jfieldID** se accede al campo que sea para leerlo con **GetStatic<*tipo*>Field** o **Get<*tipo*>Field**. Para escribirlo se usa **SetStatic<*tipo*>Field** o **Set<*tipo*>Field**. Según sea estático o no, se pasará como parámetro un **jclass** o un **jobject**.

* El tipo "decorado" de cada campo se conoce gracias a la utilidad desensambladora **javap** con las opciones **-s -p**. Esto devuelve una lista con todos los elementos de la clase junto con su tipo "decorado" en un comentario adjunto. Consultar las opciones en la ayuda de **javap**.

Acceso al objeto principal. Campos

```
class FieldAccess {
    static int si;
    String s;

    static {
        System.loadLibrary("MyImpOfFieldAccess");
    }
    private native void accessFields();
    public static void main(String args[]) {
        FieldAccess c = new FieldAccess();
        FieldAccess.si = 100;
        c.s = "abc";
        c.accessFields();
        System.out.println("In Java:");
        System.out.println("  FieldAccess.si = " +
                           FieldAccess.si);
        System.out.println("  c.s = \"" + c.s + "\"");
    }
}
```

```
#include <stdio.h>
#include "FieldAccess.h"

JNIEXPORT void JNICALL
    Java_FieldAccess_accessFields(JNIEnv *env,
    jobject obj) {
    jclass cls = (*env)->GetObjectClass(env, obj);
    jfieldID fid;
    jstring jstr;
    const char *str;
    jint si;

    printf("In C:\n");

    fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
    if (fid == 0) {
        return;
    }
    si = (*env)->GetStaticIntField(env, cls, fid);
    printf("  FieldAccess.si = %d\n", si);
    (*env)->SetStaticIntField(env, cls, fid, 200);

    fid = (*env)->GetFieldID(env, cls, "s",
"Ljava/lang/String;");
    if (fid == 0) {
        return;
    }
    jstr = (*env)->GetObjectField(env, obj, fid);
    str = (*env)->GetStringUTFChars(env, jstr, 0);
    printf("  c.s = \"%s\"\n", str);
    (*env)->ReleaseStringUTFChars(env, jstr, str);

    jstr = (*env)->NewStringUTF(env, "123");
    (*env)->SetObjectField(env, obj, fid, jstr);
}

void alfa(){} /* Punto de entrada a la dll */
```

Acceso al objeto principal. Métodos

* Desde una función nativa también es posible invocar métodos de un objeto, ya sea estático o no:

  - Los identificadores de los métodos se toman con **GetMethodID** o **GetStaticMethodID**. Ambos devuelven un **jmethodID**.
  - Los métodos se invocan con **Call<*tipo*>Method** o **CallStatic<*tipo*>Method**, que aceptan un número variable de argumentos (en función de los definidos en la clase Java).

* Hay otros métodos de invocar funciones.

```
class Callbacks {
    private native void nativeMethod(int depth);
    static {
        System.loadLibrary("MyImpOfCallbacks");
    }
    private void callback(int depth) {
        if (depth < 5) {
            System.out.println("En Java, nivel = "
                    + depth + ", entrando en C");
            nativeMethod(depth + 1);
            System.out.println("En Java, nivel = "
                    + depth + ", vuelta de C");
        } else
            System.out.println("En Java, nivel = "
                    + depth + ", límite alcanzado");
        }
    public static void main(String args[]) {
        Callbacks c = new Callbacks();
        c.nativeMethod(0);
    }
}
```

```
#include <stdio.h>
#include "Callbacks.h"

JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env,
    jobject obj, jint depth) {
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetMethodID(env, cls,
"callback", "(I)V");
    if (mid == 0) {
        return;
    }
    printf("En C, nivel = %d, entrando en Java\n",
depth);
    (*env)->CallVoidMethod(env, obj, mid, depth);
    printf("En C, nivel = %d, vuelta de Java\n",
depth);
    return;
}

void alfa(){}
```

Manejo de excepciones

* Una función nativa puede detectar si se ha producido una excepción al llamar a una función Java pura con:

jthrowable exc = (*env)->ExceptionOccurred(env);
if (exc) {/*Excepción producida*/}

* La función nativa puede elevar excepciones con:

```
newExcCls = (*env)->FindClass(env,
                              "java/lang/IllegalArgumentException");
if (newExcCls == 0) { /* No se ha podido encontrar la clase */
    return;
}
(*env)->ThrowNew(env, newExcCls, "thrown from C code");
```

* La función nativa también puede corregir una excepción elevada con:

(*env)->ExceptionClear(env);

Capítulo 14: Internacionalización

* Java dispone de mecanismos que facilitan la migración de programas escritos en un idioma a otro diferente.

* Las clases destinadas a la internacionalización se hallan en el archivo *i18n.jar*. El nombre de este archivo se deriva de que la palabra **internacionalización** posee 18 letras entre la **i** inicial y la **n** final.

* El proceso se basa en los llamados **Locale**, que es una clase que almacena información relativa a un idioma hablado en un determinado país.

* La clase **Locale** hace referencia a un conjunto de recursos de un idioma concreto: **ResourceBundle**.

* El siguiente ejemplo ilustra cómo se utiliza un objeto **Locale** para acceder a un **ResourceBundle**. A continuación, los mensajes se recogen mediante **getString(...)**.

* A continuación se muestra el contenido de los recursos i18n para los distintos idiomas admitidos.

I18n: Ejemplo básico

```
// Copyright (c) 1995-1998 Sun Microsystems, Inc. All Rights Reserved.
Import java.util.*;

public class I18NSample {

   static public void main(String[] args) {

      String language;
      String country;

      if (args.length != 2) {
         language = new String("es");
         country = new String("ES");
      } else {
         language = new String(args[0]);
         country = new String(args[1]);
      }

      Locale currentLocale;
      ResourceBundle messages;

      currentLocale = new Locale(language, country);

      messages =
        ResourceBundle.getBundle("MessagesBundle",currentLocale);

      System.out.println(messages.getString("bienvenida"));
      System.out.println(messages.getString("pregunta"));
      System.out.println(messages.getString("despedida"));
   }
}
```

Fichero **MessagesBundle_de_DE.properties**:

```
bienvenida = Hallo.
despedida = Tschüß.
pregunta = Wie geht's?
```

Fichero **MessagesBundle_en_US.properties**:

```
bienvenida = Hello.
despedida = Goodbye.
pregunta = How are you?
```

Fichero **MessagesBundle_es_ES.properties**:

```
bienvenida = Buenos días.
despedida = Adiós.
pregunta = ¿Cómo está?
```

Aspectos a internacionalizar

* Muchos son los aspectos a considerar cuando se cambia de idioma un programa:

- ✓ Mensajes
- ✓ Etiquetas sobre componentes GUI
- ✓ Ayuda en línea
- ✓ Sonidos
- ✓ Colores
- ✓ Gráficos
- ✓ Iconos
- ✓ Fechas
- ✓ Horas
- ✓ Números
- ✓ Valores monetarios
- ✓ Medidas
- ✓ Números de teléfono
- ✓ Títulos honoríficos y personales
- ✓ Direcciones de correo postal
- ✓ Distribución de páginas

* Trabajaremos con mensajes que intercalan datos, fechas y valores monetarios.

Texto y datos intercalados

* A menudo es necesario intercalar datos entre el texto:

"El 13 de abril de 1998 a las 1:15, se detectaron 7 naves sobre el planeta Marte"

* En otros idiomas, los datos pueden ir intercalados de otra forma, p.ej, en inglés la hora iría antes que la fecha.

* Para solucionar este problema se hace uso de la clase **MessageFormat**, que permite expresar en un fichero de recursos una expresión completa:

- Un objeto de tipo **MessageFormat** se asocia con un **Locale**.
- Luego se le aplica un patrón que se encuentra en el **ResourceBundle** correspondiente.
- Por último, éste mensaje se formatea con los parámetros que se quiera, almacenados en un *array* de **Object**.

* Los patrones posibles pueden consultarse en la ayuda de **MessageFormat**: se colocan entre llaves y llevan el número del parámetro del que toman los datos, además de otra información de formato.

Ejemplo con *MessageFormat*

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

public class MessageFormatDemo {
  static  void displayMessage(Locale currentLocale) {
    System.out.println("currentLocale = " + currentLocale.toString());
    System.out.println();
    ResourceBundle messages =
      ResourceBundle.getBundle("MessageBundle",currentLocale);
    Object[] messageArguments = {
      messages.getString("planet"),
      new Integer(7),
      new Date()
    };

    MessageFormat formatter = new MessageFormat("");
    formatter.setLocale(currentLocale);
    formatter.applyPattern(messages.getString("template"));
    String output = formatter.format(messageArguments);

    System.out.println(output);
  }
  static public void main(String[] args) {
    displayMessage(new Locale("es", "ES"));
    System.out.println();
    displayMessage(new Locale("en", "US"));
  }
}
```

Fichero **MessageBundle_en_US.properties**:

```
template = At {2,time,short} on {2,date,long}, we detected \
                {1,number,integer} spaceships on the planet {0}.
planet = Mars
```

Fichero **MessageBundle_es_ES.properties**:

```
template = El {2,date,long} a las {2,time,short}, se detectaron \
                {1,number,integer} naves sobre el planeta {0}.
planet = Marte
```

Formato de fechas y horas

* A estos efectos, el sistema dispone de forma predeterminada de algunos **Locale**. Probar los disponibles con el programa:

```
import java.util.*;
public class DameLocales {
	public static void main(String[] args){
		Locale[] l = Locale.getAvailableLocales();
		for (int i = 0; i<l.length; i++){
			System.out.println(l[i].toString());
		}
	}
}
```

* Se realiza mediante un proceso en dos fases:
	- Se crea un objeto de tipo **DateFormat** mediante un **getDateInstance(...)**.
	- **getDateInstance(...)** toma como parámetros el estilo de fecha que se quiere sacar y el **Locale** correspondiente.
	- Se indica la fecha a convertir.

* Los estilos de fechas que se pueden sacar son:
	- DateFormat.DEFAULT
	- DateFormat.SHORT
	- DateFormat.MEDIUM
	- DateFormat.LONG
	- DateFormat.FULL

Fechas y horas. Ejemplo

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

public class DateFormatDemo {
    static Date today=new Date();
    static String result;
    static DateFormat formatter;
    static int[] styles = {
      DateFormat.DEFAULT,
      DateFormat.SHORT,
      DateFormat.MEDIUM,
      DateFormat.LONG,
      DateFormat.FULL
    };
  static public void global(Locale currentLocale){
    System.out.println();
    System.out.println("Locale: " +
                currentLocale.toString());
    System.out.println();
  }

  static public void displayDate(Locale
currentLocale) {
    Date today;
    String dateOut;
    DateFormat dateFormatter;
    dateFormatter =
      DateFormat.getDateInstance(
                DateFormat.DEFAULT,
currentLocale);
    today = new Date();
    dateOut = dateFormatter.format(today);
    System.out.println(dateOut + "   " +
                currentLocale.toString());
  }

  static public void showBothStyles(Locale
currentLocale) {
                global(currentLocale);
    for (int k = 0; k < styles.length; k++) {
      formatter =
DateFormat.getDateTimeInstance(
                styles[k], styles[k], currentLocale);
      result = formatter.format(today);
      System.out.println(result);
    }
  }

  static public void showDateStyles(Locale
currentLocale) {
    global(currentLocale);
    for (int k = 0; k < styles.length; k++) {
      formatter =
        DateFormat.getDateInstance(styles[k],
currentLocale);
      result = formatter.format(today);
      System.out.println(result);
    }
  }

  static public void showTimeStyles(Locale
currentLocale) {
    global(currentLocale);
    for (int k = 0; k < styles.length; k++) {
      formatter =
        DateFormat.getTimeInstance(styles[k],
currentLocale);
      result = formatter.format(today);
      System.out.println(result);
    }
  }
  static public void main(String[] args) {
     Locale[] locales = {
       new Locale("fr","FR"),
       new Locale("de","DE"),
       new Locale("es","ES"),
       new Locale("en","US")
    };
    for (int i = 0; i < locales.length; i++)
      displayDate(locales[i]);
    showDateStyles(new Locale("es","ES"));
    showDateStyles(new Locale("fr","FR"));
    showTimeStyles(new Locale("es","ES"));
    showTimeStyles(new Locale("de","DE"));
    showBothStyles(new Locale("es","ES"));
    showBothStyles(new Locale("en","US"));
  }
}
```

Patrones para fechas y horas

* Es posible utilizar **SimpleDateFormat** (que hereda de **DateFormat**) para utilizar nuestro propio formato de fecha.

* Los símbolos que se pueden utilizar son:

Symbol	Meaning	Presentation	Example
G	era designator	Text	AD
y	year	Number	1996
M	month in year	Text & Number	July & 07
d	day in month	Number	10
h	hour in am/pm (1-12)	Number	12
H	hour in day (0-23)	Number	0
m	minute in hour	Number	30
s	second in minute	Number	55
S	millisecond	Number	978
E	day in week	Text	Tuesday
D	day in year	Number	189
F	day of week in month	Number	2 (2nd Wed in July)
w	week in year	Number	27
W	week in month	Number	2
a	am/pm marker	Text	PM
k	hour in day (1-24)	Number	24
K	hour in am/pm (0-11)	Number	0
z	time zone	Text	Pacific Standard Time
'	escape for text	Delimiter	(none)
'	single quote	Literal	'

Ejemplo de formato

* El siguiente trozo de programa:

```
Date today;
String result;
SimpleDateFormat formatter;

formatter = new SimpleDateFormat("EEE d MMM yy",
                        currentLocale);
today = new Date();
result = formatter.format(today);
System.out.println("Locale: " + currentLocale.toString());
System.out.println("Result: " + result);
```

podría emitir por pantalla:

```
Locale: fr_FR
Result: ven 10 avr 98
Locale: de_DE
Result: Fr 10 Apr 98
Locale: en_US
Result: Thu 9 Apr 98
```

* La clase **DateFormatSymbol** permite realizar modificaciones sobre los símbolos utilizados a la hora de visualizar nombres de meses, de días, etc.

Formato de números

* Los números se gestionan de forma parecida a las fechas. Básicamente se pueden gestionar: Números, Valores monetarios y Porcentajes.

```
import java.util.*;
import java.text.*;
public class NumberFormatDemo {
  static public void displayNumber(Locale currentLocale) {
    Integer quantity = new Integer(123456);
    Double amount = new Double(345987.246);
    NumberFormat numberFormatter;
    String quantityOut;
    String amountOut;
    numberFormatter = NumberFormat.getNumberInstance(currentLocale);
    quantityOut = numberFormatter.format(quantity);
    amountOut = numberFormatter.format(amount);
    System.out.println(quantityOut + "   " + currentLocale.toString());
    System.out.println(amountOut + "   " + currentLocale.toString());
  }
  static public void displayCurrency(Locale currentLocale) {
    Double currency = new Double(9876543.21);
    NumberFormat currencyFormatter;
    String currencyOut;
    currencyFormatter = NumberFormat.getCurrencyInstance(currentLocale);
    currencyOut = currencyFormatter.format(currency);
    System.out.println(currencyOut + "   " + currentLocale.toString());
  }
  static public void displayPercent(Locale currentLocale) {
    Double percent = new Double(0.75);
    NumberFormat percentFormatter;
    String percentOut;
    percentFormatter = NumberFormat.getPercentInstance(currentLocale);
    percentOut = percentFormatter.format(percent);
    System.out.println(percentOut + "   " + currentLocale.toString());
  }
  static public void main(String[] args) {
    Locale[] locales = {new Locale("fr","FR"), new Locale("de","DE"),
      new Locale("en","US"), new Locale("es","ES"), new Locale("es","ES", “EURO”)
    };
    for (int i = 0; i < locales.length; i++) {
      System.out.println();
      displayNumber(locales[i]);
      displayCurrency(locales[i]);
      displayPercent(locales[i]);
    } } }
```

Patrones para números

* También es posible utilizar patrones con los números. Ej.:

```
NumberFormat nf = NumberFormat.getNumberInstance(loc);
DecimalFormat df = (DecimalFormat)nf;
df.applyPattern(pattern);
String output = df.format(value);
System.out.println(pattern + " " + output + " " + loc.toString());
```

* Los símbolos que se pueden usar en los patrones son:

Symbol	Description
0	un dígito
#	un dígito, o cero si no existe
.	Lugar para el separador decimal
,	lugar para el separador de grupo
E	separa la mantisa y el exponente en el formato exponencial
;	separa formatos
-	prefijo negativo por defecto
%	multiplica por 100 y visualiza un porcentaje
?	Multiplica por 1000 y visualiza como “por milla”
$	signo monetario se reemplaza por el símbolo monetario; si aparece duplicado se reemplaza por el símbolo monetario internacional; si aparece en un patrón se utilizará el separador decimal monetario en lugar del separador
X	se puede usar cualquier carácter como prefijo o sufijo
'	se emplea para entrecomillar caracteres especiales en un prefijo o sufijo

Ordenación

* La mayoría de idiomas existente, posee un conjunto de caracteres que extiende de alguna manera el del idioma inglés.

* En estos idiomas, es necesario suministrar una serie de reglas para que esas letras de más sean consideradas a la hora de realizar ordenaciones alfabéticas.

* Para ello se emplea la clase **Collator**, que integra la capacidad de ordenar texto mediante **Compare(...)**. He aquí un ejemplo:

```
public static void sortStrings(Collator collator, String[] words) {
        String tmp;
        for (int i = 0; i < words.length; i++) {
                for (int j = i + 1; j < words.length; j++) {
                        if (collator.compare(words[i], words[j]) > 0) {
                                tmp = words[i];
                                words[i] = words[j];
                                words[j] = tmp;
                        }
                }
        }
}
```

que puede ser llamado con los **Collator**:

```
Collator fr_FRCollator = Collator.getInstance(new Locale("fr","FR"));
Collator en_USCollator = Collator.getInstance(new Locale("en","US"));
```

Reglas de ordenación

* A través de la clase **RuleBasedCollator**, es posible crear nuestras propias reglas de ordenación.

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

public class RulesDemo  {

   public static void sortStrings(Collator collator,
String[] words) {
      String tmp;
      for (int i = 0; i < words.length; i++) {
         for (int j = i + 1; j < words.length; j++) {
            // Compare elements of the words array
            if( collator.compare(words[i], words[j] ) >
0 ) {
               // Swap words[i] and words[j]
               tmp = words[i];
               words[i] = words[j];
               words[j] = tmp;
            }
         }
      }
   }

   public static void printStrings(String [] words) {
      for (int i = 0; i < words.length; i++) {
         System.out.println(words[i]);
      }
   }

   static public void main(String[] args) {

      String englishRules =
        ("< a,A < b,B < c,C < d,D < e,E < f,F " +
         "< g,G < h,H < i,I < j,J < k,K < l,L " +
         "< m,M < n,N < o,O < p,P < q,Q < r,R " +
         "< s,S < t,T < u,U < v,V < w,W < x,X " +
         "< y,Y < z,Z");

      String smallnTilde  = new String("\u00F1");
      String capitalNTilde = new String("\u00D1");
      String traditionalSpanishRules =
        ("< a,A < b,B < c,C " +
         "< ch, cH, Ch, CH "  +
         "< d,D < e,E < f,F " +
         "< g,G < h,H < i,I < j,J < k,K < l,L " +
         "< ll, lL, Ll, LL "  +
         "< m,M < n,N " +
         "< " + smallnTilde + "," + capitalNTilde + " "
+
         "< o,O < p,P < q,Q < r,R " +
         "< s,S < t,T < u,U < v,V < w,W < x,X " +
         "< y,Y < z,Z");

      String [] words = {
        "luz",
        "curioso",
        "llama",
        "chalina"
      };

      try {
         RuleBasedCollator enCollator =
           new RuleBasedCollator(englishRules);
         RuleBasedCollator spCollator =
           new
RuleBasedCollator(traditionalSpanishRules);

         sortStrings(enCollator, words);
         printStrings(words);

         System.out.println();

         sortStrings(spCollator, words);
         printStrings(words);
      } catch (ParseException pe) {
         System.out.println("Parse exception for
rules");
      }
   }
}
```