Nuevas Tecnologías de la Programación

Módulo 3: Programación en Java en entorno UNIX

Andrés Cano Utrera
Dpto. Ciencias de la Computación e I.A
Universidad de Granada

Noviembre de 2011

Índice

1. Introducción al lenguaje Java

1.1. Origen de Java

◊ **Linaje de Java**

- Java deriva su sintaxis de C y sus características orientadas a objetos las deriva casi todas de C++.

◊ **El nacimiento de la programación moderna**

- La aparición del lenguaje C en la década de los 70 supuso para algunos el comienzo de la edad moderna de los lenguajes de programación.

  Este lenguaje fue un lenguaje potente, eficiente y estructurado que se podía aprender con cierta facilidad. C hacía uso del paradigma de la *programación estructurada*.
  Además se dice que era un lenguaje para el *programador* (fue diseñado, implementado y desarrollado por programadores reales que reflejaron su forma de entender la programación).

- A finales de los 70 y principios de los 80, C dominó el mundo de la programación y todavía hoy se usa bastante.
- C++ aparece para poder construir programas cada vez más complejos, mediante el uso de la *programación orientada a objetos*.
- A finales de los 80 y principios de los 90, el control lo tenía la PDO y C++.
- Parecía que C++ era el lenguaje perfecto: alta eficiencia y uso del paradigma *orientado a objetos*.
- El desarrollo de Internet y la WWW provocó una nueva revolución en el mundo de la programación.

◊ **Creación de Java**

- La primera versión de Java (llamada Oak) fue concebida en Sun Microsystems Inc. en 1991 (por James Gosling, Patrick Naughton, Chris Warth, Ed Frank y Mike Sheridan).
- El anuncio público de Java se hizo en 1995.

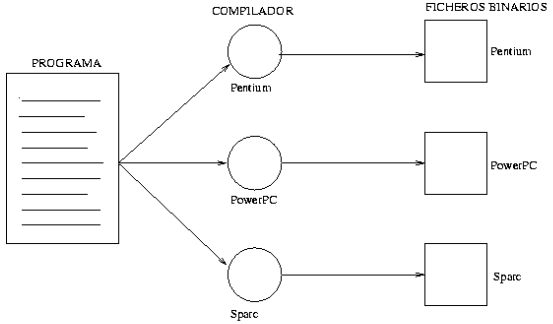
- El impulso inicial de Java no fue Internet, sino la necesidad de un **lenguaje independiente de la plataforma** para crear software para dispositivos electrónicos (microondas, controles remotos, etc), lo cual con lenguajes como C y C++ era complicado de hacer (se necesita el compilador para tal CPU).
- Mientras se elaboraban los detalles de Java, apareció un segundo y más importante factor: La **World Wide Web**. La red también exigía programas portables. La WWW hizo que Java fuese impulsado al frente del diseño de los lenguajes de programación.

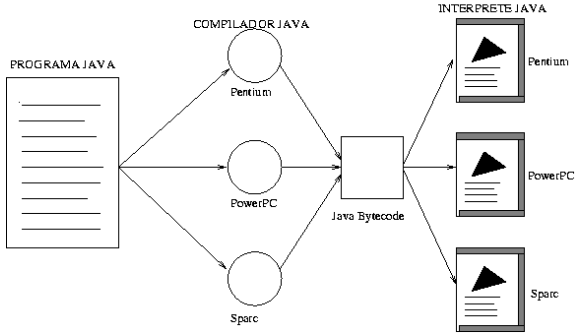
◊ **Importancia de Java para Internet**

- Internet ha ayudado a Java a situarse como líder de los lenguajes de programación y por otro lado Java ha tenido un profundo efecto sobre Internet.
  - La razón es que Java **amplía el universo de objetos** que pueden moverse libremente por el ciberespacio: programas dinámicos autoejecutables.
- Java permite crear tres tipos de programas:
  - **Aplicaciones**
  - **Applets**
  - **Servlets**
- Java controla los problemas de **seguridad** y **portabilidad** tan importantes en la red.
  - **Seguridad**:
    - Impedir infectarse con virus en computadora local.
    - Impedir el acceso a ficheros confidenciales.
    - Además Java no usa punteros evitando el acceso ilegal a memoria.
  - **Portabilidad**:
    - Código ejecutable portable a diferentes plataformas.

◊ **El código binario: bytecode**

- La salida del compilador de Java no es código ejecutable, sino un código binario intermedio (bytecode) que contiene un conjunto de instrucciones altamente optimizadas, y que luego podrán ejecutarse mediante una máquina virtual (el intérprete de Java).





- Los problemas de seguridad y portabilidad son solucionados mediante el uso de **bytecode**
  - **Portabilidad**: El bytecode al ser interpretado, sólo se requiere un intérprete para cada plataforma.
  - **Seguridad**: Por la misma razón, al ser interpretado el programa está controlado por el intérprete, evitando que provoque efectos no deseados en el sistema.

- A pesar de ser interpretado la velocidad no es mucho peor que en un lenguaje compilado.
- Además nada impide que el programa sea compilado en código nativo.

◊ **Las palabras de moda de Java**

- *Seguro*
- *Portable*
- *Simple*: Fácil de aprender y de utilizar de forma eficiente.

  Es parecido a C++, pero evita los conceptos más confusos de C++ (punteros, referencias, registros, typedef, macros, necesidad de liberar memoria), o bien los implementa de forma más clara y agradable.
- *Orientado a objetos*
- *Robusto*: Saber que los programas se comportarán de manera predecible bajo diversas condiciones. La robustez se consigue en Java con:
  - Comprueba el código en tiempo de compilación (es un lenguaje fuertemente tipado) y en tiempo de ejecución, lo que permite encontrar pronto los errores de un programa.
  - No se deja al programador la tarea de reservar y liberar memoria: Posee un **recolector automático de basura**.
  - Las situaciones excepcionales (división por 0, archivo no encontrado, etc) son manejadas con la **gestión de excepciones**.
- *Multihilo*: Permite escribir programas que hacen varias cosas a la vez.
- *Arquitectura neutral*: Independiente de la plataforma (máquina y sistema operativo).
- *Interpretado y de alto rendimiento*: El bytecode fue diseñado con cuidado para ser sencillo de traducir a código máquina nativo para conseguir un rendimiento alto.
- *Distribuido*: Diseñado para el entorno distribuido de Internet (trabaja con TCP/IP). Permite ejecutar métodos en máquinas remotas y acceder a archivos remotos (URLs).

- *Dinámico*: Java no conecta todos los módulos de una aplicación hasta el tiempo de ejecución.

1.2. Programación orientada a objetos

- La PDO es la base de Java. Todos los programas en Java son orientados a objetos.

◊ **Dos paradigmas**

- Programas orientados a proceso: Organizados conceptualmente en base al código (lo que está sucediendo).
  El código actúa sobre los datos.
- Programas orientados a objetos: Organizados en base a los datos y a un conjunto de interfaces bien definidos a esos datos.
  Los datos controlan el acceso al código.

◊ **Abstracción**

- Es un elemento esencial de la PDO: Significa ignorar los detalles de cómo funcionan los objetos, y conocer sólo cómo se usan.
- Los objetos se pueden tratar como entidades que responden a mensajes que les dicen que hagan algo.

1.2.1. Los tres principios de la programación orientada a objetos

◊ **Encapsulamiento**

Es un mecanismo que permite **juntar el código y los datos** que el código manipula en una misma entidad.

- El encapsulamiento es como un **envoltorio protector** que evita que otro código que está fuera pueda acceder arbitrariamente al código o a los datos.
- El acceso al código y a los datos se hace de forma controlada a través de una **interfaz** bien definida.

- La base del encapsulamiento es la **clase**: Define la **estructura** (datos) y **comportamiento** que serán compartidos por un conjunto de objetos.
- En la clase, los métodos y datos pueden definirse como **privados** o **públicos**.

◊ **Herencia**

Proceso mediante el cual un objeto adquiere las propiedades (datos y métodos) de otro.

- La mayor parte del conocimiento se puede organizar en clasificaciones jerárquicas.
- Usando una jerarquía de objetos, un objeto sólo necesita definir aquellas cualidades que lo hacen único dentro de su clase.
- Las subclases heredan todos los atributos de cada uno de sus antecesores en la jerarquía de clases.

◊ **Polimorfismo**

Significa que se usa un **mismo interfaz**, pero **varios métodos** distintos. Permite usar una misma interfaz para una clase general de objetos.

- Pila de números enteros, flotantes y char: En un lenguaje no orientado a objetos hay que crear tres conjuntos de rutinas diferentes y con nombres diferentes.
- En Java se puede especificar un conjunto de rutinas para las pilas con el mismo nombre para todos los tipos.
- El compilador es el que selecciona la acción específica (método) que se debe aplicar en cada situación.

1.3. Primer programa en Java

Ejemplo: P1/Example.java

```
/*
  Este es un primer programa de prueba.
  Este archivo se llama "Example.java"
*/
class Example {
  // El programa comienza con una llamada a main().
  public static void main(String args[]) {
    System.out.println("Hola mundo.");
  }
}
```

◊ **Cuestiones sobre nomenclatura**

- En Java, un fichero fuente contiene una o más definiciones de clase.
- El nombre de un fichero fuente suele ser el mismo de la clase que contenga.
- Los ficheros de programas en Java se guardarán con la extensión .java
- Debemos asegurarnos que coinciden mayúsculas y minúsculas en nombre de fichero y clase.

◊ **Compilación del programa con jdk**

- Se ejecutará la orden:

  `javac Example.java`

- El compilador **javac** crea un archivo llamado `Example.class` que contiene el bytecode compilado del programa.

◊ **Ejecución del programa con jdk**

- Se ejecutará la orden:

  `java Example`

- La salida del programa será:

  `Hola mundo`

◊ **Análisis del primer programa de prueba**

- Comentarios de varias líneas

  ```
  /*
    Este es un primer programa de prueba.
    Este archivo se llama "Example.java"
  */
  ```

- Definición de la clase

  `class Example {`

  La definición de una clase, incluyendo todos sus miembros, estará entre la llave de apertura (`{`) y la de cierre (`}`).

- Comentario de una línea

  `// El programa comienza con una llamada a main().`

- Cabecera del método main

  `public static void main(String args[]) {`

  - Todas las aplicaciones Java comienzan su ejecución llamando a main.
  - La palabra **public** es un *especificador de acceso* (puede usarse fuera de la clase en la que se declara).
  - Otro especificador de acceso es el **private** (puede usarse sólo en métodos de la misma clase).
  - La palabra **static** permite que main sea llamado sin tener que crear un objeto de esta clase (Example).
  - La palabra **void** indica que main no devuelve ningún valor.
  - El parámetro **String args[]** declara una matriz de instancias de la clase **String** que corresponde a los argumentos de la línea de ordenes.

- Siguiente línea de código:

  `System.out.println("Hola mundo.");`

  Esta línea visualiza la cadena `"Hola mundo"` en la pantalla.

2. Tipos de datos, variables y matrices

- Java es un lenguaje fuertemente tipado:
  - Cada variable tiene un tipo, cada expresión tiene un tipo y cada tipo está definido estrictamente.
  - En todas las asignaciones (directas o a través de parámetros) se comprueba la compatibilidad de los tipos.
- Java es más estricto que C en asignaciones y paso de parámetros.

2.1. Tipos simples

- No son orientados a objetos y son análogos a los de C (por razones de eficiencia).
- Todos los tipos de datos tienen un **rango definido estrictamente** a diferencia de C (por razones de portabilidad).

◊ **Enteros**

Todos los tipos son enteros con signo.

- **byte**: 8 bits. $[-128, 127]$
- **short**: 16 bits. $[-32{,}768, 32{,}767]$
- **int**: 32 bits. $[-2{,}147{,}483{,}648, 2{,}147{,}483{,}647]$
- **long**: 64 bits. $[9{,}223{,}372{,}036{,}854{,}775{,}808, 9{,}223{,}372{,}036{,}854{,}775{,}807]$

◊ **Tipos en coma flotante**

- **float**: 32 bits. $[3,4 \cdot 10^{-38}, 3,4 \cdot 10^{38}]$
- **double**: 64 bits. $[1,7 \cdot 10^{-308}, 1,7 \cdot 10^{308}]$

◊ **Caracteres**

- Java utiliza código **Unicode** para representar caracteres.
- Es un conjunto de caracteres completamente internacional con todos los caracteres de todas las lenguas del mundo.
- **char**: 16 bits ($[0, 65536]$)
- Los caracteres ASCII van del $[0, 127]$ y el ISO-Latin-1 (caracteres extendidos) del $[0, 255]$

◊ **Booleanos**

boolean: Puede tomar los valores **true** o **false**

2.2. Literales

◊ **Enteros**

- Base decimal: 1, 2, 3, etc
- Base octal: 07
- Hexadecimal: Se antepone $0x$ o $0X$.
- Long: Se añade **L**. Ej: $101L$

◊ **Coma flotante**: Son de doble precisión por defecto (double).

- Notación estándar: 2,0, 3,14, ,66
- Notación científica: $6{,}022E23$
- Añadiendo al final **F** se considera float, y añadiendo **D** double.

◊ **Booleanos**: **true** y **false**

◊ **Carácter**

- Se encierran entre comillas simples.
- Se pueden usar secuencias de escape:
  - `\141` (3 dígitos): Código en octal de la letra **a**
  - `\u0061` (4 dígitos): Código en hexadecimal (carácter Unicode) de la letra **a**
  - `\'`: Comilla simple
  - `\\`: Barra invertida
  - `\r`: Retorno de carro
  - `\t`: Tabulador `\b`: Retroceso

◊ **Cadena**: Entre comillas dobles

2.3. Variables

- La forma básica de declaración de variables es:

  ```
  tipo identificador [=valor][,identificador[=valor]...];
  ```

- `tipo` es un tipo básico, nombre de una clase o de un interfaz.
- Los inicializadores pueden ser dinámicos:

  ```
  double a=3.0, b=4.0;
  double c=Math.sqrt(a*a + b*b);
  ```

- Java permite declarar variables dentro de cualquier bloque.
- Java tiene dos **tipos de ámbito**
  - De clase:
  - De método:
- Los ámbitos se pueden anidar aunque las variables de un bloque interior no pueden tener el mismo nombre de alguna de un bloque exterior.
- Dentro de un bloque, las variables pueden declararse en cualquier momento pero sólo pueden usarse después de declararse.

2.4. Conversión de tipos

Funciona de forma parecida a C++

◊ **Conversión automática de Java**

- La **conversión automática de tipos** se hace si:
  - Los dos tipos son compatibles. Ejemplo: se puede asignar **int** a un **long**
  - El tipo destino es más grande que el tipo origen
- Los tipos **char** y **boolean** no son compatibles con el resto.

◊ **Conversión de tipos incompatibles**

Cuando queramos asignar a un tipo pequeño, otro mayor haremos uso de una conversión explícita:

```
int a;
byte b;
// ...
b = (byte) a;
```

◊ **Promoción de tipo automática en expresiones**

Además de las asignaciones, también se pueden producir ciertas conversiones automáticas de tipo en las expresiones.

- **short** y **byte** promocionan a **int** en expresiones para hacer los cálculos.
- Si un operador es **long** todo se promociona a **long**
- Si un operador es **float** todo se promociona a **float**
- Si un operador es **double** todo se promociona a **double**

Ejemplo de código con error de compilación:

```
byte b=50;
b = b*2; //Error, no se puede asignar un int a un byte
```

2.5. Vectores y matrices

Hay algunas diferencias en el funcionamiento de los vectores y matrices respecto a C y C++

2.5.1. Vectores

- **Declaración**
  `tipo nombre-vector[];`
  Esto sólo declara `nombre-vector` como vector de `tipo`, y le asigna **null**.
- **Reserva de la memoria**
  `nombre-vector=new tipo[tamaño]`
  Esto hace que se inicialicen a 0 todos los elementos.
- **Declaración y reserva al mismo tiempo**
  `int vector_int[]=new int[12];`
- **Inicialización al declarar el vector** `int vector_int[]={3,2,7}`
- El intérprete de Java comprueba siempre que no nos salimos de los índices del vector.

2.5.2. Matrices multidimensionales

- En Java las matrices son consideradas matrices de matrices.
- Declaración y reserva de memoria: `int twoD[][]=new int[4][5];`
- Podemos reservar cada fila de forma independiente:

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


2.5.3. Sintaxis alternativa para la declaración de matrices

`tipo[] nombre-matriz;`

Ejemplo:

```
int[] a2=new int[3];
char[][] twoD2=new char[3][4];
```

2.6. Punteros

- Java no permite punteros que puedan ser accedidos y/o modificados por el programador, ya que eso permitiría que los applets rompieran el cortafuegos existente entre el entorno de ejecución y el host cliente.

3. Operadores

La mayoría de los operadores de Java funcionan igual que los de C/C++, salvo algunas excepciones (los operadores a nivel de bits de desplazamiento a la derecha).

- Los tipos enteros se representan mediante codificación en *complemento a dos*: los números negativos se representan invirtiendo sus bits y sumando 1.
- Desplazamiento a la derecha: `valor >> num`
  Ejemplo 1

```
int a=35;
a = a >> 2; // ahora a contiene el valor 8
```

Si vemos las operaciones en binario:

```
00100011      35
>>2
00001000      8
```

Este operador >> rellena el nuevo bit superior con el contenido de su valor previo:

Ejemplo 2

```
11111000      -8
>>2
11111100      -4
```

- Desplazamiento a la derecha sin signo: `valor >>> num`
  Rellena el bit superior con cero.

Ejemplo

```
int a=-1;
a = a >>> 24;
```

Si vemos las operaciones en binario:

```
11111111 11111111 11111111 11111111   -1 en binario como entero
>>>24
00000000 00000000 00000000 11111111  255 en binario como entero
```

3.1. Tabla de precedencia de operadores:

```
.     []    ()
++    --    !      ~
*     /     % (se puede aplicar a flotantes)
+     -
<<    >>    >>>
<     >     <=     >=
==    !=
&
^
|
&&
||
?:
=     op=   *=   /=    %=    +=    -=    etc.)
```


4. Sentencias de control

De nuevo son prácticamente idénticas a las de C/C++ salvo en las sentencias `break` y `continue`

4.1. Sentencias de selección

- **if/else**

```
if( Boolean ) {
    sentencias;
}
else {
    sentencias;
}
```

- **switch**

```
switch( expr1 ) {
  case expr2:
       sentencias;
       break;
  case expr3:
       sentencias;
       break;
  default:
       sentencias;
       break;
  }
```

4.2. Sentencias de iteración

- **Bucles for**

```
for( expr1 inicio; expr2 test; expr3 incremento ) {
  sentencias;
}
```

- **Bucles while**

```
while( Boolean ) {
    sentencias;
}
```

- **Bucles do/while**

```
do {
    sentencias;
}while( Boolean );
```

4.3. Tratamiento de excepciones

- **Excepciones: try-catch-throw-throws-finally**

```
try {
    sentencias;
} catch( Exception ) {
    sentencias;
}
```

4.4. Sentencias de salto

Las sentencias `break` y `continue` se recomienda no usarlas, pues rompen con la filosofía de la programación estructurada:

4.4.1. break

Tiene tres usos:

- Para terminar una secuencia de sentencias en un **switch**
- Para salir de un bucle
- Como una forma de goto: `break etiqueta`

4.4.2. continue

Tiene dos usos:

- Salir anticipadamente de una iteración de un bucle (sin procesar el resto del código de esa iteración).
- Igual que antes pero se especifica una etiqueta para describir el bucle que lo engloba al que se aplica.

Ejemplo 1

```
uno: for( ) {
  dos: for( ){
         continue;     // seguiría en el bucle interno
         continue uno; // seguiría en el bucle principal
         break uno;    // se saldría del bucle principal
       }
}
```

4.4.3. return

Tiene el mismo uso de C/C++

Ejemplo 2

```
int func()
{
  if( a == 0 )
    return 1;
  return 0; // es imprescindible
}
```

5. Clases

5.1. Fundamentos

- **Clase**: La clase es la definición *general* de una entidad sobre la que estamos interesados en realizar algún tipo de procesamiento informático.

  Ejemplo: personas, coches, libros, alumnos, productos.

  La clase es la base de la PDO en Java.

- **Objeto**: Elemento *real* asociado a una clase (*instancia de una clase*).

5.1.1. Forma general de una clase

En Java, a diferencia de C++, la declaración de la clase y la implementación de los métodos se almacena en el mismo lugar:

```
class nombre_de_clase {
  tipo variable_de_instancia1;
  // ...
  tipo variable_de_instanciaN;
  tipo nombre_de_método1(lista_de_parámetros) {
    // cuerpo del método
  }
  // ...
  tipo nombre_de_métodoN(lista_de_parámetros) {
    // cuerpo del método
  }
}
```

5.1.2. Una clase sencilla

```
class Box {
  double width;
  double height;
  double depth;
}
```

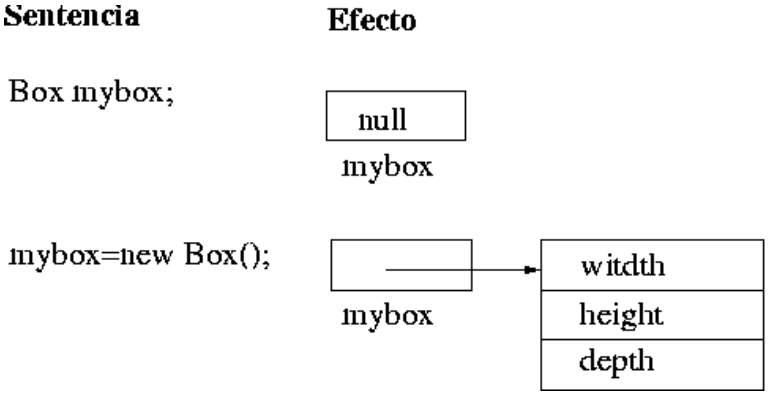

5.2. Declaración de objetos

- Para crear objetos de una clase se dan dos pasos:
  - Declarar una variable del tipo de la clase.
  - Obtener una copia física y real del objeto con el operador **new** asignándosela a la variable.

```
Box mybox=new Box();
```

o bien

```
Box mybox; // declara la referencia a un objeto
mybox=new Box(); // reserva espacio para el objeto
```



5.2.1. Operador new

- El operador **new** reserva memoria dinámicamente para un objeto.

  ```
  variable = new nombre_de_clase();
  ```

- También hace que se llame al constructor de la clase. En este caso se llama al *constructor por defecto*.

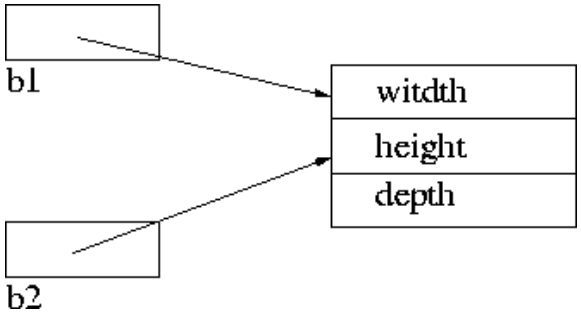
Ejemplo: P2/BoxDemo.java

```
/* Un programa que utiliza la clase Box
   Este archivo se debe llamar BoxDemo.java */
class Box {
  double width;
  double height;
  double depth;
}
// Esta clase declara un objeto del tipo Box
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;
    // asigna valores a las variables de instancia de mybox
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;
    // calcula el volumen de la caja
    vol = mybox.width * mybox.height * mybox.depth;
    System.out.println("El volumen es " + vol);
  }
}
```


5.3. Asignación de variables referencia a objeto

- Las variables referencia a objeto se comportan de manera diferente a lo que se podría esperar en asignaciones.

```
Box b1 = new Box();
Box b2 = b1;
```



5.4. Métodos

- Como dijimos anteriormente dentro de las clases podemos tener variables de instancia y métodos.
- La sintaxis para definir un método es la misma de C y C++:

```
tipo nombre_de_método(lista_de_parámetros){
  // cuerpo del método
}
```

- Dentro de los métodos podemos usar las variables de instancia directamente.
- Para llamar a un método usamos:

```
objeto.nombre_metodo(parametros);
```

Ejemplo: P3/BoxDemo4.java

```
/* Un programa que utiliza la clase Box
   Este archivo se debe llamar BoxDemo4.java
*/
class Box {
  double width;
  double height;
  double depth;
  // calcula y devuelve el volumen
  double volume() {
    return width * height * depth;
  }
}

// Esta clase declara un objeto del tipo Box
class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    // asigna valores a las variables de instancia de mybox
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```


5.4.1. Métodos con parámetros

Ejemplo: P4/BoxDemo5.java

```
/* Un programa que utiliza la clase Box */
class Box {
  double width;
  double height;
  double depth;
  // calcula y devuelve el volumen
  double volume() {
    return width * height * depth;
  }
  // establece las dimensiones de la caja
  void setDim(double w, double h, double d) {
    width = w;
    height= h;
    depth = d;
  }
}
// Esta clase declara un objeto del tipo Box
class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // inicializa cada caja
    mybox1.setDim(10,20,15);
    mybox2.setDim(3,6,9);
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.5. Constructores

- El constructor inicializa el objeto inmediatamente después de su creación.
- Tiene el mismo nombre que la clase, y no devuelve nada (ni siquiera `void`).
- Cuando no especificamos un constructor, Java crea un *constructor por defecto* , que inicializa todas las variables de instancia a cero.

Ejemplo: P5/BoxDemo6.java

```
class Box {
  double width;
  double height;
  double depth;
  Box() {
    System.out.println("Constructor de Box");
    width = 10;
    height= 10;
    depth = 10;
  }
  double volume() {
    return width * height * depth;
  }
}
class BoxDemo6 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```


5.5.1. Constructores con parámetros

Ejemplo: P6/BoxDemo7.java

```
class Box {
  double width;
  double height;
  double depth;
  Box() {
    System.out.println("Constructor de Box");
    width = 10;
    height= 10;
    depth = 10;
  }
  Box(double w,double h,double d) {
    width = w;
    height = h;
    depth = d;
  }
  double volume() {
    return width * height * depth; }
}
class BoxDemo7 {
  public static void main(String args[]) {
    Box mybox1 = new Box(10,20,15);
    Box mybox2 = new Box(3,6,9);
    Box mybox3 = new Box();
    double vol;
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox3.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.6. this

- Los métodos pueden referenciar al objeto que lo invocó con la palabra clave **this**.

  Ejemplo de su utilidad

```
Box(double width,double height,double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
  }
```

5.7. Recogida de basura

- Java libera los objetos de manera automática cuando ya no hay ninguna referencia a esos objetos.

5.7.1. Método finalize()

- Permite definir acciones específicas que se realizarán cuando el sistema de recogida de basura reclame un objeto.

  Por ejemplo, si un objeto mantiene recursos, como un descriptor de archivo, o un tipo de letra del sistema de ventanas, entonces sería conveniente que estos recursos se liberasen.

- Este método tiene el siguiente formato:

```
protected void finalize()
{
  // código de finalización
}
```


5.8. Ejemplo de clase: Clase Stack

- La clase permite el encapsulamiento de datos y código.
- La clase es como una *caja negra*: No hace falta saber qué ocurre dentro para poder utilizarla.

Ejemplo: P7/TestStack.java

```
class Stack {
  int stck[] = new int[10];
  int tos;
  Stack() {/*Inicializa la posición superior de la pila*/
    tos = -1;
  }
  void push(int item) {/*Introduce un elemento en la pila*/
    if(tos==9)
      System.out.println("La pila esta llena");
    else
      stck[++tos] = item;
  }
  int pop() {/*Extrae un elemento de la pila*/
    if(tos < 0) {
      System.out.println("La pila esta vacía");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();
    // introduce algunos números en la pila
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);
    // extrae los números de la pila
    System.out.println("Contenido de la pila mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());
    System.out.println("Contenido de la pila mystack2:");
    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());
  }
}
```


6. Métodos y clases

6.1. Sobrecarga de métodos

Consiste en que dos o más métodos de una clase tienen el mismo nombre, pero con listas de parámetros distintos.

- La sobrecarga es usada en Java para implementar el *polimorfismo*.
- Java utiliza el tipo y/o el número de argumentos como guía para ver a cual método llamar.
- El tipo que devuelve un método es insuficiente para distinguir dos versiones de un método.

Ejemplo: P8/Overload.java

```
class OverloadDemo {
  void test() {
    System.out.println("Sin parametros");
  }

  // Sobrecarga el metodo test con un parámetro entero
 void test(int a) {
    System.out.println("a: " + a);
  }

  // Sobrecarga el metodo test con dos parametros enteros
  void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  // Sobrecarga el metodo test con un parámetro double
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // llama a todas las versiones de test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.2);
    System.out.println("Resultado de ob.test(123.2): " + result);
  }
}
```

La salida del programa sería:

```
Sin parametros
a: 10
a y b: 10 20
a double: 123.2
Resultado de ob.test(123.2): 15178.2
```


6.1.1. Sobrecarga con conversión automática de tipo

Java busca una versión del método cuyos parámetros actuales coincidan con los parámetros formales, en número y tipo. Si el tipo no es exacto puede que se aplique *conversión automática de tipos*.

```
class OverloadDemo {
  void test() {
    System.out.println("Sin parametros");
  }

  // Sobrecarga el metodo test con dos parametros enteros
 void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  // Sobrecarga el metodo test con un parámetro double
  void test(double a) {
    System.out.println("Dentro de test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // esto llama a test(double)
    ob.test(123.2); // esto llama a test(double)
  }
}
```

6.1.2. Sobrecarga de constructores

Además de los métodos normales, también pueden sobrecargarse los constructores.

Ejemplo: P9/OverloadCons.java

```
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  Box() {
    width = -1;  // utiliza -1 para indicar
    height = -1; // que una caja no está
    depth = -1;  // inicializada
  }
  Box(double len) {
    width = height = depth = len;
  }

  double volume() {
    return width * height * depth;
  }
}
```



```
class OverloadCons {
  public static void main(String args[]) {
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;

    vol = mybox1.volume();
    System.out.println("El volumen de mybox1 es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen de mybox2 es " + vol);
    vol = mycube.volume();
    System.out.println("El volumen de mycube es " + vol);
  }
}
```

6.2. Objetos como parámetros

Un objeto de una determinada clase puede ser parámetro de un método de esa u otra clase.

Ejemplo: P10/PassOb.java

```
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}
class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);
    System.out.println("ob1 == ob2: " + ob1.equals(ob2));
    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

Uno de los ejemplos más usuales es en constructores para hacer copias de objetos:

```
Box(Box ob) {
  width = ob.width;
  height = ob.height;
  depth = ob.depth;
}
Box mybox = new Box(10,20,15);
Box myclone = new Box(mybox);
```


6.3. Paso de argumentos

- El paso de tipos simples a los métodos es siempre *por valor*.
- Los objetos se pasan siempre *por referencia*.

Ejemplo: P11/CallByValue.java

```
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a y b antes de la llamada:"+a+" "+b);
    ob.meth(a, b);
    System.out.println("a y b despues de la llamada:"+a+" "+b);
  }
}
```

Ejemplo: P12/CallByRef.java

```
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // pasa un objecto
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}
```

```
class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a y ob.b antes de la llamada: " +
                       ob.a + " " + ob.b);
    ob.meth(ob);
    System.out.println("ob.a y ob.b después de la llamada: " +
                       ob.a + " " + ob.b);
  }
}
```

6.4. Control de acceso

Los especificadores de acceso para variables de instancia y métodos son:

- **private**: Sólo es accesible por miembros de la misma clase.
- **public**: Accesible por miembros de cualquier clase.
- **protected**: Está relacionado con la herencia.
- Por defecto: si no se indica nada los miembros son públicos dentro de su mismo paquete

Ejemplo de control de acceso: P13/AccessTest.java

```
class Test {
  int a; // acceso por defecto
  public int b; // acceso publico
  private int c; // acceso privado
  void setc(int i) { // establece el valor de c
    c = i;
  }
  int getc() { // obtiene el valor de c
    return c;
  }
}
```



```
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // Esto es correcto, a y b pueden ser accedidas directamente
    ob.a = 10;
    ob.b = 20;

    // Esto no es correcto y generara un error de compilación
//  ob.c = 100; // Error!

    // Se debe acceder a c a través de sus métodos
    ob.setc(100); // OK

    System.out.println("a, b, y c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

6.5. Especificador static

- Un miembro **static** puede ser usado sin crear ningún objeto de su clase.
- Puede aplicarse a variables de instancia y métodos
  - Las variables **static** son variables globales. Existe una sola copia de la variable para todos los objetos de su clase.
  - Los métodos **static**
    - Sólo pueden llamar a métodos **static**
    - Sólo deben acceder a datos **static**
    - No pueden usar **this** o **super**
- Una clase puede tener un bloque static que se ejecuta una sola vez cuando la clase se carga por primera vez.

Ejemplo: P14/UseStatic.java

```
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Bloque estático inicializado.");
    b = a * 4;
  }
  public static void main(String args[]) {
    meth(42);
  }
}
```

Salida del programa:

```
Bloque estático inicializado.
x = 42
a = 3
b = 12
```


6.6. Especificador final con datos

Una variable **final** es una variable a la que no podemos modificar su contenido.

- Puede ser una constante en tiempo de compilación que nunca cambiará, o bien, puede asignarsele un valor en tiempo de ejecución que no se cambiará.
- Son similares al **const** de C/C++
- Suele utilizar identificadores en mayúscula.
- Las variables final suelen ser también static (existe una sola copia de ellas para todos los objetos de la clase).
- **final** se puede aplicar a:
  - Datos primitivos: Significa que el valor no se cambiará.
  - Referencias a objetos: Significa que la referencia no se cambiará, aunque si podemos cambiar el contenido del objeto.
  - Parámetros de un método.

Ejemplo de contantes en tiempo de compilación

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Ejemplo con datos primitivos y referencias: P65/FinalData.java

```
class Value {
  int i = 1;
}
public class FinalData {
  final int i1 = 9;
  static final int VAL_TWO = 99;
  public static final int VAL_THREE = 39;
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();

  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
```



```
  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: no se puede cambiar el valor
    fd1.v2.i++; // El objeto no es constante!
    fd1.v1 = new Value(); // OK -- no es final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object no es constante!
    //! fd1.v2 = new Value(); // Error: No se puede
    //! fd1.v3 = new Value(); // ahora cambiar la referencia
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
}
```

Salida del programa

```
fd1: i4 = 1, i5 = 2
Creating new FinalData
fd1: i4 = 1, i5 = 2
fd2: i4 = 16, i5 = 2
```

Ejemplo con parámetros final en un método: P66/FinalArguments.java

```
class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Ilegal -- g es final
    g.spin();
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g no es final
    g.spin();
  }
  // void f(final int i) { i++; } // No se puede cambiar
  // Sólo se puede leer de un tipo de dato primitivo:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    Gizmo g = new Gizmo();
    bf.without(g);
    bf.with(g);
  }
}
```


6.6.1. Constantes blancas

Son campos declarados como **final** pero a los que no se da un valor de inicialización en la declaración, pero tendrán que inicializarse en el constructor.

Ejemplo con parámetros final en un método: P67/BlankFinal.java

```
class Poppet { }

class BlankFinal {
  final int i = 0; // Constante inicializada
  final int j; // Constante blanca
  final Poppet p; // Referencia constante blanca
  // Las constantes BLANCAS deben inicializarse
  // en el constructor:
  BlankFinal() {
    j = 1; // Inicializar la constante blanca
    p = new Poppet();
  }
  BlankFinal(int x) {
    j = x; // Initializar la constante blanca
    p = new Poppet();
  }
  public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

6.7. Clase String

Es una clase muy usada, que sirve para almacenar cadenas de caracteres (incluso los literales).

- Los objetos **String** no pueden modificar su contenido.
- Los objetos **StringBuffer** si que pueden modificarse.
- El operador + permite concatenar cadenas.
- El método **boolean equals(String objeto)** compara si dos cadenas son iguales.
- El método **int length()** obtiene la longitud de una cadena.
- El método **char charAt(int pos)** obtiene el carácter que hay en la posición **pos**.

Ejemplo: P15/StringDemo2.java

```
// Muestra la utilización de algunos metodo de la clase String
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "Primera cadena";
    String strOb2 = "Segunda cadena";
    String strOb3 = strOb1;
    System.out.println("La longitud de strOb1 es: " +
                       strOb1.length());
    System.out.println("El carácter de la posición 3 de strOb1 es: " +
                       strOb1.charAt(3));
    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");
    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```


6.8. Argumentos de la línea de órdenes

Ejemplo: P16/CommandLine.java

```
// Presenta todos los argumentos de la línea de órdenes
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

Si ejecutamos este programa con:

```
java CommandLine esto es una prueba 100 -1
```

obtendremos como salida:

```
args[0]: esto
args[1]: es
args[2]: una
args[3]: prueba
args[4]: 100
args[5]: -1
```

6.9. Clases internas

Es posible definir una clase o interfaz dentro de otra clase o interfaz.

- La clase interior se denomina *clase interna*
- La clase exterior se denomina *clase contenedora o externa*
- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.

    No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**
- Pueden declararse como públicas, protegidas, de paquete o privadas.

6.9.1. Clases internas estáticas: nested classes

Se definen anteponiendo el modificador **static** en la definición de la clase.

- No se necesita un objeto de la clase externa para crear un objeto de la clase interna.
- No podemos acceder directamente a los campos no estáticos de la clase externa desde los métodos de la clase interna.
- Pueden definirse también dentro de un interfaz, en cuyo caso son automáticamente **public** y **static**.
- Pueden contener campos y métodos **static**, y otras clases internas **static** (a diferencia de las clases internas no estáticas).
- El nombre completo de una clase interna es: *paquete.claseExterna.claseInterna*

Clase interna estática: P120/Parcel11.java

```
1  public interface Contents {
2    int value();
3  } ///:~
4  public interface Destination {
5    String readLabel();
6  } ///:~
7  public class Parcel11 {
8    private static class ParcelContents implements Contents {
9      private int i = 11;
10     public int value() { return i; }
11   }
12   protected static class ParcelDestination
13   implements Destination {
14     private String label;
15     private ParcelDestination(String whereTo) {
16       label = whereTo;
17     }
18     public String readLabel() { return label; }
19     // Nested classes can contain other static elements:
20     public static void f() {}
21     static int x = 10;
22     static class AnotherLevel {
23       public static void f() {}
24       static int x = 10;
25     }
26   }
27   public static Destination destination(String s) {
28     return new ParcelDestination(s);
29   }
30   public static Contents contents() {
31     return new ParcelContents();
32   }
33   public static void main(String[] args) {
34     Contents c = contents();
35     Destination d = destination("Tasmania");
36   }
37 } ///:~
```

Clase interna en un interfaz: P121/ClassInInterface.java

```
1  public interface ClassInInterface {
2    void howdy();
3    class Test implements ClassInInterface {
4      public void howdy() {
5        System.out.println("Howdy!");
6      }
7      public static void main(String[] args) {
8        new Test().howdy();
9      }
10   }
11 }
```

Clase interna para probar la clase externa: P122/TestBed.java

```
1  public class TestBed {
2    public void f() { System.out.println("f()"); }
3    public static class Tester {
4      public static void main(String[] args) {
5        TestBed t = new TestBed();
6        t.f();
7      }
8    }
9  }
```

Otro ejemplo de clase interna estática: P123/Graph1.java

```
1  import java.util.Hashtable;
2  import java.util.Enumeration;
3  /** Class representing an undirected graph composed
4    * of nodes.  The node class is a top-level class
5    * nested within the Graph1 class.
6    */
7  public class Graph1 {
8     private Hashtable nodeList = new Hashtable();
9
10    public void addNode( int x, int y ) {
11       Node n = new Node( x, y );
12       if ( ! nodeList.containsKey( n.key() ) ) {
13          nodeList.put( n.key(), n );
14       }
15    }
```



```
16
17    public String toString() {
18       StringBuffer sb = new StringBuffer( "[ " );
19       Enumeration e = nodeList.elements();
20       while ( e.hasMoreElements() ) {
21          sb.append( e.nextElement().toString()
22                     + " " );
23       }
24       sb.append( "]" );
25       return sb.toString();
26    }
27
28    public static void main( String[] args ) {
29       System.out.println( "creating the graph" );
30       Graph1 g = new Graph1();
31       System.out.println( "adding nodes" );
32       g.addNode( 4, 5 );
33       g.addNode( -6, 11 );
34       System.out.println( g );
35    }
36
37    /** The class representing nodes within the graph */
38    private static class Node {
39       private int x, y;
40
41       public Node( int x, int y ) {
42          this.x = x;
43          this.y = y;
44        }
45
46       public Object key() {
47          return x + "," + y;
48       }
49
50       public String toString() {
51          return "(" + x + "," + y + ")";
52       }
53    } // end of Node class
54 }
```

6.9.2. Clases internas no estáticas: inner classes

Los métodos de una clase interna no estática tienen acceso a todas las variables y métodos de la clase externa de la misma forma que cualquier otro método no estático de la clase externa.

Ejemplo de clase interna no estática: P60/InnerClassDemo.java

```
1  class Outer {
2    int outer_x = 100;
3    void test() {
4      Inner inner = new Inner();
5      inner.display();
6    }
7    class Inner {
8      void display() {
9        System.out.println("display: outer_x = " + outer_x);
10     }
11   }
12 }
13
14 class InnerClassDemo {
15   public static void main(String args[]) {
16     Outer outer = new Outer();
17     outer.test();
18   }
19 }
```

Salida del programa

```
display: outer_x = 100
```


- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.

Otro ejemplo (Implementación del patrón iterador): P126/Sequence.java

```
1  interface Selector {
2    boolean end();
3    Object current();
4    void next();
5  }
6
7  public class Sequence {
8    private Object[] items;
9    private int next = 0;
10   public Sequence(int size) { items = new Object[size]; }
11   public void add(Object x) {
12     if(next < items.length)
13       items[next++] = x;
14   }
15   private class SequenceSelector implements Selector {
16     private int i = 0;
17     public boolean end() { return i == items.length; }
18     public Object current() { return items[i]; }
19     public void next() { if(i < items.length) i++; }
20   }
21   public Selector selector() {
22     return new SequenceSelector();
23   }
24   public static void main(String[] args) {
25     Sequence sequence = new Sequence(10);
26     for(int i = 0; i < 10; i++)
27       sequence.add(Integer.toString(i));
28     Selector selector = sequence.selector();
29     while(!selector.end()) {
30       System.out.print(selector.current() + " ");
31       selector.next();
32     }
33   }
34 }
```

- Un objeto de una clase interna no puede crearse sin un objeto de la clase externa.
- Para crear un objeto de la clase interna usaremos:

  `objetoClaseExterna.new claseInterna(argumentos)`

Creación de un objeto de la clase interna con .new: P124/DotNew.java

```
1  public class DotNew {
2    public class Inner {}
3    public static void main(String[] args) {
4      DotNew dn = new DotNew();
5      DotNew.Inner dni = dn.new Inner();
6    }
7  }
```

- En métodos no estáticos de la clase externa, no es necesario poner `objetoClaseExterna`.
- Para referenciar el objeto de la clase externa usamos `ClaseExterna.this`

Referenciando el objeto de la clase externa con .this: P125/DotThis.java

```
1  public class DotThis {
2    void f() { System.out.println("DotThis.f()"); }
3    public class Inner {
4      public DotThis outer() {
5        return DotThis.this;
6        // A plain "this" would be Inner's "this"
7      }
8    }
9    public Inner inner() { return new Inner(); }
10   public static void main(String[] args) {
11     DotThis dt = new DotThis();
12     DotThis.Inner dti = dt.inner();
13     dti.outer().f();
14   }
15 }
```


Referenciando el objeto de la clase externa con .this: P127/Graph2.java

```
1  import java.util.Hashtable;
2  import java.util.Enumeration;
3
4  /** Class representing an undirected graph composed
5    * of nodes.  The node class is a top-level class
6    * nested within the Graph2 class.
7    */
8  public class Graph2 {
9     private Hashtable nodeList = new Hashtable();
10
11    public void addNode( int x, int y ) {
12      // the use of "this." is not required here
13      this.new Node( x, y );
14    }
15
16    public String toString() {
17       StringBuffer sb = new StringBuffer( "[ " );
18       Enumeration e = nodeList.elements();
19       while ( e.hasMoreElements() ) {
20          sb.append( e.nextElement().toString()
21                     + " " );
22       }
23       sb.append( "]" );
24       return sb.toString();
25    }
26
27    public static void main( String[] args ) {
28       System.out.println( "creating the graph" );
29       Graph2 g = new Graph2();
30       System.out.println( "adding nodes" );
31       g.addNode( 4, 5 );
32       g.addNode( -6, 11 );
33       System.out.println( g );
34    }
35
36    private class Node {
37       private int x, y;
38       public Node( int x, int y ) {
39          this.x = x;
40          this.y = y;
```

```
41          // the use of "Graph2.this." is not
42          // required here
43          if ( ! Graph2.this.nodeList
44                       .containsKey( key() ) ) {
45             nodeList.put( key(), this );
46          }
47       }
48
49       public Object key() {
50          return x + "," + y;
51       }
52
53       public String toString() {
54          return "(" + x + "," + y + ")";
55       }
56    } // end of Node class
57 }
```

- Puede haber más de un objeto de la clase interna compartiendo el mismo objeto de la clase externa.
- Los objetos de la clase interna tienen acceso a los miembros privados de la clase externa.
- No es posible cambiar el objeto de la clase externa de un objeto de la clase interna.
- Las clases internas no estáticas no pueden contener miembros **static**.

- Es posible definir **subclases** de una clase interna, aunque es necesario el uso de una forma especial de la sentencia `super()`:

Uso de la forma especial de super: P131/InheritInner.java

```
1  class WithInner {
2    class Inner {}
3  }
4
5  public class InheritInner extends WithInner.Inner {
6    //! InheritInner() {} // Won't compile
7    InheritInner(WithInner wi) {
8      wi.super();
9    }
10   public static void main(String[] args) {
11     WithInner wi = new WithInner();
12     InheritInner ii = new InheritInner(wi);
13   }
14 } ///:~
```

Ejemplo de clase interna para manejo de eventos de un applet: P63/InnerClassDemo.java

```
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>

*/
public class InnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Mouse Pressed");
    }
  }
}
```


6.9.3. Clases internas locales

Las clases internas pueden aparecer definidas dentro de cualquier bloque de código.

- No pueden declararse como **public**, **private** o **protected**.
- Sólo pueden usarse en el método en que se definen.

Ejemplo de clase interna dentro de un for: P62/InnerClassDemo.java

```
1  class Outer {
2    int outer_x = 100;
3    void test() {
4      for(int i=0; i<5; i++) {
5        class Inner {
6          void display() {
7            System.out.println("display: outer_x = " + outer_x);
8          }
9        }
10       Inner inner = new Inner();
11       inner.display();
12     }
13   }
14 }
15 class InnerClassDemo {
16   public static void main(String args[]) {
17     Outer outer = new Outer();
18     outer.test();
19   }
20 }
```

Salida del programa

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

- Los objetos de una clase interna local pueden tener una vida más allá del ámbito en que se declaró la clase interna.

Ejemplo: P128/Parcel5.java

```
public interface Destination {
  String readLabel();
} ///:~


public class Parcel5 {
  public Destination destination(String s) {
    class PDestination implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    Destination d = p.destination("Tasmania");
  }
}
```


- Los métodos de las clases internas locales tienen acceso a sus propios campos, las variables locales **final**, los argumentos del método **final** y los campos del objeto de la clase externa en que esté incluida.

Ejemplo: P128/Parcel5.java

```
public class Equation1 {
   private int equationInput;

   public interface Result {
      public double getAnswer();
   }

   public Equation1( int ei ) {
      equationInput = ei;
   }

   public Result getResult( final int input1,
                            final int input2 ) {
      final int [] localVar = { 2,6,10,14};
      class MyResult implements Result {
         private int normalField;

         public MyResult() {
            normalField = 2;
         }
         public double getAnswer() {
           return (double) input1 / input2
                           - equationInput
                           + localVar[2]
                           - normalField;
         }
      }
      return new MyResult();
   }

   public static void main( String[] args ) {
      Equation1 e = new Equation1( 10 );
      Result r = e.getResult( 33, 5 );
      System.out.println( r.getAnswer() );
   }
}
```

6.9.4. Clases internas locales anónimas

Una clase interna anónima es aquella que no tiene asignado un nombre.

- Se crean usando la siguiente sintaxis:

  `new [clase_o_interfaz()]{cuerpo de la clase}`

- No pueden tener constructores.
- El nombre opcional `clase_o_interfaz` es el nombre de una clase que se extiende o un interfaz que se implementa. Si se omite, la clase anónima extiende **Object**.
- Si incluimos parámetros en la sentencia `new`, éstos serán pasados al constructor de la superclase.

Ejemplo de clase interna anónima para manejo de eventos de un applet: P64/InnerClassDemo.java

```
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=100>
  </applet>
*/

public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Mouse Pressed");
      }
    });
  }
}
```


- Para inicializar los objetos de la clase anónima podemos usar ***inicializadores de instancia***. Éstos pueden usarse no sólo en clases anónimas, sino en cualquier clase.
- Los inicializadores de instancia se ejecutan antes del constructor y después del constructor de la superclase.

Ejemplo de inicializador de instancia: P130/Parcel10.java

```
1  public class Parcel10 {
2    public Destination
3    destination(final String dest, final float price) {
4      return new Destination() {
5        private int cost;
6        // Instance initialization for each object:
7        {
8          cost = Math.round(price);
9          if(cost > 100)
10           System.out.println("Over budget!");
11       }
12       private String label = dest;
13       public String readLabel() { return label; }
14     };
15   }
16   public static void main(String[] args) {
17     Parcel10 p = new Parcel10();
18     Destination d = p.destination("Tasmania", 101.395F);
19   }
20 }
```

7. Herencia

La herencia es uno de los pilares de la PDO ya que permite la creación de clasificaciones jerárquicas.

- Permite definir una clase general que define características comunes a un conjunto de elementos relacionados.
- Cada subclase puede añadir aquellas cosas particulares a ella.
- Las subclases heredan todas las variables de instancia y los métodos definidos por la superclase, y luego pueden añadir sus propios elementos.

7.1. Fundamentos

- Para definir que una subclase hereda de otra clase usamos **extends**.
- Java no permite la herencia múltiple.
- La subclase no puede acceder a aquellos miembros declarados como **private** en la superclase.

Ejemplo: P17/DemoBoxWeight.java

```
// Este programa utiliza la herencia para extender la clase Box.
class Box {
  double width;
  double height;
  double depth;
  Box(Box ob) {
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```



```
  Box() {
    width = -1;
    height = -1;
    depth = -1;
  }
  Box(double len) {
    width = height = depth = len;
  }
  double volume() {
    return width * height * depth;
  }
}
class BoxWeight extends Box {
  double weight; // peso de la caja
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;
    vol = mybox1.volume();
    System.out.println("Volumen de mybox1 es " + vol);
    System.out.println("Peso de mybox1 es " + mybox1.weight);
    System.out.println();
    vol = mybox2.volume();
    System.out.println("Volumen de mybox2 es " + vol);
    System.out.println("Peso de mybox2 es " + mybox2.weight);
  }
}
```

7.1.1. Una variable de la superclase puede referenciar a un objeto de la subclase

El tipo de la variable referencia, (y no el del objeto al que apunta) es el que determina los miembros que son accesibles.

Ejemplo: P18/RefDemo.java

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("El volumen de weightbox ex " + vol);
    System.out.println("El peso de weightbox es " +
                       weightbox.weight);
    System.out.println();

    // asigna una referencia de BoxWeight a una referencia de Box
    plainbox = weightbox;
    vol = plainbox.volume(); // OK,  volume() definido en Box
    System.out.println("Volumen de plainbox es " + vol);

    /* La siguiente sentencia no es válida porque plainbox
       no define un miembro llamado weight. */
//  System.out.println("El peso de plainbox es " +
                     //  plainbox.weight);
  }
}
```


7.2. Uso de super

La palabra reservada **super** permite a una subclase referenciar a su superclase inmediata. Es utilizada en las siguientes situaciones:

1. Para llamar al constructor de la superclase desde el constructor de la subclase.

   En este caso `super()` debe ser la primera sentencia ejecutada dentro del constructor.

2. Para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

Ejemplo de uso en caso 1

```
class BoxWeight extends Box {
  double weight;

  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // llama al constructor de la superclase
    weight = m;
  }
  BoxWeight(BoxWeight ob) {
    super(ob);
    weight = ob.weight;
  }
}
```

Ejemplo de uso en caso 2: P19/UseSuper.java

```
// Utilización de super para evitar la ocultación de nombres
class A {
  int i;
}

class B extends A {
  int i; // esta i oculta la i de A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i en la superclase: " + super.i);
    System.out.println("i en la subclase: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```


7.3. Orden de ejecución de constructores

Los constructores de una jerarquía de clases se ejecutan en el orden de derivación.

Ejemplo: P20/CallingCons.java

```
// Muestra cuando se ejecutan los constructores.
class A {
  A() {
    System.out.println("En el constructor de A.");
  }
}
class B extends A {
  B() {
    System.out.println("En el constructor de B.");
  }
}
class C extends B {
  C() {
    System.out.println("En el constructor de C.");
  }
}
class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

La salida del programa es:

```
En el constructor de A.
En el constructor de B.
En el constructor de C.
```

7.4. Sobreescritura de métodos (Overriding)

Consiste en construir un método en una subclase con el mismo nombre, parámetros y tipo que otro método de una de sus superclases (inmediata o no).

Ejemplo: P21/Override.java

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  void show() {
    System.out.println("i y j: " + i + " " + j);
  }
}
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  // muestra k -- sobreescribe el metodo show() de A
  void show() {
    System.out.println("k: " + k);
  }
}
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show(); // llama al metodo show() de B
  }
}
```


7.5. Selección de método dinámica

Es el mecanismo mediante el cual una llamada a una función sobreescrita se resuelve en **tiempo de ejecución** en lugar de en tiempo de compilación: **polimorfismo en tiempo de ejecución**

> Cuando un método sobreescrito se llama a través de una referencia de la superclase, Java determina la versión del método que debe ejecutar en función del objeto que está siendo referenciado.

Ejemplo: P22/Dispatch.java

```
class A {
   void callme() {
     System.out.println("Llama al metodo callme dentro de A");
  }
}

class B extends A {
  void callme() {
    System.out.println("Llama al metodo callme dentro de B");
  }
}

class C extends A {
  void callme() {
    System.out.println("Llama al metodo callme dentro de C");
  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // objeto de tipo A
    B b = new B(); // objeto de tipo B
    C c = new C(); // objeto de tipo C
    A r; // obtiene una referencia del tipo A
    r = a; // r hace referencia a un objeto A
    r.callme(); // llama al metodo callme() de A
    r = b; // r hace referencia a un objeto B
    r.callme(); // llama al metodo callme() de B
    r = c; // r hace referencia a un objeto C
    r.callme(); // llama al metodo callme() de C
  }
}
```

7.5.1. Aplicación de sobreescritura de métodos

Ejemplo: P23/FindAreas.java

```
class Figure {
  double dim1;
  double dim2;
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  double area() {
    System.out.println("Area para Figure es indefinida.");
    return 0;
  }
}
```



```
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro de Area para Rectangle.");
    return dim1 * dim2;
  }
}
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro de Area para Triangle.");
    return dim1 * dim2 / 2;
  }
}
class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;
    figref = r;
    System.out.println("Area es " + figref.area());
    figref = t;
    System.out.println("Area es " + figref.area());
    figref = f;
    System.out.println("Area es " + figref.area());
  }
}
```

7.6. Clases abstractas

Permiten definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de sus métodos.

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por las subclases.
- Los métodos abstractos tienen la forma:

  `abstract tipo nombre(parámetros);`
- Cualquier clase con uno o más métodos abstractos debe declararse como **abstract**.
- No se pueden crear objetos de clases abstractas (usando **new**).
- No se pueden crear constructores **abstract** o métodos **static abstract**.
- Sí que podemos declarar variables referencia de una clase abstracta.

Ejemplo: P24/AbstractAreas.java

```
abstract class Figure {
  double dim1, dim2;
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  abstract double area();
}
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro del metodo area para un Rectangle.");
    return dim1 * dim2;
  }
}
```



```
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro del metodo area para un Triangle.");
    return dim1 * dim2 / 2;
  }
}
class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // Esto no es correcto
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // esto es CORRECTO, no se crea ningún objeto
    figref = r;
    System.out.println("Area es " + figref.area());
    figref = t;
    System.out.println("Area es " + figref.area());
  }
}
```

7.7. Utilización de final con la herencia

La palabra reservada **final** tiene tres usos:

1. Para creación de constantes con nombre.
2. Para evitar sobreescritura de métodos

   Los métodos declarados como **final** no pueden ser sobreescritos
3. Para evitar la herencia

   Se usa **final** en la declaración de la clase para evitar que la clase sea heredada: O sea, todos sus métodos serán **final** implícitamente.

Ejemplo para evitar sobreescritura de métodos

```
class A {
  final void meth() {
    System.out.println("Este es un metodo final.");
  }
}

class B extends A {
  void meth() { // ERROR! No se puede sobreescribir.
    System.out.println("No es correcto!");
  }
}
```

Ejemplo para evitar la herencia

```
final class A {
  // ...
}

// La clase siguiente no es válida.
class B extends A { // ERROR! No se puede crear una subclase de A
  // ...
}
```


8. Paquetes e Interfaces

8.1. Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos.

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo: Las clases del paquete **MiPaquete** (ficheros `.class` y `.java`) se almacenarán en directorio **MiPaquete**
- Se importan explícitamente en las definiciones de nuevas clases con la sentencia

  `import nombre-paquete;`
- Permiten restringir la visibilidad de las clases que contiene:

  Se pueden definir clases en un paquete sin que el mundo exterior sepa que están allí.
- Se pueden definir miembros de una clase, que sean sólo accesibles por miembros del mismo paquete

8.1.1. Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

`package nombre-paquete;`

- Todas las clases de ese archivo serán de ese paquete
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*
- Una misma sentencia `package` puede incluirse en varios archivos fuente
- Se puede crear una jerarquía de paquetes:
  `package paq1[.paq2[.paq3]];`
  Ejemplo: `java.awt.image`

8.1.2. La variable de entorno CLASSPATH

Supongamos que construimos la clase **PaquetePrueba** perteneciente al paquete **prueba**, dentro del fichero `PaquetePrueba.java` (directorio `prueba`)

- **Compilación**: Situarse en directorio `prueba` y ejecutar `javac PaquetePrueba.java` o bien situarse en directorio padre de `prueba` y ejecutar `javac prueba/PaquetePrueba.java`
- **Ejecución**: Situarse en directorio padre de **prueba** y ejecutar `java prueba.PaquetePrueba` o bien añadir directorio padre de **prueba** a **CLASSPATH** y ejecutar `java prueba.PaquetePrueba` desde cualquier directorio.

8.1.3. Ejemplo de paquete: P25/MyPack

```
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }
  void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": $" + bal);
  }
}
```



```
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);
    for(int i=0; i<3; i++) current[i].show();
  }
}
```

8.2. Protección de acceso

Las clases y los paquetes son dos medios de encapsular y contener el espacio de nombres y el ámbito de las variables y métodos.

- **Paquetes**: Actúan como recipientes de clases y otros paquetes subordinados.
- **Clases**: Actúan como recipientes de datos y código.

8.2.1. Tipos de acceso a miembros de una clase

Desde método en ...	private	sin modif.	protected	public
misma clase	sí	sí	sí	sí
subclase del mismo paquete	no	sí	sí	sí
no subclase del mismo paquete	no	sí	sí	sí
subclase de diferente paquete	no	no	sí	sí
no subclase de diferente paquete	no	no	no	sí

8.2.2. Tipos de acceso para una clase

- **Acceso por defecto**: Accesible sólo por código del mismo paquete
- **Acceso public**: Accesible por cualquier código

Ejemplo: P26

```
package p1;
public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;
  public Protection() {
    System.out.println("constructor base ");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class Derived extends Protection {
  Derived() {
    System.out.println("constructor de Derived");
    System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("constructor de SamePackage");
    System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```



```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("constructor de Protection2");
//  System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");
//  System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
//  System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

8.3. Importar paquetes

Todas las clases estándar de Java están almacenadas en algún paquete con nombre. Para usar una de esas clases debemos usar su *nombre completo*. Por ejemplo para la clase **Date** usaríamos **java.util.Date**. Otra posibilidad es usar la sentencia **import**.

- **import**: Permite que se puedan ver ciertas clases o paquetes enteros sin tener que introducir la estructura de paquetes en que están incluidos, separados por puntos.
- Debe ir tras la sentencia **package**:

  `import paquete[.paquete2].(nombre_clase|*);`
- Al usar `*` especificamos que se importa el paquete completo. Esto incrementa el tiempo de compilación, pero no el de ejecución.
- Las clases estándar de Java están dentro del paquete **java**.
- Las funciones básicas del lenguaje se almacenan en el paquete. **java.lang**, el cual es importado por defecto.
- Al importar un paquete, sólo están disponibles los elementos públicos de ese paquete para clases que no sean subclases del código importado.

Ejemplo: P27

```
package MyPack;
/* La clase Balance, su constructor, y su metodo show()
   deben ser públicos. Esto significa que pueden ser utilizados por
   código que no sea una subclase y este fuera de su paquete.
*/
public class Balance {
  String name;
  double bal;
  public Balance(String n, double b) {
    name = n;
    bal = b;
  }
```



```
  public void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": $" + bal);
  }
}
```

```
import MyPack.*;
class TestBalance {
  public static void main(String args[]) {
    /* Como Balance es pública, se puede utilizar la
       clase Balance y llamar a su constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);
    test.show(); // también se puede llamar al metodo show()
  }
}
```

8.4. Interfaces

Son sintácticamente como las clases, pero no tienen variables de instancia y los métodos declarados no contienen cuerpo.

- Se utilizan para especificar lo que debe hacer una clase, pero no cómo lo hace.
- Una clase puede implementar cualquier número de interfaces.

8.4.1. Definición de una interfaz

Una interfaz se define casi igual que una clase:

```
acceso interface nombre {
  tipo_devuelto método1(lista_de_parámetros);
  tipo_devuelto método2(lista_de_parámetros);
  tipo var_final1=valor;
  tipo var_final2=valor;
// ...
  tipo_devuelto métodoN(lista_de_parámetros);
  tipo var_finalN=valor;
}
```

- *acceso* puede ser **public** o no usarse.
  - Si no se utiliza (acceso por defecto) la interfaz está sólo disponible para otros miembros del paquete en el que ha sido declarada.
  - Si se usa **public**, puede ser usada por cualquier código (todos los métodos y variables son implícitamente **públicos**).
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Un interfaz puede tener variables pero serán implícitamente **final** y **static**.

Ejemplo definición de interfaz

```
interface Callback {
  void callback(int param);
}
```

8.4.2. Implementación de una interfaz

- Para implementar una interfaz, la clase debe implementar todos los métodos de la interfaz. Se usa la palabra reservada **implements**.
- Una vez declarada la interfaz, puede ser implementada por varias clases.
- Cuando implementemos un método de una interfaz, tiene que declararse como **public**.
- Si una clase implementa dos interfaces que declaran el mismo método, entonces los clientes de cada interfaz usarán el mismo método.

Ejemplo de uso de interfaces: P28

```
class Client implements Callback {
  public void callback(int p) {
    System.out.println("callback llamado con " + p);
  }
  void nonIfaceMeth() {
    System.out.println("Las clases que implementan interfaces " +
                       "además pueden definir otros miembros.");
  }
}
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

8.4.3. Acceso a implementaciones a través de referencias de la interfaz

- Se pueden declarar variables usando como tipo un interfaz, para referenciar objetos de clases que implementan ese interfaz.
- El método al que se llamará con una variable así, se determina en tiempo de ejecución.

Ejemplo

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

- Con estas variables sólo se pueden usar los métodos que hay en la interfaz.

Otro ejemplo: P29

```
// Otra implementación de  Callback.
class AnotherClient implements Callback {
  public void callback(int p) {
    System.out.println("Otra versión de callback");
    System.out.println("El cuadrado de p es " + (p*p));
  }
}


class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob; // c hace referencia a un objeto AnotherClient
    c.callback(42);
  }
}
```


8.4.4. Implementación parcial

- Si una clase incluye una interfaz, pero no implementa todos sus métodos, entonces debe ser declarada como **abstract**.

8.4.5. Variables en interfaces

- Las variables se usan para importar constantes compartidas en múltiples clases.
- Si una interfaz no tiene ningún método, entonces cualquier clase que incluya esta interfaz no tendrá que implementar nada.

Ejemplo: P30/AskMe.java

```
import java.util.Random;
interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30) return NO;          // 30%
    else if (prob < 60) return YES;    // 30%
    else if (prob < 75) return LATER; // 15%
    else if (prob < 98) return SOON;  // 13%
    else return NEVER;                 // 2%
  }
}
```

```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No"); break;
      case YES:
        System.out.println("Si"); break;
      case MAYBE:
        System.out.println("Puede ser"); break;
      case LATER:
        System.out.println("Mas tarde"); break;
      case SOON:
        System.out.println("Pronto"); break;
      case NEVER:
        System.out.println("Nunca"); break;
    }
  }
  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

8.4.6. Las interfaces se pueden extender

- Una interfaz puede heredar otra utilizando la palabra reservada **extends**.
- Una clase que implemente una interfaz que herede de otra, debe implementar todos los métodos de la cadena de herencia.

Ejemplo: P31/IFExtend.java

```
interface A {
  void meth1();
  void meth2();
}
interface B extends A {
  void meth3();
}
class MyClass implements B {
  public void meth1() {
    System.out.println("Implemento meth1().");
  }
  public void meth2() {
    System.out.println("Implemento meth2().");
  }
  public void meth3() {
    System.out.println("Implemento meth3().");
  }
}
class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

9. Gestión de excepciones

Una excepción es una condición anormal que surge en una secuencia de código durante la ejecución de un programa. O sea, es un error en tiempo de ejecución.

9.1. Fundamentos

- Cuando surge una condición excepcional se crea un objeto que representa la excepción, y se *envía* al método que ha provocado la excepción.

  Este método puede gestionar la excepción él mismo o bien pasarla al método llamante. En cualquier caso, la excepción es *capturada* y procesada en algún punto.

- La gestión de excepciones usa las palabras reservadas **try**, **catch**, **throw**, **throws** y **finally**.

Forma general de un bloque de gestión de excepciones

```
try {
  // bloque de código
}
catch (TipoExcepcion1 exOb){
  // gestor de excepciones para TipoExcepcion1
}
catch (TipoExcepcion2 exOb){
  // gestor de excepciones para TipoExcepcion2
}
// ...
finally {
  // bloque de código que se ejecutara antes de
  // que termine el bloque try
}
```


9.2. Tipos de excepción

Todos los tipos de excepción son subclase de **Throwable**. Esta clase tiene dos subclases:

1. **Exception**: Se usa para las excepciones que deberían capturar los programas de usuario.

   Esta clase tiene como subclase a **RuntimeException**, que representa excepciones definidas automáticamente por los programas (división por 0, índice inválido de matriz, etc). Además tiene otras subclases como `ClassNotFoundException`, `InterruptedException`, etc.

2. **Error**: Excepciones que no se suelen capturar en condiciones normales.

   Suelen ser fallos catastróficos no gestionados por nuestros programas. Ejemplo: desbordamiento de la pila.

9.3. Excepciones no capturadas

Ejemplo: P32/Exc1.java

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

- Cuando el intérprete detecta la división por 0 construye un objeto de excepción y lanza la excepción. Esto detiene la ejecución de `Exc1`, ya que no hemos incluido un *gestor de la excepción*.
- Cualquier excepción no tratada por el programa será tratada por el *gestor por defecto* que muestra la excepción y el trazado de la pila en la salida estándar, y termina el programa.

Salida generada por anterior programa

```
java.lang.AritmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

9.4. try y catch

Si incluimos nuestro propio gestor de excepciones evitamos que el programa termine automáticamente. Usaremos un bloque **try-catch**.

Ejemplo: P33/Exc2.java

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // controla un bloque de código.
      d = 0;
      a = 42 / d;
      System.out.println("Esto no se imprimirá.");
    }
    catch (ArithmeticException e) {// captura el error de división
      System.out.println("División por cero.");
    }
    System.out.println("Después de la sentencia catch.");
  }
}
```


El objetivo de una sentencia **catch** bien diseñada debería ser resolver la condición de excepción y continuar.

Otro ejemplo: P34/HandleError.java

```
// Gestiona una excepción y continua.
import java.util.Random;
class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division por cero.");
        a = 0; // asigna a la variable el valor 0 y continua
      }
      System.out.println("a: " + a);
    }
  }
}
```

9.4.1. Descripción de una excepción

La clase **Throwable** sobreescribe el método **toString()** de la clase **Object**, devolviendo una cadena con la descripción de la excepción.

```
catch (ArithmeticException e) {
  System.out.println("Excepcion: " + e);
  a = 0; // hacer a=0 y continuar
}
```

Salida producida cuando se produce la excepción

```
Excepcion: java.lang.ArithmeticException: / by zero
```

9.5. Clausula catch múltiple

En algunos casos un bloque de código puede activar más de un tipo de excepción. Usaremos varios bloques **catch**.

El siguiente programa produce una excepción si se ejecuta sin parámetros y otra distinta si se ejecuta con un parámetro.

Ejemplo: P35/MultiCatch.java

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Division por 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indice fuera de limites: " + e);
    }
    System.out.println("Despues del bloque try/catch.");
  }
}
```

Al ordenar los bloques **catch**, las subclases de excepción deben ir antes que la superclase (en caso contrario no se ejecutarían nunca y daría error de compilación por código no alcanzable).

Ejemplo: P36/SuperSubCatch.java

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    }
    catch(Exception e) {
      System.out.println("catch para cualquier tipo de excepción.");
    }
    /* Este catch nunca se ejecutará */
    catch(ArithmeticException e) { // ERROR - no alcanzable
      System.out.println("Esto nunca se ejecutará.");
    }
  }
}
```

9.6. Sentencias try anidadas

- Una sentencia **try** puede estar incluida dentro del bloque de otra sentencia **try**.
- Cuando un **try** no tiene **catch** para una excepción se busca si lo hay en el **try** más externo, y así sucesivamente.

Ejemplo: P37/NestTry.java

```
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;
/* Si no hay ningún argumento en la línea, de órdenes, la siguiente
   sentencia generará una excepción de división por cero. */
      int b = 42 / a;
      System.out.println("a = " + a);
      try { // bloque try anidado
/* Si se utiliza un argumento en la línea, de órdenes, la siguiente
   sentencia generará una excepción de división por cero. */
        if(a==1) a = a/(a-a); // división por cero
        /* Si se le pasan dos argumentos en la línea de órdenes,
           se genera una excepción al sobrepasar los límites
           del tamaño de la matriz. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // genera una excepción de fuera de límites
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Indice fuera de limites: " + e);
      }
    } catch(ArithmeticException e) {
      System.out.println("División por 0: " + e);
    }
  }
}
```


◊ Sentencias try anidadas en forma menos obvia

Ejemplo: P38/MethNestTry.java

```
/* Las sentencias try pueden estar implícitamente anidadas
   a través de llamadas a métodos. */
class MethNestTry {
  static void nesttry(int a) {
    try { // bloque try anidado
      /* Si se utiliza un argumento en la línea de órdenes, la
         siguiente sentencia efectúa división por cero */
      if(a==1) a = a/(a-a); // división por zero
      /* Si se le pasan dos argumentos en la línea de órdenes,
         se sobrepasan los límites de la matriz */
      if(a==2) {
        int c[] = { 1 };
        c[42] = 99; // genera una excepción de fuera de límites
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indice fuera de limites: " + e);
    }
  }
  public static void main(String args[]) {
    try {
      int a = args.length;
      /* Si no hay ningún argumento en la línea de órdenes, la
         siguiente sentencia generará una excepción de división
         por cero */
      int b = 42 / a;
      System.out.println("a = " + a);
      nesttry(a);
    } catch(ArithmeticException e) {
      System.out.println("División por 0: " + e);
    }
  }
}
```

9.7. Lanzar excepciones explícitamente: throw

Usando la sentencia **throw** es posible hacer que el programa lance una excepción de manera explícita: `throw objetoThrowable;`

El `objetoThrowable` puede obtenerse mediante:

1. El parámetro de una sentencia **catch**.
2. Con el operador **new** .

Ejemplo: P39/ThrowDemo.java

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Captura dentro de demoproc.");
      throw e; // relanza la excepción
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Nueva captura: " + e);
    }
  }
}
```

El flujo de ejecución se detiene tras la sentencia **throw** (cualquier sentencia posterior no se ejecuta).

Salida del anterior ejemplo:

```
Captura dentro de demoproc.
Nueva captura: java.lang.NullPointerException: demo
```


9.8. Sentencia throws

Sirve para listar los tipos de excepción que un método puede lanzar.

- Debe usarse para proteger los métodos que usan a éste, si tal método lanza la excepción pero no la maneja.
- Es necesario usarla con todas las excepciones excepto **Error** y **RuntimeException** y sus subclases.
- Si las excepciones que lanza el método y no maneja no se ponen en **throws** se producirá error de compilación.

Forma general de declaración de método con throws

```
tipo metodo(lista_de_parametros) throws lista_de_excepciones
{
  // cuerpo del metodo
}
```

Ejemplo: P40/ThrowsDemo.java

```
// Programa erróneo que no compila
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

El método que use al del **throws** debe capturar todas las excepciones listadas con la sentencia **throws**.

Ejemplo: P41/ThrowsDemo.java

```
// Programa correcto
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Captura " + e);
    }
  }
}
```

9.9. Sentencia finally

- Se puede usar esta sentencia en un bloque **try-catch** para ejecutar un bloque de código después de ejecutar las sentencias del **try** y del **catch**.
- Se ejecutará tanto si se lanza, como si no una excepción, y aunque no haya ningún **catch** que capture esa excepción.
- Podría usarse por ejemplo para cerrar los archivos abiertos.

Ejemplo de uso de finally: P42/FinallyDemo.java

```
class FinallyDemo {
  static void procA() {// Lanza una excepción fuera del metodo
    try {
      System.out.println("Dentro de procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("Sentencia finally de procA");
    }
  }
```



```
  static void procB() {// Ejecuta la sentencia return dentro del try
    try {
      System.out.println("Dentro de procB");
      return;
    } finally {
      System.out.println("Sentencia finally de procB");
    }
  }
  static void procC() {// Ejecuta un bloque try normalmente
    try {
      System.out.println("Dentro de procC");
    } finally {
      System.out.println("Sentencia finally de procC");
    }
  }
  public static void main(String args[]) {
    try {procA();
    } catch (Exception e) {
      System.out.println("Excepción capturada");
    }
    procB(); procC();
  }
}
```

Salida del anterior programa

```
Dentro de procA
Sentencia finally de procA
Excepción capturada
Dentro de procB
Sentencia finally de procB
Dentro de procC
Sentencia finally de procC
```

9.10. Subclases de excepciones propias

Sirven para crear tipos propios de excepción que permitan tratar situaciones específicas en una aplicación.

- Para ello sólo hay que definir una subclase de **Exception**.
- Estas subclases de excepciones no necesitan implementar nada.
- La clase **Exception** no define ningún método, sólo hereda los de **Throwable**.

Ejemplo: P43/ExceptionDemo.java

```
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return "MyException[" + detail + "]";
  }
}
```



```
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Ejecuta compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Finalización normal");
  }
  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Captura " + e);
    }
  }
}
```

Salida de este programa

```
Ejecuta compute(1)
Finalización normal
Ejecuta compute(20)
Captura MyException[20]
```

10. Programación Multihilo (Multihebra)

- Un programa multihilo contiene dos o más partes que pueden ejecutarse concurrentemente (aunque sólo tengamos una CPU).
- Esto permite escribir programas muy eficientes que utilizan al máximo la CPU, reduciendo al mínimo, el tiempo que está sin usarse.
- Java incluye características directamente en el lenguaje y API, para construir programas multihilo (a diferencia de otros lenguajes).

10.1. El hilo principal

- El hilo principal es lanzado por la JVM (Java Virtual Machine) cuando ejecutas una aplicación o por un navegador cuando se ejecuta un applet.
- Desde este hilo se crearán el resto de hilos del programa.
- Debe ser el último que termine su ejecución. Cuando el hilo principal finaliza, termina el programa.

Ejemplo de acceso al hilo principal:
P44/CurrentThreadDemo.java

```
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Hilo actual: " + t);
    t.setName("Mi hilo"); //cambia el nombre del hilo
    System.out.println("Después del cambio de nombre: " + t);
    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");}
  }
}
```


Salida del programa

```
Hilo actual: Thread[main,5,main]
Después del cambio de nombre: Thread[Mi hilo,5,main]
5
4
3
2
1
```

Cuando se usa `t` como argumento de `println()` aparece por orden: nombre del hilo, prioridad y el nombre del grupo.

10.2. Creación de un hilo

En Java hay dos formas de crear nuevos hilos:

- Implementando el interfaz Runnable.
- Extendiendo la clase Thread.

10.2.1. Implementación del interfaz Runnable

- Consiste en declarar una clase que implemente **Runnable** y sobreescribir el método `run()`:

  `public abstract void run()`

- Dentro de `run()` incluimos el código a ejecutar por el nuevo hilo.
- Luego se crea un objeto de la clase **Thread** dentro de esa clase. Al constructor de *Thread* le pasamos como argumento el objeto creado de la nueva clase (instancia de una clase que implemente **Runnable**):

  `Thread(Runnable objetoHilo,String nombreHilo)`

- Finalmente llamamos al método `start()` con el objeto anterior.

  `synchronized void start()`

Ejemplo: P45/ThreadDemo.java

```
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    t = new Thread(this, "Hilo hijo");// Crea un nuevo hilo
    System.out.println("Hilo hijo: " + t);
    t.start(); // Comienza el hilo
  }
  public void run() {//Punto de entrada del segundo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo hijo: " + i);
        Thread.sleep(500); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo hijo."); }
    System.out.println("Sale del hilo hijo.");
  }
}
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // crea un nuevo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo principal: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal.");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```


Salida del anterior programa

```
Hilo hijo: Thread[Hilo hijo,5,main]
Hilo principal:5
Hilo hijo:5
Hilo hijo:4
Hilo principal:4
Hilo hijo:3
Hilo hijo:2
Hilo principal:3
Hilo hijo:1
Sale del hilo hijo.
Hilo principal:2
Hilo principal:1
Sale del hilo principal.
```

10.2.2. Extensión de la clase Thread

- Consiste en declarar una clase que herede de la clase **Thread** y sobreescribir también el método `run()`:
- Dentro de `run()` incluimos el código a ejecutar por el nuevo hilo.
- Luego se crea un objeto de la nueva clase.
- Finalmente llamamos al método `start()` con el objeto anterior.

Ejemplo: P46/ExtendThread.java

```
class NewThread extends Thread {
  NewThread() {
    super("Hilo Demo"); // Crea un nuevo hilo
    System.out.println("Hilo hijo: " + this);
    start(); // Comienza el hilo
  }
  public void run() {// Este es el punto de entrada del segundo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo hijo: " + i);
        Thread.sleep(500); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo hijo.");
    }
    System.out.println("Sale del hilo hijo.");
  }
}
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // crea un nuevo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo principal: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal.");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```


10.2.3. Elección de una de las dos opciones

Depende del tipo de clase que se vaya a crear.

- Si la clase hereda de otra, no queda más remedio que implementar **Runnable**.
- Normalmente es más sencillo heredar de **Thread**.
- Pero algunos programadores piensan que una clase no debe extenderse si no va a ser ampliada o modificada. Por eso, si no vamos a sobreescribir ningún otro método de **Thread**, quizás sería mejor implementar **Runnable**.

10.3. Creación de múltiples hilos

Un programa en Java puede crear tantos hilos como quiera.

Ejemplo de creación de tres hilos: P47/MultiThreadDemo.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }
  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    }
    catch (InterruptedException e) {
      System.out.println(name + "Interrupción del hilo hijo" +name);
    }
    System.out.println("Sale del hilo hijo" + name);
  }
}
```



```
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("Uno"); // comienzan los hilos
    new NewThread("Dos");
    new NewThread("Tres");
    try {
      // espera un tiempo para que terminen los otros hilos
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```

Salida del programa

```
Nuevo hilo: Thread[Uno,5,main]
Nuevo hilo: Thread[Dos,5,main]
Nuevo hilo: Thread[Tres,5,main]
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Dos: 3
Tres: 3
Uno: 2
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Sale del hilo.Uno
```

```
Sale del hilo.Dos
Sale del hilo.Tres
Sale del hilo principal.
```

10.4. Utilización de isAlive() y join()

Hay dos formas de determinar si un hilo ha terminado. Se basan en los siguientes métodos de la clase `Thread`.

- Con el método `isAlive()`. Devuelve **true** si el hilo al que se hace referencia está todavía ejecutándose.

  ```
  final boolean isAlive() throws InterruptedException
  ```

- Con el método `join()`. Este método detiene el hilo actual hasta que termine el hilo sobre el que se llama `join()`. Es usado por tanto para que unos hilos esperen a la finalización de otros.

  ```
  final void join throws InterruptedException
  ```

Ejemplo de uso de isAlive() y join(): P48/DemoJoin.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }
```



```
  public void run() {// Este es el punto de entrada del hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo"+name); }
    System.out.println("Sale del hilo " + name);
  }
}
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("Uno");
    NewThread ob2 = new NewThread("Dos");
    NewThread ob3 = new NewThread("Tres");
    System.out.println("El hilo Uno está vivo: " + ob1.t.isAlive());
    System.out.println("El hilo Dos está vivo: " + ob2.t.isAlive());
    System.out.println("El hilo Tres está vivo: " + ob3.t.isAlive());
    try {// espera hasta que terminen los otros hilos
      System.out.println("Espera finalización de los otros hilos.");
      ob1.t.join(); ob2.t.join(); ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal"); }
    System.out.println("El hilo Uno está vivo: " + ob1.t.isAlive());
    System.out.println("El hilo Dos está vivo " + ob2.t.isAlive());
    System.out.println("El hilo Tres está vivo: " + ob3.t.isAlive());
    System.out.println("Sale del hilo principal.");
  }
}
```

Salida del programa

```
Nuevo hilo: Thread[Uno,5,main]
Nuevo hilo: Thread[Dos,5,main]
Nuevo hilo: Thread[Tres,5,main]
El hilo Uno está vivo: true
El hilo Dos está vivo: true
El hilo Tres está vivo: true
Espera finalización de los otros hilos.
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Dos: 3
Tres: 3
Uno: 2
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Sale del hilo Uno
Sale del hilo Dos
Sale del hilo Tres
El hilo Uno está vivo: false
El hilo Dos está vivo false
El hilo Tres está vivo: false
Sale del hilo principal.
```


10.5. Prioridades de los hilos

- La prioridad de un hilo es un valor entero que indica la prioridad relativa de un hilo respecto a otro.
- Se utiliza para decidir cuando pasar a ejecutar otro hilo (cambio de contexto).
  - Cuando un hilo cede el control (por abandono explícito o por bloqueo en E/S), se ejecuta a continuación el que tenga *mayor prioridad*.
  - Un hilo puede ser desalojado por otro con prioridad más alta, tan pronto como éste desee hacerlo.
- Pero en la práctica la cantidad de CPU que recibe cada hilo depende además de otros factores como la forma en que el sistema operativo implementa la multitarea.
- Para establecer la prioridad de un hilo usamos **setPriority()** de la clase **Thread** .

  ```
  final void setPriority(int level)
  ```
- `level` puede variar entre `MIN_PRIORITY` y `MAX_PRIORITY` (1 y 10 en la actualidad). La prioridad por defecto es `NORM_PRIORITY` (5 actualmente).
- Para obtener la prioridad de un hilo:

  ```
  final int getPriority()
  ```

Ejemplo: P49/HiLoPri.java

```
class clicker implements Runnable {
  int click = 0;
  Thread t;
  private volatile boolean running = true;
  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }
```

```
    public void run() {
      while (running) {
        click++;
      }
    }
    public void stop() {
      running = false;
    }
    public void start() {
      t.start();
    }
  }
  class HiLoPri {
    public static void main(String args[]) {
      Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
      clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
      clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
      lo.start();
      hi.start();
      try {
        Thread.sleep(10000);
      } catch (InterruptedException e) {
          System.out.println("Hilo principal interrumpido.");}
      lo.stop();
      hi.stop();
      try {
        hi.t.join();
        lo.t.join();
      } catch (InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
      System.out.println("Hilo de prioridad baja: " + lo.click);
      System.out.println("Hilo de prioridad alta: " + hi.click);
    }
  }
```


Salida del programa en linux Redhat 8.0

```
Hilo de prioridad baja: 715311411
Hilo de prioridad alta: 797505916
```

Salida del programa en Windows 98

```
Hilo de prioridad baja: 23904884
Hilo de prioridad alta: 413745323
```

10.6. Sincronización

- Cuando dos o más hilos necesitan acceder a un recurso compartido, entonces necesitan alguna forma de asegurar que el recurso se usa sólo por un hilo al mismo tiempo: **Sincronización**.
- Un *monitor* (*semáforo*) es un objeto usado como un cerrojo de exclusión mutua. Sólo un hilo puede poseer el monitor en un momento determinado.
- Cuando un hilo entra en el monitor, los demás hilos que intentan entrar en él, quedan suspendidos hasta que el primer hilo lo deja.
- En Java el código puede sincronizarse de dos formas:
  - Con métodos sincronizados.
  - Con sentencia **synchronized**.

10.6.1. Uso de métodos sincronizados

- Todos los objetos de Java tienen asociado su propio monitor implícito.
- Para entrar en el monitor de un objeto sólo hay que llamar a un método **synchronized**.
- Cuando un hilo esté ejecutando un método sincronizado, todos los demás hilos que intenten ejecutar cualquier método sincronizado del mismo objeto tendrán que esperar.

Ejemplo de programa que no usa sincronización: P50/Synch.java

```
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}
```



```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hola");
    Caller ob2 = new Caller(target, "Sincronizado");
    Caller ob3 = new Caller(target, "Mundo");
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
  }
}
```

Salida del anterior programa

```
[Hola[Sincronizado[Mundo]
]
]
```

- El resultado de este programa muestra que la salida de los tres mensajes aparece mezclada, ya que los tres hilos están ejecutando el método `call()` del mismo objeto a la vez.
- Solución: Restringimos el acceso a `call()` a un sólo hilo en un momento determinado.

  Ejemplo: P51/Synch.java

  ```
  class Callme {
      synchronized void call(String msg) {
      ...
  ```

- Nueva salida:

  ```
  [Hola]
  [Sincronizado]
  [Mundo]
  ```

10.6.2. Sentencia synchronized

- A veces la solución de sincronizar un método no es posible porque no tenemos el código fuente de la clase, y no podemos hacer que el método sea sincronizado.
- En ese caso podemos usar la sentencia **synchronized**

```
synchronized(objeto){
  // sentencias que se sincronizan
}
```

- `objeto` es el objeto que sincronizamos.
- Esta sentencia hace que cualquier llamada a algún método de `objeto` se ejecutará después de que el hilo actual entre en el monitor de `objeto`.

Ejemplo que hace lo mismo de antes: P52/Synch1.java

```
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrumpido");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this); t.start();
  }
```



```
  public void run() {
    synchronized(target) { target.call(msg);}
  }
}
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hola");
    Caller ob2 = new Caller(target, "Sincronizado");
    Caller ob3 = new Caller(target, "Mundo");
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
  }
}
```

10.7. Comunicación entre hilos

- Los hilos pueden comunicarse entre sí mediante el uso de los métodos **wait()**, **notify()** y **notifyAll()** de la clase **Object** (son métodos finales).
  - **wait()** dice al hilo llamante que deje el monitor y que pase a estado suspendido (dormido) hasta que otro hilo entre en el mismo monitor y llame a **notify()**.
  - **notify()** despierta el primer hilo que llamó a **wait()** sobre el mismo objeto.
  - **notifyAll()** despierta todos los hilos que llamaron a **wait()** sobre el mismo objeto.
- Estos métodos sólo pueden ser llamados desde métodos sincronizados.

Productor/consumidor de un sólo carácter (versión errónea): P53/PC.java

```
class Q {
  int n;
  synchronized int get() {
    System.out.println("Obtengo: " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out.println("Pongo: " + n);
  }
}
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Productor").start();
  }
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
```



```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Pulsa Control-C para parar.");
  }
}
```

Salida del programa

```
Pongo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Pongo: 2
Pongo: 3
Pongo: 4
Pongo: 5
Pongo: 6
Pongo: 7
Obtengo: 7
```

Solución correcta con wait y notify: P54/PCFixed.java

```
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    if(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
    System.out.println("Obtengo: " + n);
    valueSet = false;
    notify();
    return n;
  }
  synchronized void put(int n) {
    if(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
    this.n = n;
    valueSet = true;
    System.out.println("Pongo: " + n);
    notify();
  }
}
```



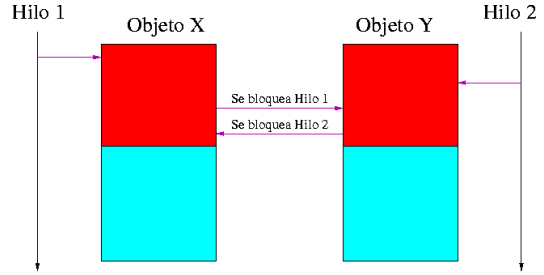
```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Productor").start();
  }
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Pulsa Control-C para parar.");
  }
}
```

Salida del programa

```
Pongo: 1
Obtengo: 1
Pongo: 2
Obtengo: 2
Pongo: 3
Obtengo: 3
Pongo: 4
Obtengo: 4
Pongo: 5
Obtengo: 5
Pongo: 6
Obtengo: 6
Pongo: 7
Obtengo: 7
```

10.7.1. Interbloqueos

- Un interbloqueo ocurre cuando dos hilos tienen una dependencia circular sobre un par de objetos sincronizados.
- Por ejemplo, un hilo entra en el monitor del objeto X y otro entra en el monitor de Y. Si X intenta llamar cualquier método sincronizado de Y, se bloqueará. Sin embargo, si el hilo de Y, intenta ahora llamar cualquier método sincronizado de X, los dos hilos quedarán detenidos para siempre.



Ejemplo de interbloqueo: P55/Deadlock.java

```
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entró en A.foo");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrumpido");
    }
    System.out.println(name + " intentando llamar a B.last()");
    b.last();
  }
  synchronized void last() {
    System.out.println("Dentro de A.last");
  }
}
class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entró en B.bar");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("B Interrumpido");
    }
    System.out.println(name + " intentando llamar a A.last()");
    a.last();
  }
  synchronized void last() {
    System.out.println("Dentro de A.last");
  }
}
```

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();
  Deadlock() {
    Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();
    a.foo(b);
    System.out.println("Regreso al hilo principal");
  }
  public void run() {
    b.bar(a);
    System.out.println("Regreso al otro hilo");
  }
  public static void main(String args[]) {
    new Deadlock();
  }
}
```

Salida del programa hasta que queda bloqueado

```
MainThread entró en A.foo
RacingThread entró en B.bar
MainThread intentando llamar a B.last
RacingThread intentando llamar a A.last
```

10.8. Suspender, reanudar y terminar hilos

- La suspensión de la ejecución de un hilo es útil en algunos casos.

  Por ejemplo, un hilo puede usarse para mostrar la hora actual. Si el usuario no desea verla en un momento determinado, entonces tal hilo debería suspenderse.

- Una vez suspendido un hilo, éste puede reanudar su ejecución como veremos.

10.8.1. En Java 1.1 y anteriores: suspend(), resume() y stop()

Dentro de la clase **Thread** encontramos los métodos:

```
final void suspend()
final void resume()
void stop()
```

Ejemplo de uso de suspend() y resume(): P57/SuspendResume.java

```
class NewThread implements Runnable {
  String name; // nombre del metodo
  Thread t;
  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }
  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo " + name);
    }
    System.out.println("Sale del hilo " + name);
  }
}
```

```
class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("Uno");
    NewThread ob2 = new NewThread("Dos");
    try {
      Thread.sleep(1000);
      ob1.t.suspend();
      System.out.println("Suspende el hilo Uno");
      Thread.sleep(1000);
      ob1.t.resume();
      System.out.println("Reanuda el hilo Uno");
      ob2.t.suspend();
      System.out.println("Suspende el hilo Dos");
      Thread.sleep(1000);
      ob2.t.resume();
      System.out.println("Reanuda el hilo Dos");
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");
    }
    // espera hasta que terminen los otros hilos
    try {
      System.out.println("Espera finalización de los otros hilos.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```


Salida del anterior programa

```
Nuevo hilo: Thread[Uno,5,main]
Nuevo hilo: Thread[Dos,5,main]
Uno: 15
Dos: 15
Uno: 14
Dos: 14
Uno: 13
Dos: 13
Uno: 12
Dos: 12
Uno: 11
Dos: 11
Suspende el hilo Uno
Dos: 10
Dos: 9
Dos: 8
Dos: 7
Dos: 6
Reanuda el hilo Uno
Suspende el hilo Dos
Uno: 10
Uno: 9
Uno: 8
Uno: 7
Uno: 6
Reanuda el hilo Dos
Espera finalización de los otros hilos.
Dos: 5
Uno: 5
Dos: 4
Uno: 4
Dos: 3
Uno: 3
Dos: 2
```

```
Uno: 2
Dos: 1
Uno: 1
Sale del hilo Dos
Sale del hilo Uno
Sale del hilo principal.
```

10.8.2. En Java 2

- Los métodos `suspend()`, `resume()` y `stop()` están obsoletos (*deprecated*) en Java 2, ya que pueden causar serios fallos en el sistema (su uso puede provocar interbloqueos).
- En Java 2, para poder suspender, reanudar y terminar la ejecución de un hilo usaremos una variable bandera que indica el estado de ejecución del hilo. Haremos uso también de `wait()` y `notify()` o `notifyAll()`.

Ejemplo de uso con wait() y notify(): P57/SuspendResume.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    suspendFlag = false;
    t.start(); // Comienza el hilo
  }
```



```
  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo" + name);
    }
    System.out.println("Sale del hilo" + name);
  }
  void mysuspend() {
    suspendFlag = true;
  }
  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}
class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("Uno");
    NewThread ob2 = new NewThread("Dos");
    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspende el hilo Uno");
      Thread.sleep(1000);
      ob1.myresume();
```

```
        System.out.println("Reanuda el hilo Uno");
        ob2.mysuspend();
        System.out.println("Suspende el hilo Dos");
        Thread.sleep(1000);
        ob2.myresume();
        System.out.println("Reanuda el hilo Dos");
      } catch (InterruptedException e) {
        System.out.println("Interrupción del hilo principal");
      }
      // espera hasta que terminen los otros hilos
      try {
        System.out.println("Espera finalización de los otros hilos.");
        ob1.t.join();
        ob2.t.join();
      } catch (InterruptedException e) {
        System.out.println("Interrupción del hilo principal");
      }
      System.out.println("Sale del hilo principal.");
    }
  }
```


11. Abstract Windows Toolkit

11.1. Introducción

- **AWT**: parte de Java para construir interfaces gráficas de usuario.
- Para crear una interfaz gráfica de usuario hace falta:
  1. Un ***contenedor*** (*container*), que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc).
  2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc.
  3. ***Modelo de eventos***:
     - Cada vez que el usuario realiza una acción (normalmente con ratón y teclado) sobre un componente, se produce un ***evento***, que el sistema operativo transmite al AWT.
     - El AWT crea entonces un objeto de una determinada clase de evento, derivada de `AWTEvent`.
     - Este evento es transmitido a un determinado método para que lo gestione.

11.2. Modelo de delegación de eventos

- **Event sources**: Son los objetos sobre los que se producen los eventos.
  - Suelen ser los componentes del interfaz gráfico.
  - Estos objetos *registran* los objetos (**Event listeners**) que habrán de gestionarlos.
- **Event listeners**: Objetos que gestionan los eventos.
  - Disponen de métodos que se llamarán automáticamente cuando se produzca un evento en el *objeto fuente*.
  - La forma de garantizar que disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interfaz ***Listener***.

- Existe un emparejamiento entre clases de evento y definiciones de interfaz receptora de eventos.

  Ejemplo: Un botón del interfaz gráfico puede generar un evento `ActionEvent`. La clase `ActionEvent` tiene asociado el interfaz receptor `ActionListener`.

- Cada interfaz receptora de eventos define los métodos adecuados para tratar los eventos particulares de ese tipo.

  `MouseListener` define los métodos
  `mouseClicked(MouseEvent e)`,
  `mouseEntered(MouseEvent e)`, `mouseExited(MouseEvent e)`,
  `mousePressed(MouseEvent e)` y
  `mouseReleased(MouseEvent e)`.

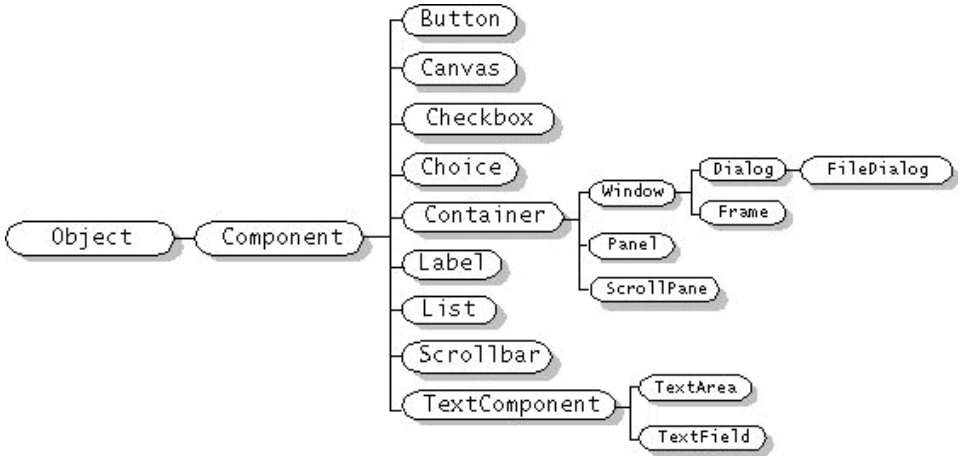
Ejemplo: AWT/P1/MiFrame.java

```
import java.awt.*;
import java.awt.event.*;
public class MiFrame extends Frame implements ActionListener {
  public MiFrame() {
    Button boton = new Button("Salir");
    boton.setName("Salir");
    setSize(300,300);
    add("Center",boton);
    setVisible(true);
    boton.addActionListener(this);
  }
  public void actionPerformed(ActionEvent event) {
    if(event.toString().indexOf("on Salir") != -1) {
       System.exit(0);
    }
  }
  static public void main(String args[]) {
    MiFrame frame=new MiFrame();
    frame.setVisible(true);
  }
}
```


11.3. Introducción a los componentes y eventos

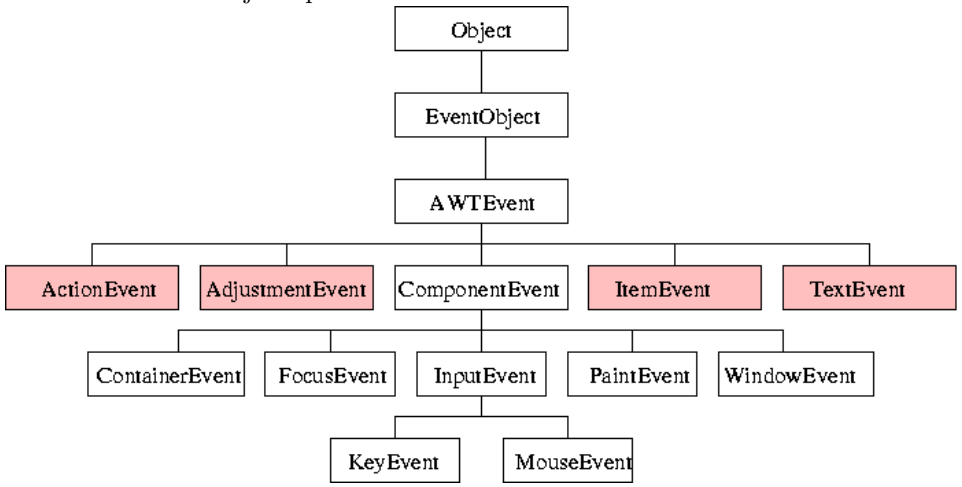
11.3.1. Jerarquía de componentes

- Los componentes de AWT pertenecen a la siguiente jerarquía de clases:



11.3.2. Jerarquía de eventos

- Todos los eventos de Java son objetos de clases que pertenecen a una determinada jerarquía de clases.



- **Eventos de alto nivel** (o semánticos): La acción de la que derivan tiene un significado en sí misma, en el contexto del interfaz gráfico de usuario.
  - Tipos
    - `ActionEvent`: Se produce al hacer click en botones, al elegir una opción de un menú, al hacer doble click en un objeto `List`, etc.
    - `AdjustmentEvent`: Se produce al cambiar valores en una barra de desplazamiento.
    - `ItemEvents`: Se produce al elegir valores (items).
    - `TextEvent`: Se produce al cambiar texto.
- **Eventos de bajo nivel**: Representan una entrada de bajo nivel sobre un componente.
  - Son los que se producen con las operaciones elementales con el ratón, teclado, containers y windows.
  - La principal diferencia con los anteriores es que permiten acceder al componente que generó el evento con el método `getComponent()`.

11.3.3. Relación entre Componentes y Eventos generados

Component	Eventos	Significado
Button	ActionEvent	Click en botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un item
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un item
Choice	ItemEvent	Seleccionar o deseleccionar un item
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente; mover o arrastrar el ratón
Container	ContainerEvent	Añadir o eliminar un componente de un container
List	ActionEvent	Hacer doble click sobre un item
	ItemEvent	Seleccionar o deseleccionar un item
MenuItem	ActionEvent	Seleccionar un item de un menú
Scrollbar	AdjustementEvent	Cambiar el valor del scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro
Window	WindowEvent	Acciones sobre una ventana: abrir, cerrar, iconizar, restablecer, iniciar el cierre.

- Además de los eventos de la tabla, hay que tener en cuenta que los eventos propios de una superclase del componente pueden afectar también a este componente.

  Por ejemplo la clase `TextArea` no tiene ningún evento específico propio, pero puede recibir los de su superclase `TextComponent`.

11.3.4. Interfaces Listener: Receptores de eventos

- Cada objeto que puede recibir un evento (event source), debe registrar uno o más objetos para que los gestione (event listener). Para ello usamos un método que tiene la forma:

  ```
  eventSourceObject.addEventListener(eventListenerObject);
  ```

  donde `eventSourceObject` es el objeto en el que se produce el evento, y `eventListenerObject` es el objeto que deberá gestionar los eventos.

- La clase del `eventListenerObject` debe implementar un interfaz `Listener` de un tipo correspondiente al tipo de evento asociado.

- La clase receptora de eventos proporciona la declaración de los métodos que serán llamados cuando se produzca un evento.

- Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente el interfaz Listener correspondiente, se deben definir todos los métodos de dicha interfaz.

- La siguiente tabla relaciona los distintos tipos de eventos, con la interfaz que se debe implementar para gestionarlos. Además muestra los métodos declarados en cada interfaz.

Evento	**Listener**	**Métodos del Listener**
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponenListener	componentHidden(), componentMoved() componentResized(), componetShown()
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked() ,mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

11.3.5. Clases Adapter

- Las interfaces receptoras de eventos de bajo nivel tienen más de un método, que deben ser implementados en las clases receptoras.

  El interfaz `MouseListener` contiene los métodos `mouseClicked()`, `mouseEntered()`, `mouseExited()`, `mousePressed()` y `mouseReleased()`.

- Las clases ***Adapter*** son clases predefinidas que contienen definiciones vacías para todos los métodos del interfaz receptor de eventos.
- Para crear una clase receptora de eventos, en vez de implementar un determinado ***Listener***, extenderemos la clase Adapter correspondiente, definiendo sólo los métodos de interés.

Ejemplo: AWT/P2/VentanaCerrable2.java

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

class VentanaCerrable2 extends Frame{
  public VentanaCerrable2(String title){
    super(title);
    setSize(500,500);
    CerrarVentana cv = new CerrarVentana();
    this.addWindowListener(cv);
  }
  public static void main(String args[]){
    VentanaCerrable2 ventana=new VentanaCerrable2(
                              "Ejemplo clase Adapter");
    ventana.setVisible(true);
  }
}
class CerrarVentana extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

11.3.6. Ejemplos de gestión de eventos

Ejemplo con eventos de bajo nivel: AWT/java1102.java

```
import java.awt.*;
import java.awt.event.*;  // Este es un paquete nuevo del JDK 1.1
public class java1102 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
    }
}
class MiFrame extends Frame {
    int ratonX;
    int ratonY;
    public void paint( Graphics g ) {
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
    }
}
class IHM {
    public IHM() {
        MiFrame ventana = new MiFrame();
        ventana.setSize( 300,300 );
        ventana.setTitle( "Tutorial de Java, Eventos" );
        ventana.setVisible( true );
        Proceso1 procesoVentana1 = new Proceso1();
        ventana.addWindowListener( procesoVentana1 );
        ProcesoRaton procesoRaton = new ProcesoRaton( ventana );
        ventana.addMouseListener( procesoRaton );
    }
}
```



```
class ProcesoRaton extends MouseAdapter {
  MiFrame ventanaRef; // Referencia a la ventana
    ProcesoRaton( MiFrame ventana ) {
        ventanaRef = ventana;
    }
    public void mousePressed( MouseEvent evt ) {
        ventanaRef.ratonX = evt.getX();
        ventanaRef.ratonY = evt.getY();
        ventanaRef.repaint();
   }
}
class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```

Otro ejemplo de eventos de bajo nivel: AWT/java1103.java

```
import java.awt.*;
import java.awt.event.*;
public class java1103 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();
    }
}
class MiFrame extends Frame {
    int ratonX; int ratonY;
    MiFrame( String nombre ) {
        setTitle( "Tutorial de Java, Eventos" );
        setSize( 300,200 );
        setName( nombre );
    }
    public void paint( Graphics g ) {
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY ); }
}
```

```
class IHM {
    public IHM() {
        MiFrame miFrame1 = new MiFrame( "Frame1" );
        miFrame1.setVisible( true );
        MiFrame miFrame2 = new MiFrame( "Frame2" );
        miFrame2.setVisible( true );
        Proceso1 procesoVentana1 = new Proceso1();
        miFrame1.addWindowListener( procesoVentana1 );
        miFrame2.addWindowListener( procesoVentana1 );
        ProcesoRaton procesoRaton = new ProcesoRaton( miFrame1,
                                                miFrame2 );
        miFrame1.addMouseListener( procesoRaton );
        miFrame2.addMouseListener( procesoRaton );  }
}
class ProcesoRaton extends MouseAdapter{
    MiFrame frameRef1,frameRef2;
    ProcesoRaton( MiFrame frame1,MiFrame frame2 ) {
        frameRef1 = frame1; frameRef2 = frame2; }
    public void mousePressed( MouseEvent evt ) {
        if(evt.getComponent().getName().compareTo("Frame1")==0){
            frameRef1.ratonX = evt.getX();
            frameRef1.ratonY = evt.getY();
            frameRef1.repaint();}
        else {
            frameRef2.ratonX = evt.getX();
            frameRef2.ratonY = evt.getY();
            frameRef2.repaint();
        }
    }
}
class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```


Ejemplo con eventos de nivel semántico y de bajo nivel: AWT/java1106.java

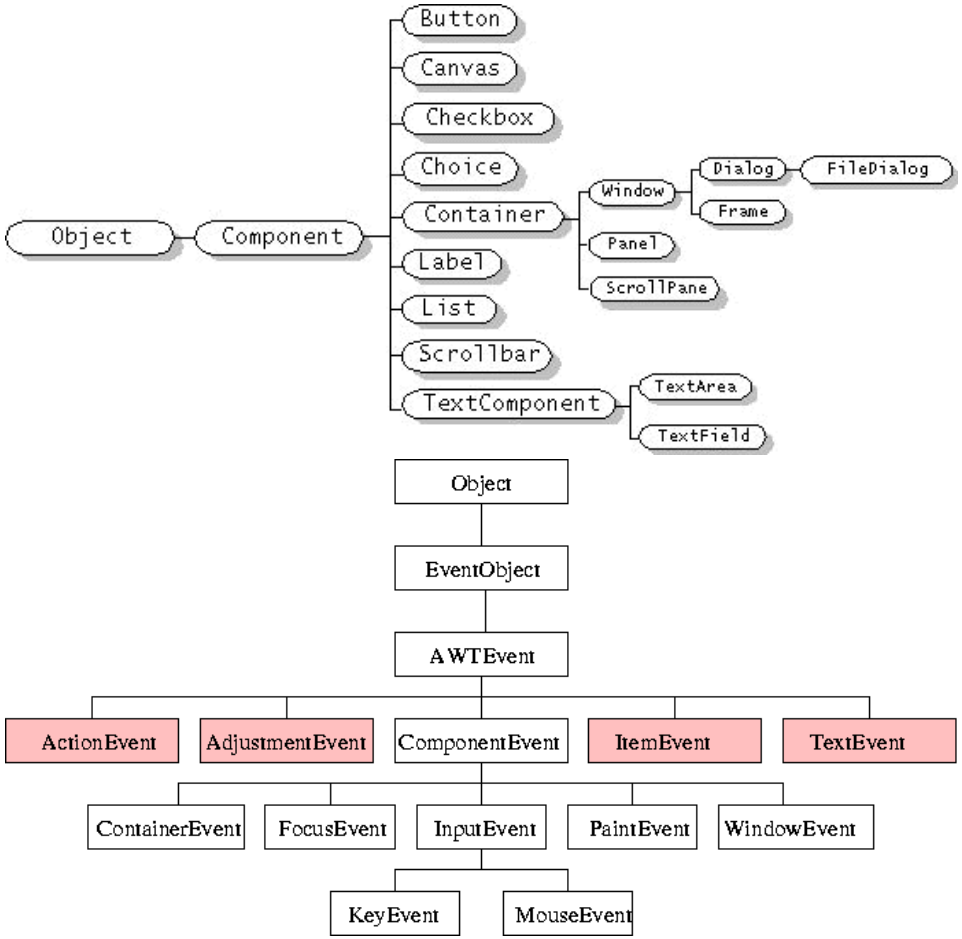
```
import java.awt.*;
import java.awt.event.*;
public class java1106 {
    public static void main( String args[] ) {
        IHM ihm = new IHM();}
}
class IHM {
    public IHM() {
        TextField miTexto = new TextField( "Cadena Inicial" );
        miTexto.setName( "CampoTexto" );
        Button miBoton = new Button( "Púlsame" );
        miBoton.setName( "Boton" );
        Frame miFrame = new Frame();
        miFrame.setSize( 300,200 );
        miFrame.setTitle( "Tutorial de Java, Eventos" );
        miFrame.setName( "Frame" );
        miFrame.add( "North",miBoton );
        miFrame.add( "South",miTexto );
        miFrame.setVisible( true );
        ProcesoAccion procesoAccion = new ProcesoAccion();
        miTexto.addActionListener( procesoAccion );
        miBoton.addActionListener( procesoAccion );
        ProcesoFoco procesoFoco = new ProcesoFoco();
        miTexto.addFocusListener( procesoFoco );
        miBoton.addFocusListener( procesoFoco );
        ProcesoRaton procesoRaton = new ProcesoRaton();
        miFrame.addMouseListener( procesoRaton );
        miTexto.addMouseListener( procesoRaton );
        miBoton.addMouseListener( procesoRaton );
        Proceso1 procesoVentana1 = new Proceso1();
        miFrame.addWindowListener( procesoVentana1 );
        }
    }
```

```
class ProcesoAccion implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
        System.out.println( "evt.getActionCommand() = " +
            evt.getActionCommand() );
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
          System.out.println(
          "Capturado actionPerformed sobre el objeto CampoTexto" );
        }
        if( evt.toString().indexOf("on Boton") != -1 ) {
          System.out.println(
          "Capturado actionPerformed sobre el objeto Boton" );
        }
    }
}
class ProcesoFoco implements FocusListener{
    public void focusGained( FocusEvent evt ) {
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
            "Capturado focusGained sobre el objeto CampoTexto" );
        }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
            "Capturado focusGained sobre el objeto Boton" );
        }
    }
    public void focusLost( FocusEvent evt ) {
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
            "Capturado focusLost sobre el objeto CampoTexto" );
        }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
                "Capturado focusLost sobre el objeto Boton" ); }
    }
}
```



```
class ProcesoRaton extends MouseAdapter {
    public void mousePressed( MouseEvent evt ) {
        if( evt.toString().indexOf("on Frame") != -1 ) {
            System.out.println(
            "Capturado mousePressed sobre el objeto Frame" );
        }
        if( evt.toString().indexOf("on CampoTexto") != -1 ) {
            System.out.println(
            "Capturado mousePressed sobre el objeto CampoTexto" );
        }
        if( evt.toString().indexOf("on Boton") != -1 ) {
            System.out.println(
            "Capturado mousePressed sobre el objeto Boton" );
        }
    }
}
class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```

11.4. Componentes y eventos de AWT



11.4.1. Clase Component

- Es una clase abstracta de la que derivan todas las clases del AWT.
- Representa todo lo que tiene posición, tamaño, puede ser pintado en pantalla y puede recibir eventos.
- Son los controles básicos de las aplicaciones.

Métodos de Component	**Función que realizan**
`boolean isVisible()` `void setVisible(boolean)`	Chequear o establecer la visibilidad de un componente
`boolean isShowing()`	Saber si un componente se está viendo
`isEnabled()` `setEnabled()`	Saber si un componente está activado y activarlo o desactivarlo
`Point getLocation()` `Point getLocationScreen()`	Obtener la posición de la esquina superior izquierda del componente
`setLocation(Point)` `setLocation(int x, int y)`	Desplazar un componente a la posición especificada respecto al container o pantalla

`Dimension getSize()`
`void setSize(int width, int height)`
`void setSize(Dimension d)`
	Obtener o establecer el tamaño de un componente

`Rectangle getBounds()`
`void setBounds(Rectangle)`
`void setBounds(int x,int y,`
`int width,int height)`
	Obtener o establecer posición y tamaño de un componente

`paint(Graphics)`
`repaint()`
`update(Graphics)`
	Métodos gráficos para dibujar en el componente

`setBackground(Color)` `setForeground(Color)`	Establecer los colores por defecto

11.4.2. Clases EventObject y AWTEvent

- Todos los métodos de las interfaces `Listener` relacionados con AWT tienen como argumento un único objeto de alguna clase que desciende de la clase `java.awt.AWTEvent`.
- La clase `AWTEvent` no define ningún método, aunque hereda el método `Object getSource()` de `EventObject`.

11.4.3. Clase ComponentEvent

- Los eventos de esta clase se generan cuando un `Component` se oculta, cambia de posición o tamaño.
- La clase `ComponentEvent` define el método `Component getComponent()` que devuelve el componente que generó el evento.

11.4.4. Clases InputEvent, MouseEvent

- La clase `InputEvent` es la superclase de los eventos de ratón y teclado.
- Esta clase define unas constantes para saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento: `SHIFT_MASK`, `ALT_MASK`, `CTRL_MASK`, `BUTTON1_MASK`, `BUTTON2_MASK`, `BUTTON3_MASK`.
- También dispone de métodos para detectar si los botones del ratón o las teclas especiales han sido pulsadas: `isShiftDown()`, `isAltDown`, `isControlDown()`, `getModifiers()`.
- Los eventos `MouseEvent` se producen cuando el cursor del ratón entra o sale de un componente, al hacer click, o cuando se pulsa o suelta un botón del ratón.
- Los métodos del interfaz `MouseListener` se relacionan con los eventos `MouseEvent`: `Point getPoint()`, `int getX()`, `int getY()`, etc.
- Los eventos `MouseEvent` disponen además de la interfaz `MouseMotionListener`, que define métodos relacionados con el movimiento o arrastre del ratón: `mouseMoved()` y `mouseDragged()`.

11.4.5. Clase FocusEvent

- El componente que tiene el ***Focus*** es el único que puede recibir acciones de teclado.
- Este componente aparece resaltado de alguna forma.
- El ***Focus*** se cambia con tecla ***Tab*** o con el ratón.
- Se produce un evento `FocusEvent` cada vez que un componente pierde o gana el ***Focus***.

11.4.6. Clase Container

- Es una clase muy general: normalmente no se crean objetos de esta clase, sino de sus clases derivadas (Frame, Dialog, Panel).
- Los containers mantienen una lista de los objetos que se le han ido añadiendo.
- Un objeto contenedor puede a su vez contener otros objetos contenedores.

Métodos de Container	**Función que realizan**

`void add(Component comp)`
`void add(Component comp,`
`Object constraint)`
	Añadir un componente al contenedor

`Object Component getComponent(int n)`	Obtener el componente `n`
`Component[] getComponents()`	Devolver un array con los componentes
`LayoutManager getLayout()`	Obtener el objeto gestor de posicionamiento
`void remove(int n)`	Eliminar el componente `n`
`void remove(Component comp)`	Eliminar un componente del contenedor
`void removeAll()`	Eliminar todos los componentes
`void setLayout(LayoutManager mgr)`	Activar `mgr` como gestor de posicionamiento

11.4.7. Clase ContainerEvent

- Se generan cada vez que un componente se añade o retira de un Container.
- Dispone de los métodos `Component getChild()` que nos devuelve el Component añadido o eliminado, y `Container getContainer()`, que devuelve el Container que generó el evento.

11.4.8. Clase Window

- Los objetos de esta clase son ventanas de máximo nivel, pero sin bordes y barra de menús.

11.4.9. Clase WindowEvent

- Se produce un `WindowEvent` cada vez que se abre, cierra, iconiza, restaura, activa o desactiva una ventana.
- La interfaz `WindowListener` contiene los siete métodos para tratar tales eventos.
- El que se utiliza más frecuentemente es el de cerrar una ventana: `void windowClosing(WindowEvent we)`

11.4.10. Clase Frame

(ver `AWT/java1312.java`)

- Es una ventana con borde y que puede tener barra de menús.
- Por defecto usa un BorderLayout.

1. Creación del frame:

   ```
   Frame miFrame=new Frame("Tutorial de Java, AWT");
   ```

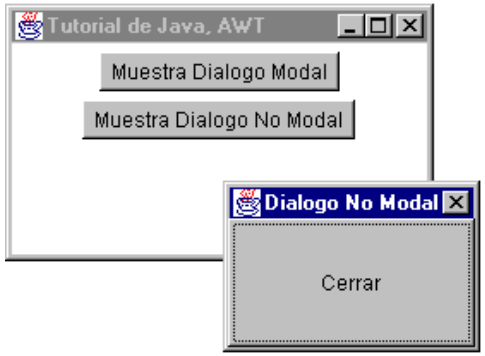
2. Se le pueden asociar iconos y distintos cursores.

   ```
   miFrame.setCursor(new Cursor(Cursor.HAND_CURSOR));
   miFrame.setIconImage(
           Toolkit.getDefaultToolkit().getImage("icono.gif");
   ```

Métodos	Función que realiza
`Frame()`, `Frame(String title)`	Constructores
`String getTitle()` `setTitle(String)`	Obtienen o determinan el título
`MenuBar getMenuBar()` `setMenuBar(MenuBar)` `remove(MenuComponent)`	Obtener, establecer o eliminar la barra de menús
`Image getIconImage()` `setIconImage(Image)`	Obtener o determinar el icono
`setResizable(boolean)` `boolean isResizable()`	Determinan o chequean si se puede cambiar el tamaño.
`dispose()`	Libera los recursos utilizados por el frame

11.4.11. Clase Dialog

- Un Dialog es una ventana que depende de otra ventana (Frame): si el Frame se cierra se cierran todos los Dialogs que dependen de ella.
- Existen dos tipos de diálogos: (ver ejemplos AWT/java1314.java AWT/java1315.java)
  - **Modales**: Todas las entradas del usuario serán recogidas por esa ventana, bloqueando cualquier entrada sobre otros objetos.
  - **No modal:** Es el tipo por defecto.



11.4.12. Clase FileDialog

- Muestra una ventana en la que se puede seleccionar un fichero.

11.4.13. Clase Panel

- Es un contenedor genérico de componentes.
- Puede incluso contener otros paneles.
- Por defecto usa un FlowLayout.

11.4.14. Clase Button

- De los seis tipos de eventos que puede recibir el más usado es el `ActionEvent`

Método de Button	**Función que realiza**
`Button(String)` `Button()`	Constructores
`addActionListener(ActionListener)` `removeActionListener(ActionListener)`	Añade o borra un receptor de eventos de tipo Action
`setLabel()` `getLabel()`	Establece o devuelve la etiqueta del botón
`setActionCommand(String)` `String getActionCommand()`	Establece y recupera un nombre de comando para el botón

Ejemplo de uso

```
Button boton1 = new Button();
boton1.setLabel("Mi boton");
boton2 = new Button("Otro boton");
receptoreventos=new MiActionListener();
boton1.addActionListener(receptoreventos);
boton2.addActionListener(receptoreventos);
class MiActionListener implements ActionListener{
   public void actionPerformed(ActionEvent e) {
      ....
   }
}
```

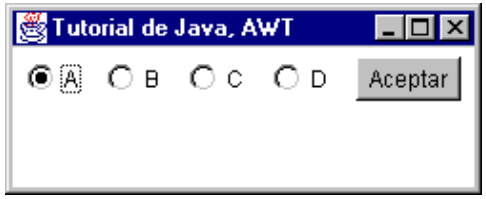
11.4.15. Clase ActionEvent

- Los eventos ActionEvent se producen al hacer click en un Button, al elegir una opción de un menú (MenuItem), al hacer doble click en un elemento de un List y al pulsar Intro en una caja de texto (TextField).
- El método `String getActionCommand()` devuelve el nombre de comando asociado con la acción que provocó el evento.
- El método `String getModifiers()` devuelve un entero que permite saber las teclas especiales que hay pulsadas al producirse el evento mediante las constantes `SHIFT_MASK`, `CTRL_MASK`, `META_MASK`, `ALT_MASK`.
  Ejemplo: `actionEvent.getModifiers() & ActionEvent.CTRL_MASK`

11.4.16. Clase Checkbox y CheckboxGroup

(ver ejemplo AWT/java1303.java)

- Son botones de opción o comprobación con dos posibles valores: on y off.
- Esta clase implementa el interfaz **ItemSelectable**.
- Al cambiar la selección de un Checkbox se produce un ItemEvent: Se ejecutará el método `itemStateChanged()` del receptor de eventos (`ItemListener`).
- Se suelen agrupar usando la clase **CheckboxGroup** (agrupaciones de botones de comprobación) para que uno y sólo uno esté seleccionado.
- El método `getSelectedChexbox()` de `CheckboxGroup` nos da el item seleccionado.



Ejemplo

```
CheckboxGroup miCheckboxGroup = new CheckboxGroup();
miFrame.add( new Checkbox( "A",true,miCheckboxGroup ) );
miFrame.add( new Checkbox( "B",false,miCheckboxGroup ) );
miFrame.add( new Checkbox( "C",false,miCheckboxGroup ) );
miFrame.add( new Checkbox( "D",false,miCheckboxGroup ) );
```

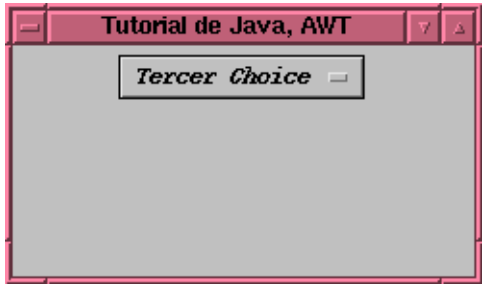
11.4.17. Clase ItemEvent

- Se produce un `ItemEvent` cuando ciertos componentes (`Checkbox`, `CheckboxMenuItem`, `Choice` y `List`) cambian de estado (on/off).
- Estos componentes son los que implementan el interfaz `ItemSelectable`.
- El método `ItemSelectable getItemSelectable()` devuelve el objeto que originó el evento.
- `int getStateChange()` devuelve `SELECTED` o `DESELECTED`.

11.4.18. Clase Choice

(ver ejemplo AWT/java1302.java)

- Similares a los `Checkbox`, permiten elegir un item de una lista desplegable.
- Los métodos `add(String)` y `addItem(String)` añaden items a la lista.
- Los métodos `String getSelectedItem()` y `int getSelectedIndex()` permiten obtener el item seleccionado.



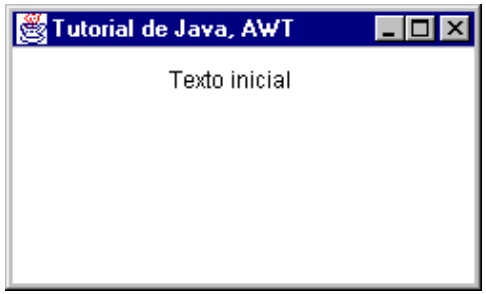
Ejemplo

```
Choice choice;
choice = new Choice();
choice.add("Primer Choice");
choice.add("Segundo Choice");
choice.add("Tercer Choice");
choice.select("Tercer Choice");
```

11.4.19. Clase Label

(ver ejemplo AWT/java1307.java)

- Introduce un texto no seleccionable y no editable.



Ejemplo

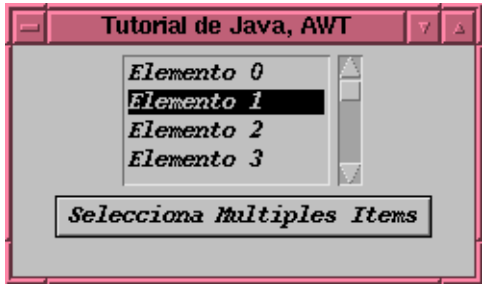
```
Label miEtiqueta=new Label ("Texto inicial");
```

11.4.20. Clase List

(ver ejemplo AWT/java1304.java)

- Definida por una zona de pantalla de varias líneas, de las que se muestran sólo algunas, y entre las que se puede hacer una selección simple o múltiple.
- Generan eventos de la clase `ActionEvent` al hacer doble click, e `ItemEvents` al seleccionar o deseleccionar.
- `String getSelectedItem()` devuelve el item seleccionado.

- `String[] getSelectedItems()` devuelve un array de los items seleccionados cuando hay varios seleccionados.



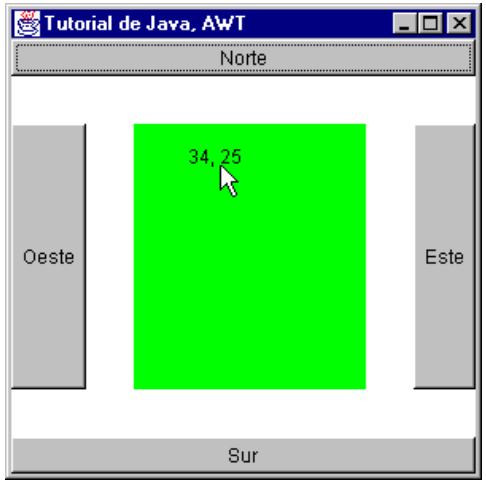
Ejemplo

```
List miLista = new List();
for( int i=0; i < 15; i++ )
    miLista.add( "Elemento "+i );
miLista.setMultipleMode( true );
miLista.select( 1 );
```

11.4.21. Clase Canvas

(ver ejemplo AWT/java1308.java)

- Un Canvas es una zona rectangular en la que se puede dibujar y en la que se pueden generar eventos.
- El método `void paint(Graphics)` debe ser sobreescrito y sirve para repintar el canvas.
- El método `void repaint()` se usa en los programas para llamar a `paint()` y así repintar el canvas.
- Esta clase no tiene eventos propios, pero puede recibir los eventos `ComponentEvent` de la superclase `Component`.



Ejemplo

```
class MiCanvas extends Canvas {
  int posicionX;
  int posicionY;
  public MiCanvas() {
    this.setBackground( Color.green );
  }
  public void paint( Graphics g ) {
    g.drawString( "" + posicionX + ", " + posicionY,
      posicionX,posicionY );
  }
}
MiCanvas miObjCanvas = new MiCanvas();
```

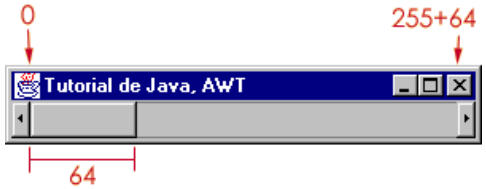
11.4.22. Clase Scrollbar: Barras de desplazamiento

- Es una barra de desplazamiento con un cursor que permite introducir y modificar valores, entre unos valores mínimo y máximo, con pequeños y grandes incrementos.
- Se pueden utilizar aisladas, o bien unidas a una ventana para mostrar una parte de su información.

- Los métodos `int getValue()`, `setValue(int)` permiten obtener y fijar el valor.

Ejemplo AWT/java1309.java

Ejemplo AWT/java1310.java



Ejemplo AWT/java1311.java

Ejemplo

```
Scrollbar scroll = new Scrollbar(Scrollbar.HORIZONTAL,0,64,0,255 );
```

11.4.23. Clase AdjustmentEvent

- Se produce este evento cada vez que se cambia el valor de un `Scrollbar`.

11.4.24. Clase ScrollPane

- Es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos `Scrollbars`, uno horizontal y otro vertical.
- El componente puede ser por ejemplo una imagen.

11.4.25. Clases TextArea y TextField

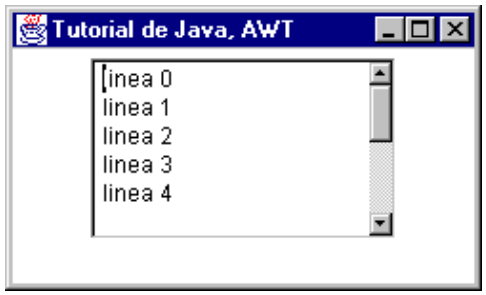
- Ambas heredan de la clase `TextComponent` y muestran texto seleccionable y editable.
- `String getSelectedText()` devuelve el texto seleccionado.
- `String getText()` devuelve el contenido completo.
- `TextField` sólo tiene una línea, mientras que `TextArea` puede tener varias.
- Un `TextField` puede generar eventos `ActionEvent` al pulsar Intro.
- Ambos objetos pueden generar eventos `TextEvent` que se producen cada vez que se modifica el texto del componente.
- También pueden recibir `FocusEvent`, `MouseEvent` y `KeyEvent`.

Ejemplo de creación de campo de texto (ver ejemplo java1305.java)

```
TextField miCampoTexto = new TextField( "Texto inicial" );
```

Ejemplo de creación de un área de texto (ver ejemplo AWT/java1306.java)

```
TextArea miAreaTexto = new TextArea( "",5,20,
         TextArea.SCROLLBARS_VERTICAL_ONLY );
for( int i=0; i < 10; i++ )
  miAreaTexto.append( "linea "+i+"\n" );
```

11.4.26. Clase TextEvent

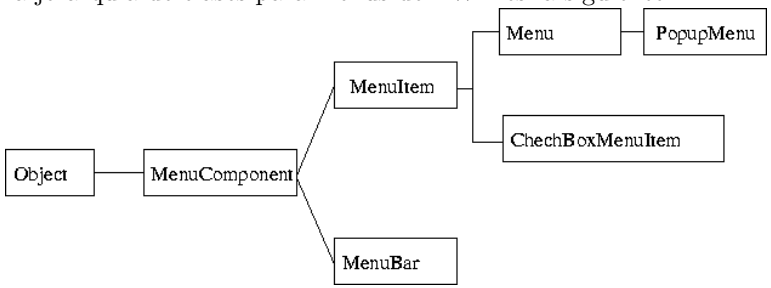
- Se produce un `TextEvent` cada vez que se cambia algo en un `TextComponent` (`TextArea` o `TextField`)
- La interfaz receptora de eventos `TextListener` contiene el método `void textValueChanged(TextEvent)` para gestionar estos eventos.

11.4.27. Clase KeyEvent

- Se produce un `KeyEvent` al pulsar sobre teclado.
- Es el objeto que tiene el focus quien genera estos eventos.
- Hay dos tipos de `KeyEvent`:
  1. **key-typed**, que representa la introducción de carácter Unicode.
  2. **key-pressed** y **key-released**, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como F1.

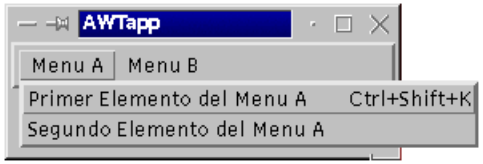
11.5. Menús

- La jerarquía de clases para menús de AWT es la siguiente:



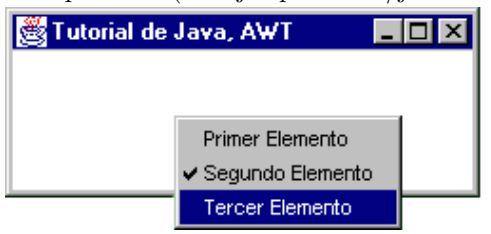
- Tienen un comportamiento similar a los componentes, pues aceptan `Events`
- Para crear un `Menu` se debe crear primero un `MenuBar`; después se crean los Menús que se insertan en el `MenuBar`; luego se añaden los `MenuItem` a cada Menú.
- El `MenuBar` será añadido a un `Frame`.
- Puede también añadirse un `Menu` a otro `Menu` para crear submenús.

1. **Clase MenuComponent:** Superclase de todos los componentes relacionados con menús.
2. **Clase Menu:** Componente de una barra de menú. (ver ejemplo AWT/java1317.java)
3. **Clase MenuItem:** Una opción de un menú. (ver ejemplo AWT/java1317.java)
4. **Clase MenuShortcut:** Acelerador de teclado (combinación de teclas) para acceder a un MenuItem.
5. **Clase Menubar:** Encapsula el concepto de barra de menú de un Frame. (ver ejemplo AWT/java1317.java)



6. **Clase CheckboxMenuItem:** Caja de selección que representa una opción de un menú. (ver ejemplo AWT/java1318.java)

7. **Clase PopupMenu:** Menú que puede ser presentado dinámicamente dentro de un Componente. (ver ejemplo AWT/java1319.java)



11.6. Layouts (gestores de posicionamiento)

Ayudan a adaptar los diversos componentes que se desean incorporar a un contenedor. El método `setLayout()` permite especificar un layout en un contenedor.

1. **FlowLayout:** Los elementos se colocan de izquierda a derecha, y de arriba hacia abajo. (ver ejemplo AWT/java1322.java)
2. **BorderLayout:** Divide el contenedor en 5 áreas: North, South, East, West y Center (ver ejemplo AWT/java1323.java y AWT/java1324.java)
3. **CardLayout:** Se usa cuando se quiere que en una zona del contenedor se puedan poner distintos componentes según la situación del programa. (ver ejemplo AWT/java1327.java)
4. **Posicionamiento absoluto:** Mediante los métodos *setBounds(int x ,int y ,int ancho ,int alto)* y *setBounds(Rectangle)* se puede especificar la posición y tamaño de un componente en coordenadas absolutas en pixels. (ver ejemplo AWT/java1328.java)
5. **GridLayout:** Organiza los componentes según una cuadrícula de dos dimensiones. (ver ejemplo AWT/java1325.java)
6. **GridBagLayout:** Igual a anterior, con la diferencia que los componentes no necesitan tener el mismo tamaño. (ver ejemplo AWT/java1326.java)

11.7. Dibujando con AWT

(ver `http://java.sun.com/products/jfc/tsc/articles/painting/index.html`)

- La *hebra despachadora de eventos* (event dispatch thread) se encarga de repintar los componentes cuando se solicita en nuestro código.
- En AWT hay dos mecasimos por los que se producen las operaciones de repintado, dependiendo de quién sea el que ordene ese repintado, el *sistema* o la *aplicación*.
  - **Sistema**: él es quien indica a un Componente que debe regenerar su contenido debido a:
    - El componente se hace visible por primera vez.
    - El componente ha cambiado de tamaño.
    - El componente se ha deteriorado y necesita ser regenerado (por ejemplo, porque se oculta otra ventana que estaba encima).
  - **Aplicación**: El propio componente decide la necesidad de actualización, normalmente debido a algún cambio en su estado interno (por ejemplo un botón detecta que el ratón ha sido pulsado sobre él y debe pasar de estado normal a estado pulsado).

11.7.1. Métodos para repintado

Los siguientes métodos de la clase `javax.awt.Component` están involucrados en el repintado de componentes:

- `void paint(Graphics g)`:
  - Es llamado por la *event dispatch thread* para repintar el componente por primera vez y cada vez que necesita repintarse.
  - Podemos sobreescribirlo para dibujar nuestros propios contenidos en el componente.
  - Se usarán el objeto **Graphics** (o uno derivado de él) para dibujar en el componente.
  - Cuando AWT llama a `paint(Graphics g)` el objeto **Graphics** está preconfigurado de la siguiente forma:

- El *color* del objeto Graphics se fija a la propiedad **foreground** del Componente.
- La *fuente de caracteres* se fija a la propiedad **font** del Componente.
- La *traslación* también se determina, teniendo en cuenta que la coordenada (0,0) representa la esquina superior-izquierda del Componente.
- El *rectángulo de recorte*, o clipping, se fija al área del Componente que es necesario repintar.

Ejemplo de paint(): AWT/java1502.java

```
public void paint( Graphics g ) {
  // Se calcula el tamaño de la zona
  Dimension tam = getSize();
  // Ahora el diámetro
  int d = Math.min( tam.width,tam.height );
  int x = (tam.width - d) / 2;
  int y = (tam.height - d) / 2;

  // Se pinta el círculo, fijando el color al de frente
  g.fillOval( x,y,d,d );
  // Se pinta la circunferencia de borde, en color negro
  g.setColor( Color.black );
  g.drawOval( x,y,d,d );
}
```

- `void update(Graphics g)`:
  - La implementación por defecto para **componentes pesados** (como por ejemplo Canvas) redibuja el componente con el color de fondo y luego llama a `paint()`.
  - En **componenentes ligeros** (por ejemplo clase Component) no se borra el fondo.

- `void repaint()`, `void repaint(long tm)`, `void repaint(int x, int y, int w, int h)`, `void repaint(long tm,int x, int y, int w, int h)`: Por defecto llaman a `update()` para redibujar el componente.
  Son los métodos a los que deben llamar nuestros programas para redibujar el componente.

11.7.2. Dibujando componentes pesados

La forma en que se dibujan los componentes pesados depende de quien lo ordena:

- **Sistema**:
  - AWT determina si el componente necesita ser repintado completamente o solamente parte de él.
  - AWT llama al método `paint()` sobre el componente.

  Resumen de métodos involucrados

`paint()`

- **Aplicación**:
  - El programa determina si parte o todo el componente debe ser repintado, en respuesta a cambios en algún estado interno.
  - El programa invoca al método `repaint()` sobre el componente, el cual lanza una petición a AWT indicándole que ese componente necesita ser repintado.
  - AWT hace que la *hebra despachadora de eventos* llame a `update()` sobre el componente.

    En el caso de que se produzcan **múltiples llamadas** al método `repaint()` antes de que se inicie el repintado, todas se reunirán en una sola; el algoritmo para decidir cuándo reunir todas las llamadas a `repaint()` en una sola es dependiente de la implementación.
    Cuando se reunen las llamadas a `repaint()`, la zona que se actualizará será la unión de los rectángulos indicados en cada una de las peticiones de repintado.

- Si el componente no sobrecarga el método `update()`, su implementación por defecto limpia el fondo del componente (si no se trata de un componente lightweight) y luego hace una llamada a `paint()`.

Resumen de métodos involucrados

```
repaint()  -->  update()  -->  paint()
```

Ejemplo de uso con *dibujo incremental*: (AWT/java1503.java).

Sobreescribe `update()` en clase `MiCanvasSuave` para que dibuje sólo los nuevos segmentos dibujados.

- **Aplicación**: Al pinchar con ratón (añadir un nuevo segmento) se llama a `repaint() --> update()`: *dibujo incremental*.
- **Sistema**: Cuando el sistema necesite repintar el Canvas llama a `paint()`: se dibuja todo.

11.7.3. Dibujando componentes ligeros

- La forma en que se dibujan los componenentes ligeros es básicamente la misma que los pesados:

  Nuestras aplicaciones sobreescriben `paint()` y llaman a `repaint()`

- Pero hay alguna diferencia:
  - Para que exista un componente ligero debe existir algún componente pesado en la jerarquía que lo contenga.
  - Cuando tal componente pesado se deba redibujar, hará llamadas al método `paint()` de todos los componentes ligeros que contenga: esto se hace en el método `paint()` de `java.awt.Container`

    Tal método llama al método `paint()` de los componentes que estén visibles y que intersecten con el rectángulo a pintar.

- Por ello es necesario que las subclases de `Container` (ligeras y pesadas) que sobreescriban `paint()` hagan lo siguiente:

```
public class MiContainer extends Container {
   public void paint(Graphics g) {
       // dibujar mis contenidos ...
       // y luego, dibujar los componentes ligeros
       super.paint(g);
   }
}
```

Dibujo de componentes ligeros ordenadas por el sistema

Una orden del **sistema** de dibujar un componente ligero puede involucrar los siguientes métodos:

- Petición de dibujo originada en el *sistema nativo*

  ```
  paint()
  ```

  Ejemplo: El componente pesado antecesor del ligero se muestra por primera vez

- Petición de dibujo originada en el *marco de componentes ligeros*

  ```
  repaint()  -->  update()  -->  paint()
  ```

  Ejemplo: El componente ligero se redimensiona.

Componentes ligeros y transparencia

- Los componentes ligeros soportan la *transparencia*, ya que son pintados de atrás hacia delante.
- Si un componente ligero deja alguno o todos su pixels sin dibujar, entonces será visible el componente que haya debajo (por encima en el árbol jerárquico).
- Esta es la razón por la que la implementación por defecto de update() no limpia el fondo en los componentes lightweight.

(Ver por ejemplo **Ejemplo de trasparencia: AWT/java1504.java**)

11.7.4. Animaciones

Para realizar la animación de algún objeto haremos lo siguiente:

- Construiremos una hebra en la que el método `run()` incluirá un *bucle de animación*

```
while (/* la hebra de animacion se esté ejecutando */) {
    Avanzar el objeto a animar
    repaint(); // Mostrarlo
    Thread.sleep(tiempo); // Delay dependiente de velocidad deseada
}
```

(Ver como ejemplo **Programa serpienteAWT/**)

11.7.5. Eliminación del parpadeo

- Con frecuencia, se produce *parpadeo* al llamar a `repaint()`.
  (Ver como ejemplo **AWT/FlashingGraphics.java**
- El parpadeo se debe al borrado que hace `update()` del componente (por defecto `repaint()` llama a `update()` y éste a `paint()`).
- Soluciones al parpadeo:
  - **Sobreescribir** `update()` para que contenga sólo una llamada a `paint()`.
    (Ver como ejemplo **AWT/Update.java**)
  - Implementar **doble buffer**
    (Ver como ejemplo **AWT/DoubleBuffer.java** o **Programa serpienteAWT**)

11.7.6. Mover imágenes

(Ver como ejemplo **AWT/MovingImages.java**)

11.7.7. Mostrar una secuencia de imágenes

(Ver como ejemplo **AWT/ImageSequence.java**)

11.7.8. Clase java.awt.Graphics

Contiene métodos para dibujar líneas, rectángulos, arcos ... así como información relativa a características de dibujo tales como el componente donde se va a dibujar, color, font, zona de recorte, función (XOR o Paint)

- `drawLine(int x1,int y1,int x2,int y2)`
- `drawRect(int x1,int y1,int w,int h)`
- `fillRect(int x1,int y1,int w,int h)`
- `clearRect(int x1,int y1,int w,int h)`
- `draw3DRect(int x1,int y1,int w,int h,boolean raised)`
- `fill3DRect(int x1,int y1,int w,int h,boolean raised)`
- `drawRoundRect(int x1,int y1,int w,int h,int arch)`
- `fillRoundRect(int x1,int y1,int w,int h,int arch)`
- `drawOval(int x1,int y1,int w,int h)`
- `fillOval(int x1,int y1,int w,int h)`
- `drawArc(int x1,int y1,int w,int h,`
  `int startAngle,int arcAngle)`
- `drawPolygon(int x[],int y[],int nPoints)`
- `drawPolyline(int x[],int y[],int nPoints)`
- `fillPolygon(int x[],int y[],int nPoints)`
- `drawBytes(byte data[],int offset,int lenght, int x, int y)`
- `drawChars(byte data[],int offset,int lenght, int x, int y)`
- `drawString(String str,int x, int y)`

11.7.9. Clase java.awt.Graphics2D

Esta clase extiende a `Graphics` incrementando sus prestaciones.
Para usarla en `paint()` y `update()` hay que hacer un casting:

```
public void paint (Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```


12. Swing

12.1. Introducción a las Java Foundation Classes (JFC)

- Conjunto de clases para crear interfaces gráficos de usuario
- Sus principales componentes son:
  - **AWT**: Toolkit original de Java para desarrollar GUIs.
  - **Componentes Swing**: Extensión de **AWT** que permite incorporar en las aplicaciones elementos gráficos de una forma mucho más versátil y con más capacidades que **AWT**
  - **Java2D**: Permite incorporar texto, imágenes y gráficos en dos dimensiones de gran calidad, además de soporte para imprimir documentos complejos.
  - **Accessibility**: Interfaz que suministra tecnologías tales como lectores de pantalla, software de reconocimiento del habla ...
  - **Drag and Drop**: Permite mover datos entre programas creados en Java.
  - **Internationalization**: Permite que una aplicación pueda usarse en varios idiomas.
  - **Pluggable Look-and-Feel Support**: Permite elegir aspecto-comportamiento.

12.2. Introducción a Swing

- Cualquier programa construido con **Swing** puede elegir el aspecto (look-and-feel) que desea para sus ventanas y elementos gráficos.
- Los componentes presentan más propiedades que los correspondientes en **AWT**
- La mayoría de los componentes son ligeros (lightweight), o sea que no dependen de código nativo (son `100%` Java puros):
  - Consumen menos recursos del sistema.

  - Se comportan igual en distintas plataformas.
  - Los componentes pesados (heavyweight) son: **JApplet**, **JDialog**, **JFrame** y **JWindow**.

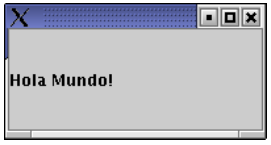
- **Swing** está compuesto de más de 250 clases y 75 interfaces agrupados en las siguientes categorías:
  - Componente **JComponent** que extiende a **java.awt.Container** y un conjunto de componentes (del paquete **javax.swing**) subclase de **JComponent** (que comienzan con la letra J):
    Todos los componentes Swing son contenedores y son descendientes de **java.awt.Component**
  - Clases de objetos no visibles tales como clases de eventos. (No comienzan con la letra J)
  - Conjunto de interfaces que son implementados por sus componentes y clases soporte.
- Las aplicaciones usan los componentes **Swing** y **AWT** de la misma forma: mismos manejadores de posicionamiento (layout managers), objetos de eventos, receptores de eventos.
- Diferencia: el *content-pane* en clases **JFrame**, **JApplet**, **JDialog** y **JWindow**.
  - Ahora no se usa `add` para añadir componentes directamente a un contenedor que implemente el interfaz **RootPaneContainer**.
  - Tales contenedores tienen un contenedor *content-pane* donde se colocan los componentes.
  - El interfaz **RootPaneContainer** define el método `getContentPane()` para acceder al objeto *content-pane* (de la clase **JRootPane**).
    Por defecto el *content-pane* de un **JFrame** usa **BorderLayout**.
- Otra diferencia: **TextArea** y **List** disponen de barras de scroll, pero **JTextArea** y **JList** deben ser colocados en un contenedor **JScrollPane** para que la tengan.

12.3. Primeros programas con Swing

Programa Hola Mundo: Swing/P1/JHolaMundo.java

```
import javax.swing.*;
public class JHolaMundo extends JFrame {
  public static void main( String argv[] ) {
    new JHolaMundo();
  }
  JHolaMundo() {
    addWindowListener(new Proceso1());
    JLabel hola = new JLabel( "Hola Mundo!" );
    getContentPane().add( hola,"Center" );
    setSize( 200,100);
    setVisible( true );
  }
}

class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```



Otro ejemplo: Swing/java1401.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class java1401 extends JPanel {
  JButton boton1 = new JButton( "JButton 1" );
  JButton boton2 = new JButton( "JButton 2" );
  JTextField texto = new JTextField( 20 );
  public java1401() {
    ActionListener al = new ActionListener() {
      public void actionPerformed( ActionEvent evt ) {
        String nombre = ( (JButton)evt.getSource()).getText();
          texto.setText( nombre+" Pulsado" ); } };
    boton1.addActionListener( al );
    boton1.setToolTipText( "Soy el JBoton 1" );
    add( boton1 );
    boton2.addActionListener( al );
    boton2.setToolTipText( "Soy el JBoton 2" );
    add( boton2 );
    texto.setToolTipText( "Soy el JCampoDeTexto" );
    add( texto );
  }
  public static void main( String args[] ) {
    JFrame ventana = new JFrame( "Tutorial de Java, Swing" );
    ventana.addWindowListener( new WindowAdapter() {
            public void windowClosing( WindowEvent evt ){
                System.exit( 0 );
                }
            } );
    ventana.getContentPane().add( new java1401(),
                                  BorderLayout.CENTER );
    ventana.setSize( 300,100 );
    ventana.setVisible( true );
  }
}
```


12.4. Clase javax.swing.JComponent

- `void addxxxListener( xxxListener object)`: Para registrar el receptor de eventos para un tipo específico de evento.
- `void repaint()`
  `void repaint(long msec)`
  `void repaint(int x, int y, int height, int width)`
  `void repaint(long msec, int x, int y, int height, int width)`: Para repintar el componente.
- `void setBackground(Color c)`: Para poner el color del background del componente.
- `void setBorder(Border b)`: Para añadir un borde al componente (por defecto no lo tienen). Ver clase **javax.swing.BorderFactory** para crear un borde.
- `void setDoubleBuffered(boolean b)`: Para usar doble buffering (permite eliminar el parpadeo de pantalla). Por defecto está activo.
- `void setEnabled(boolean b)`: Para activar o desactivar interacción con usuario. Activo por defecto.
- `void setFont(Font f)`: Para especificar un font para el texto del componente.
- `void setForeground(Color c)`: Para especificar el color de foreground del componente.
- `void setPreferredSize(Dimension d)`: Para especificar el tamaño ideal para el componente. Los manejadores de posicionamiento usan este tamaño cuando colocan el componente.
- `void setSize( Dimension d)`: Para dar el tamaño en pixels.

- `void setToolTipText( String s)`: Para asociar un string con el componente que se muestra al colocar el puntero del ratón sobre el componente.
- `void setVisible( boolean b)`: Para hacer que sea o no visible.
- `void update( Graphics context)`: Este método llama a `paint` para repintar el componente.

12.5. Clases contenedoras

Todos los componentes Swing heredan de `java.awt.Container` pero sólo ciertos componentes pueden contener otros componentes Swing.

12.5.1. Contenedores pesados

- Son los contenedores de alto nivel en una aplicación: JApplet, JDialog, JFrame y JWindow.
- Clase `javax.swing.JDialog`:
  Se suelen usar para solicitar alguna entrada del usuario.
  Extiende a `java.awt.Dialog`.
  Usa por defecto BorderLayout.
  - `JDialog(Frame parent)`
    `JDialog(Frame parent, boolean modal)`
    `JDialog(Frame parent, String title)`
    `JDialog(Frame parent, String title, boolean modal)`:
    Con `modal=true` bloquea la interacción con otras ventanas mientras el diálogo está visible.
- Clase `javax.swing.JFrame`:
  Es una ventana con bordes y barra de títulos (y puede que también una barra de menús).
  Extiende a `java.awt.Frame`.
  Usa por defecto BorderLayout.
  - `JFrame()`
  - `JFrame(String title)`

- Clase `javax.swing.JWindow`:
  Son ventanas vacías que no tienen título ni barra de menús.
  Se usa para crear componentes optimizados.
  Usa por defecto BorderLayout.
  - `JWindow(Frame parent)`

12.5.2. Contenedores ligeros

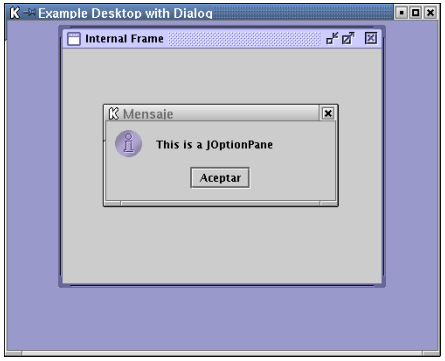
- Clase `javax.swing.JDestopPane`: Es un contenedor para objetos `JInternalFrame`
  Usa por defecto un manejador de posicionamiento nulo.
  - `JDesktopPane()`
- Clase `javax.swing.JInternalFrame`:
  - `JInternalFrame()`
  - `JInternalFrame(String title)`
  - `JInternalFrame(String title,boolean resizable)`
  - `JInternalFrame(String title,boolean resizable, boolean closable)`
  - `JInternalFrame(String title,boolean resizable, boolan closable,boolean maximizable)`
  - `JInternalFrame(String title,boolean resizable, boolan closable,boolean maximizable,boolean iconifiable)`

- Clase `javax.swing.JOptionPane`: Proporciona una forma sencilla para crear y mostrar la mayoría de los diálogos más comunes.
  - `JOptionPane()`
  - `JOptionPane(Object message)`
  - `JOptionPane(Object message, int messageType)`
  - `JOptionPane(Object message, int messageType, int optionType)`
  - `JOptionPane(Object message, int messageType, int optionType, Icon icon)`
  - `JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options)`
  - `JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options, Object initialValue)`
- Clase `javax.swing.JPanel`:
  Por defecto usa FlowLayout.
  - `JPanel()`
  - `JPanel(boolean isDoubleBuffered)`
  - `JPanel(LayoutManager layout)`
  - `JPanel(LayoutManager layout, boolean isDoubleBuffered)`

Otro programa de ejemplo: Swing/P3/DesktopAndDialog.java

```
import javax.swing.*;
import java.awt.*;
/** Clase que demuestra el uso de JDesktopPane,
  * JInternalFrame y JOptionPane
  */
public class DesktopAndDialog extends JFrame {
   private static final boolean RESIZABLE = true;
   private static final boolean CLOSABLE = true;
   private static final boolean MAXIMIZABLE = true;
   private static final boolean ICONIFIABLE = true;
   private static final boolean MODAL = false;
   /** constructor
     * @param titleText Barra de título de la ventana
     */
   public DesktopAndDialog( String titleText ) {
      super( titleText );
      addWindowListener( new WindowCloser() );
      JInternalFrame ifrm = new JInternalFrame(
         "Internal Frame",
         RESIZABLE, CLOSABLE, MAXIMIZABLE, ICONIFIABLE );
      ifrm.setPreferredSize( new Dimension( 375, 300 ) );
      JDesktopPane dt = new JDesktopPane();
      dt.setLayout( new FlowLayout() );
      dt.add( ifrm );
      getContentPane().add( dt, BorderLayout.CENTER );
      setSize( 500, 400 );
      setVisible( true );
      JOptionPane.showMessageDialog(
         ifrm, "This is a JOptionPane" );
   }
```

```
   /** El main
     * @param args no usado
     */
   public static void main( String[] args ) {
      new DesktopAndDialog(
         "Example Desktop with Dialog" );
   }
}
```



12.6. Etiquetas, botones y cajas de comprobación (check)

- Clase `javax.swing.JLabel`: Objeto con una línea de texto de solo lectura.
  Extiende a `JComponent`
  - `JLabel()`
  - `JLabel(Icon icon)`
  - `JLabel(Icon icon, int horizontalAlignment)`
  - `JLabel(String text)`
  - `JLabel(String text,int horizontalAlignment)`
  - `JLabel(String text,Icon icon,int horizontalAlignment)`

- Clase `javax.swing.JButton`: Extiende a `AbstractButton` que a su vez extiende a `JComponent`
  - `JButton()`
  - `JButton(Icon icon)`
  - `JButton(String label)`
  - `JButton(String label, Icon icon)`
- Clase `javax.swing.JToggleButton`: Botón que puede estar en dos estados: seleccionado o no.
  Extiende a `AbstractButton` que a su vez extiende a `JComponent`
  - `JToggleButton()`
  - `JToggleButton(Icon icon)`
  - `JToggleButton(String label)`
  - `JToggleButton(String label, Icon icon)`
  - `JToggleButton(Icon icon,boolean pressed)`
  - `JToggleButton(String label,boolean pressed)`
  - `JToggleButton(String label, Icon icon,boolean pressed)`
- Clase `javax.swing.JCheckBox`:
  Extiende a `JToggleButton`
  - `JCheckBox()`
  - `JCheckBox(Icon icon)`
  - `JCheckBox(String label)`
  - `JCheckBox(String label, Icon icon)`
  - `JCheckBox(Icon icon,boolean selected)`
  - `JCheckBox(String label,boolean selected)`
  - `JCheckBox(String label, Icon icon,boolean selected)`

  (ver Swing/java1404.java)

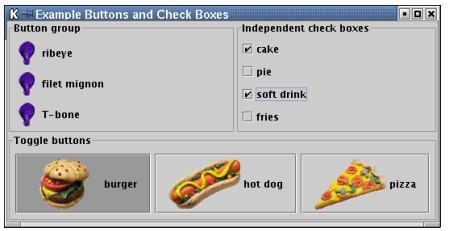
- Clase `javax.swing.JRadioButton`:
  Extiende a `JToggleButton`
  - `JRadioButton()`
  - `JRadioButton(Icon icon)`
  - `JRadioButton(String label)`
  - `JRadioButton(String label, Icon icon)`
  - `JRadioButton(Icon icon,boolean selected)`
  - `JRadioButton(String label,boolean selected)`
  - `JRadioButton(String label, Icon icon,boolean selected)`
- Clase `javax.swing.ButtonGroup`: Objeto para contener un grupo de botones (`JToggleButton`, `JRadioButton` o `JCheckBox`) mutuamente exclusivos.
  El método `buttongroup.add(boton)` añade un botón al grupo.
  - `ButtonGroup()`

Ejemplo de botones y grupos: Swing/P4/ButtonsAndBoxes.java

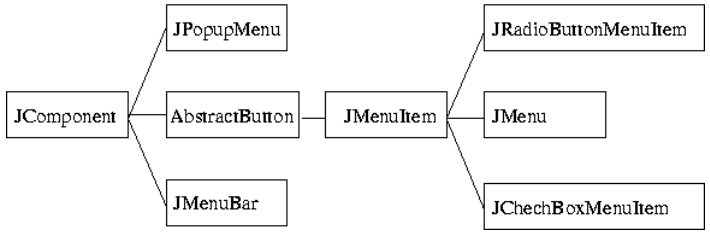
```
import javax.swing.*;
import java.awt.*;
import java.util.Enumeration;
/** Ejemplo para demostrar el uso de los componentes
  * JToggleButton, JCheckBox, and JRadioButton
  */
public class ButtonsAndBoxes extends JFrame {
   /** Class constructor method
     * @param titleText Window's title bar text
     */
   public ButtonsAndBoxes( String titleText ) {
      super( titleText );
      addWindowListener( new WindowCloser() );

      JPanel left = new JPanel( new GridLayout( 0, 1 ) );
      left.setBorder(
         BorderFactory.createTitledBorder(
            "Button group" ) );
      ButtonGroup bg = new ButtonGroup();
      bg.add( new JRadioButton( "ribeye" ) );
      bg.add( new JRadioButton( "filet mignon" ) );
      bg.add( new JRadioButton( "T-bone" ) );
      Enumeration e = bg.getElements();
      while( e.hasMoreElements() ) {
         JRadioButton rb = (JRadioButton) e.nextElement();
         rb.setIcon( new ImageIcon( "bulb1.gif" ) );
         rb.setSelectedIcon(
            new ImageIcon( "bulb2.gif" ) );
         left.add( rb );
      }
      JPanel right = new JPanel( new GridLayout( 0, 1 ) );
      right.setBorder(BorderFactory.createTitledBorder(
            "Independent check boxes" ) );
```

```
        right.add( new JCheckBox( "cake" ) );
        right.add( new JCheckBox( "pie" ) );
        right.add( new JCheckBox( "soft drink" ) );
        right.add( new JCheckBox( "fries" ) );
        JPanel bottom = new JPanel();
        bottom.setBorder(BorderFactory.createTitledBorder(
              "Toggle buttons" ) );
        bottom.add( new JToggleButton(
           "burger", new ImageIcon( "burger.gif" ) ) );
        bottom.add( new JToggleButton(
           "hot dog", new ImageIcon( "hotdog.gif" ) ) );
        bottom.add( new JToggleButton(
           "pizza", new ImageIcon( "pizza.gif" ) ) );
        Container cp = getContentPane();
        cp.setLayout( new GridBagLayout() );
        GridBagConstraints c = new GridBagConstraints();
        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;
        c.weighty = 1.0;
        cp.add( left, c );
        c.gridwidth = GridBagConstraints.REMAINDER;
        cp.add( right, c );
        cp.add( bottom, c );
        pack();
        setVisible( true );
    }
    /** El metodo main
      * @param args no usado
      */
    public static void main( String[] args ) {
        new ButtonsAndBoxes(
           "Example Buttons and Check Boxes" );
    }
}
```

12.7. Menús, barras de utilidades y acciones



- Clase `javax.swing.JMenuBar`. Barra de menús de una aplicación. Usar método `setJMenuBar(menuBar)` para ponerla en una ventana.
  - `JMenuBar()`
- Clase `javax.swing.JMenuItem`: Items de un menú. Puede ponerse un nemónico o acelerador con `setMnemonic` y `setAccelerator`
  - `JMenuItem()`
  - `JMenuItem(Icon icon)`
  - `JMenuItem(String label,Icon icon)`
  - `JMenuItem(String label,int mnemonic)`
- Clase `javax.swing.JMenu`: Un menú. Un `JMenu` es a su vez un `JMenuItem`.
  - `JMenu()`
  - `JMenu(String label)`
  - `JMenu(String label, boolean tearoff)`

- Clase `javax.swing.JPopupMenu`: Menús popup que aparecen al hacer click en un componente.
  El menú se asocia al componente con `componente.add(popupmenu)`
  - `JPopouMenu()`
  - `JPopupMenu(String name)`
- Clase `javax.swing.JCheckBoxMenuItem`: Items tipo check box.
  - `JCheckBoxMenuItem()`
  - `JCheckBoxMenuItem(Icon icon)`
  - `JCheckBoxMenuItem(String label)`
  - `JCheckBoxMenuItem(String label, Icon icon)`
  - `JCheckBoxMenuItem(String label, boolean selected)`
  - `JCheckBoxMenuItem(String label, Icon icon, boolean selected)`
- Clase `javax.swing.JRadioButtonMenuItem`: Items tipo botón radio.
  - `JRadioButtonMenuItem()`
  - `JRadioButtonMenuItem(Icon icon)`
  - `JRadioButtonMenuItem(Icon icon,boolean selected)`
  - `JRadioButtonMenuItem(String label)`
  - `JRadioButtonMenuItem(String label, Icon icon)`
  - `JRadioButtonMenuItem(String label, boolean selected)`
  - `JRadioButtonMenuItem(String label, Icon icon, boolean selected)`
- Clase `javax.swing.JToolBar`:
  - `JToolBar()`
  - `JToolBar(int orientation)`

- Clase `javax.swing.AbstractAction`: Permite definir una misma acción para varios controles.
  Swing proporciona el interfaz `javax.swing.Action` (que extiende a `ActionListener`) para definir el mismo receptor de eventos `ActionEvent` con varios controles.
  Esto permite desactivar o activar el receptor para todos los objetos que lo tengan registrado.
  Además se permite dar la etiqueta e icono para estos controles.
  `javax.swing.AbstractAction` implementa este interfaz `Action`. En nuestro programa extenderemos esta clase implementando el método `actionPerformed(ActionEvent event)`
  - `AbstractAction()`
  - `AbstractAction(String name)`
  - `AbstractAction(String name, Icon icon)`

Programa de ejemplo: Swing/P5/MenusToolbar.java

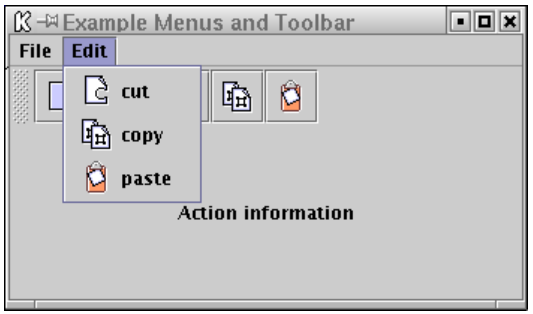
```
package examples.windows;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/** Clase ejemplo para demostrar el uso de menus
  * y barras de utilidades.
  */
public class MenusToolbar extends JFrame {
   private JLabel actionInfo
      = new JLabel( "Action information", JLabel.CENTER );
   /** Class constructor method
     * @param titleText Window's title bar text
     */
   public MenusToolbar( String titleText ) {
      super( titleText );
      addWindowListener( new WindowCloser() );

      JToolBar tb = new JToolBar();
      JMenu file = new JMenu( "File" );
      JMenu edit = new JMenu( "Edit" );
      JMenuBar mb = new JMenuBar();
      mb.add( file );
      mb.add( edit );
      NewAction na = new NewAction();
      file.add( na ).setMnemonic( 'N' );
      tb.add( na );
      SaveAction sa = new SaveAction();
      KeyStroke ks
         = KeyStroke.getKeyStroke( KeyEvent.VK_S,
                                   Event.CTRL_MASK );
      file.add( sa ).setAccelerator( ks );
      tb.add( sa );
      tb.addSeparator();
      CutAction cta = new CutAction();
```



```
      edit.add( cta );
      tb.add( cta );
      CopyAction cpa = new CopyAction();
      edit.add( cpa );
      tb.add( cpa );
      PasteAction pa = new PasteAction();
      edit.add( pa );
      tb.add( pa );
      setJMenuBar( mb );
      Container cp = getContentPane();
      cp.add( tb, BorderLayout.NORTH );
      cp.add( actionInfo, BorderLayout.CENTER );
      setSize( 350, 200 );
      setVisible( true );
   }
   class NewAction extends AbstractAction {
      public NewAction() {
         super( "new", new ImageIcon( "new.gif" ) );
      }
      public void actionPerformed( ActionEvent e ) {
         actionInfo.setText( "new selected" );
      }
   }
   class SaveAction extends AbstractAction {
      public SaveAction() {
         super( "save", new ImageIcon( "save.gif" ) );
      }
      public void actionPerformed( ActionEvent e ) {
         actionInfo.setText( "save selected" );
      }
   }
```

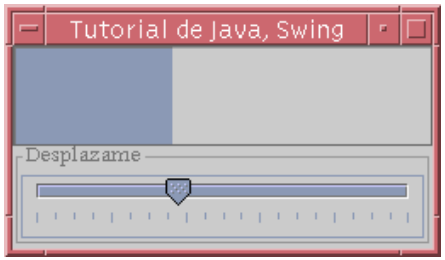
```
    class CutAction extends AbstractAction {
       public CutAction() {
          super( "cut", new ImageIcon( "cut.gif" ) );
       }
       public void actionPerformed( ActionEvent e ) {
          actionInfo.setText( "cut selected" );
       }
    }
    class CopyAction extends AbstractAction {
       public CopyAction() {
          super( "copy", new ImageIcon( "copy.gif" ) );
       }
       public void actionPerformed( ActionEvent e ) {
          actionInfo.setText( "copy selected" );
       }
    }
    class PasteAction extends AbstractAction {
       public PasteAction() {
          super( "paste", new ImageIcon( "paste.gif" ) );
       }
       public void actionPerformed( ActionEvent e ) {
          actionInfo.setText( "paste selected" );
       }
    }
    /** El metodo main
      * @param args no usado
      */
    public static void main( String[] args ) {
       new MenusToolbar( "Example Menus and Toolbar" );
    }
}
```

12.8. Sliders, barras de progreso y Scrollbars

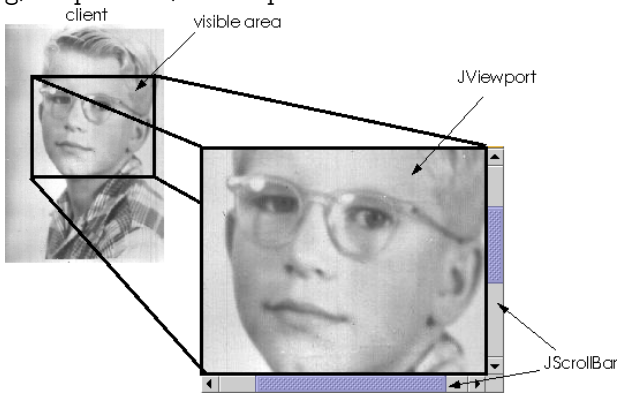
- Clase `javax.swing.JProgressBar`: Usado normalmente para mostrar el progreso de una operación larga.
  - `JProgressBar()`
  - `JProgressBar(BoundedRangeModel brm)`
  - `JProgressBar(int orientation)`
  - `JProgressBar(int min, int max)`
  - `JProgressBar(int min, int max, int value)`
  - `JProgressBar(int orientation, int min, int max, int value)`
- Clase `javax.swing.JSlider`: Permite seleccionar un valor entre un determinado rango.
  - `JSlider()`
  - `JSlider(BoundedRangeModel brm)`
  - `JSlider(int orientation)`
  - `JSlider(int min, int max)`
  - `JSlider(int min, int max, int value)`
  - `JSlider(int orientation, int min, int max, int value)`

(ver ejemplo `Swing/java14-13.java` y `Swing/java1428.java`)



- Clase `javax.swing.JScrollBar`: Usado para controlar la parte visible de un componente.
  Se usa como parte de un `JScrollPane`
  - `JScrollBar()`
  - `JScrollBar(int orientation)`
  - `JScrollBar(int orientation, int value, int extent, int min, int max)`
- Clase `javax.swing.JScrollPane`: Usado para mostrar un componente con una barra de scroll. Se usa por ejemplo para contener `JTextArea` y `JList`.
  `vsbPolicy` y `hsbPolicy` permiten pone la barra vertical u horizontal siempre, cuando se necesita o nunca.
  - `JScrollPane()`
  - `JScrollPane(Component view)`
  - `JScrollPane(int vsbPolicy,inthsbPolicy)`
    `JScrollPane(Component view, int vsbPolicy,int hsbPolicy)`

(ver ejemplos en `http://java.sun.com/docs/books/tutorial/uiswing/components/scrollpane.html`



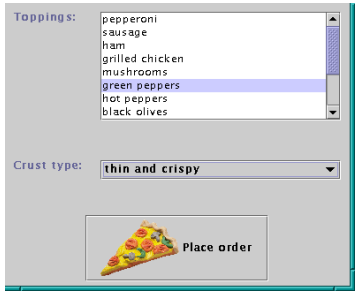
12.9. Listas y cajas combinadas (combo boxes)

- Clase `javax.swing.JComboBox`: Presenta una lista en la que se puede seleccionar un item.
  No tiene capacidad de scroll.
  - `JComboBox()`
  - `JComboBox(ComboBoxModel model)`
  - `JComboBox(Object[] items)`
  - `JComboBox(Vector items)`
- Clase `javax.swing.JList`: Presenta una lista en la que se puede seleccionar uno o varios items.
  Tiene capacidad de scroll si se coloca en un `JScrollPane`.
  - `JList()`
  - `JList(ListModel model)`
  - `JList(Object[] items)`
  - `JList(Vector items)`

- Ejemplo:

```
private JComboBox crustBox;
private JList toppingList;
crustBox = new JComboBox( new Object[] {
         "thick and chewy",
         "thin and crispy",
         "Chicago deep dish"
      }
   );
toppingList = new JList( new Object[] {
         "pepperoni",
         "sausage",
         "ham",
         "grilled chicken",
         "mushrooms",
         "green peppers",
         "hot peppers",
         "black olives",
         "tomato slices",
         "sun-dried tomatoes",
         "extra cheese",
         "pineapple",
         "anchovies"
      }
   );
toppingList.setSelectionMode(
      ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
JScrollPane scrollToppingList
      = new JScrollPane( toppingList );
```

(ver ejemplo Swing/P6/ListsAndComboBoxes.java)



12.10. Componentes para entrada de texto

- Clase `java.swing.JTextArea`: Región que puede contener varias líneas de texto.
  Puede colocarse en un `JScrollPane`.
  - `JTextArea()`
  - `JTextArea(String dcoModel)`
  - `JTextArea(int rows,int columns)`
  - `JTextArea(String text,int rows, int columns)`
  - `JTextArea(Document docModel, String text,int rows, int column)`

  (ver `Swing/P10/TextExamples.java`)
- Clase `java.swing.JTextField`: Para solicitar una línea de texto.
  - `JTextField()`
  - `JTextArea(String text)`
  - `JTextArea(int columns)`
  - `JTextArea(String text,int columns)`
  - `JTextArea(Document docModel, String text,int column)`
- Clase `javax.swing.JEditorPane`: permite mostrar texto plano, html y rtf.
- Clase `javax.swing.JPasswordField`

- Clase `javax.swing.JTextPane`: permite presentar varías líneas de texto con diferentes fuentes, tamaños, colores ... (ver `Swing/java1409.java`)

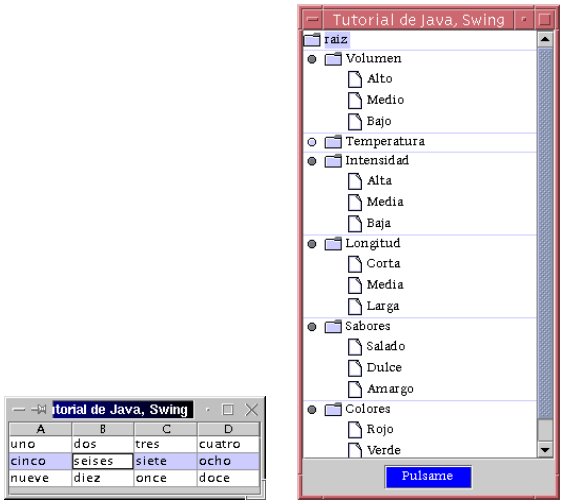
12.11. Diálogos predefinidos (choosers)

- Clase `javax.swing.JColorChooser`: permite seleccionar un color de una paleta.
- Clase `javax.swing.JFileChooser`: permite seleccionar un fichero.
- Clase `javax.swing.JOptionPane`

(ver Swing/P7/Choosers.java)

12.12. Tablas y árboles

- Clase `javax.swing.JTable`: (ver `Swing/P8/TableExample.java` y `Swing/java1416.java`)
- Clase `javax.swing.JTree`: (ver `Swing/P9/TreeExample.java` y `Swing/java1414.java`)



12.13. Dibujando con Swing

(ver `http://java.sun.com/products/jfc/tsc/articles/painting/index.html` y `http://java.sun.com/docs/book/tutorial/uiswing/14painting/index.html`)

(Ver los ejemplos **Swing/CoordinatesDemo.java**, **Swing/SelectionDemo.java Swing/IconDisplayer.java** y **serpienteSwing/**

- Swing sigue el modelo básico de pintado de AWT, extendiéndolo para mejorar su rendimiento.
  - El método `paint()` en componentes Swing es llamado como resultado de llamadas del *sistema* y de *aplicación*.
  - `update()` nunca se llama para componentes Swing.
  - Soporta también las llamadas a `repaint()` para repintar los componentes.
  - Swing utiliza doble buffer para dibujar los componentes
  - Proporciona el API de **RepaintManager** para los programas que quieran desarrollar su propio mecanismo de pintado (ver `http://java.sun.com/products/jfc/tsc/articles/painting/index.html` para más detalles).

12.13.1. Soporte de doble buffer

- Por defecto los compenentes son dibujados automáticamente usando *doble buffer*.
- El doble buffer se controla con la propiedad *doubleBuffered* del componente. Existen dos métodos para acceder y modificarla:

```
public boolean isDoubleBuffered()
public void setDoubleBuffered( boolean o )
```

- Este mecanismo utiliza un solo buffer oculto de pantalla por cada jerarquía de contenedores (el resultado de fijar doble buffer para un

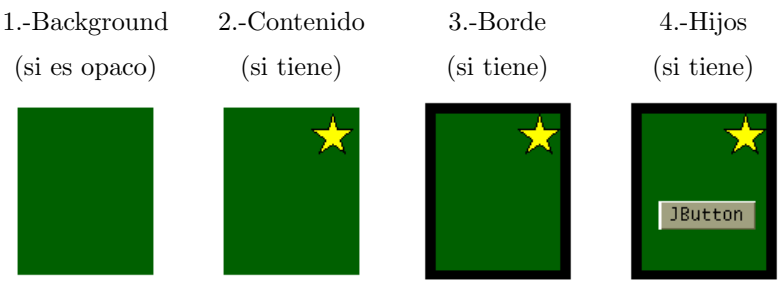
determinado contenedor hace que todos los componentes ligeros que contenga ese contenedor sean pintados en el buffer oculto).

- Por defecto es *true* para todos los componentes, pero el valor que realmente importa es el que tenga `JRootPane`.

12.13.2. Los métodos de dibujo en Swing

Las reglas que se aplican a los componentes ligeros del AWT, también se aplican a los componentes Swing; por ejemplo, `paint()` se llama cuando es necesario el repintado, excepto que en el caso de Swing, la llamada a `paint()` se convierte en la llamada a tres métodos separados, que siempre se invocan en el siguiente orden:

- `protected void paintComponent( Graphics g )`: Por defecto, dibuja el background del componente, si éste es opaco, y luego dibuja su contenido.
- `protected void paintBorder( Graphics g )`: Dibuja el borde del componente.
- `protected void paintChildren( Graphics g )`: Indica que se dibujen los componentes que contenga este componente.



- Componentes Swing sobreescribirán `paintComponent()` y no `paint()`.
- Generalmente no se suelen sobreescribir los otros dos métodos.
- Los programas llamarán a `repaint()` cuando necesiten actualizar el componente (al igual que en AWT).

(Ver como ejemplo **Swing/java1505.java**).

12.13.3. Propiedades adicionales de los componentes

Swing introduce un conjunto de propiedades nuevas en JComponent para aumentar la eficiencia de los algoritmos internos de repintado.

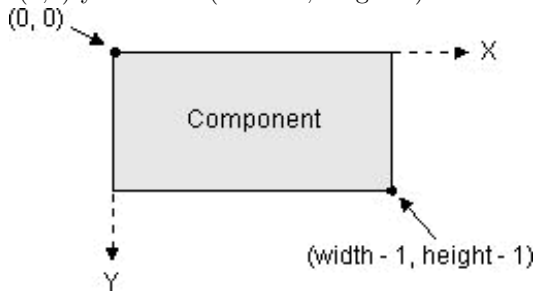
- *Transparencia*: Si un componente ligero es repintado, es posible que ese componente no pinte todos los bits asociados a él si es parcial o totalmente transparente.
  - Esto significa que siempre que sea repintado, cualquier cosa que haya debajo de él ha de repintarse en primer lugar.
  - Esto requiere que el sistema navegue a través de la jerarquía de componentes para encontrar el primer *antecesor pesado* desde el cual comenzar las operaciones de repintado (de atrás hacia delante).
- *Componentes solapados*: Si un componente ligero es pintado, es posible que haya algún otro componente ligero que lo solape parcialmente.
  - Siempre que el componente ligero original sea repintado, cualquier componente que solape a ese componente debe ser repintado.
  - Esto requiere que el sistema navegue a través de la jerarquía de componentes, comprobando los componentes solapados en cada operación de repintado.

Opacidad

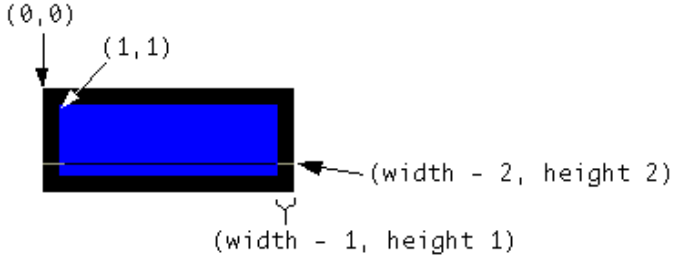
- Para mejorar el rendimiento en el caso de componentes opacos, Swing añade la propiedad de *opacidad* (propiedad `opaque`) a javax.swing.JComponent que puede valer:
  - *true*: El componente repintará todos los bits contenidos dentro de sus límites rectangulares.
  - *false*: El componente no garantiza que se vayan a repintar todos los bits que se encuentren dentro de sus límites rectangulares.
- La opacidad permite al sistema de pintado de Swing detectar cuándo una petición de repintado sobre un componente necesitará repintados adicionales de sus antecesores.
- El valor por defecto suele ser *true* para la mayoría de los componentes.

12.13.4. Algunos aspectos sobre el dibujo en Swing

- Cada componente utiliza su propio sistema de coordenadas que empieza en (0,0) y va hasta (width-1, height-1)



- Al dibujar, debemos tener en cuenta, además del tamaño del componente, el tamaño del borde:



- Puede obtener la anchura y altura del borde con el método `getInsets()`
- Para hacer que el repintado sea más rápido pueden usarse versiones de `repaint()` que recorten el área de dibujo (ver **Swing/SelectionDemo.java**)
- El componente debería implementar alguno de los métodos `getMinimumSize()`, `getPreferredSize()` y `getMaximumSize()`.

13. Entrada/salida de datos en Java

- Java realiza entradas y salidas a través de **streams** (flujos de datos).
- Un stream es una abstracción que produce o consume información.
- Un flujo está relacionado con un dispositivo físico a través del sistema de E/S de Java.
- Todos los flujos se comportan de la misma manera, incluso aunque estén relacionados con distintos dispositivos físicos: se usan las mismas clases I/O y métodos.
- Un flujo puede abstraer distintos tipos de entrada: archivo de disco, teclado, conexión a red.
- Un flujo puede abstraer distintos tipos de salida: consola, archivo de disco o conexión a red.
- Con un stream la información se traslada **en serie** (carácter a carácter o byte a byte) a través de esta conexión.

13.1. Clases de Java para E/S de datos

- El paquete `java.io` contiene las clases necesarias para E/S en Java.
- Dentro de este paquete existen dos familias de jerarquías distintas para la E/S de datos.
  - Clases que operan con bytes. (introducidas con Java 1.0)
  - Clases que operan con caracteres (un carácter en Java ocupa dos bytes porque sigue el código **Unicode**): deben utilizarse cuando queremos **internacionalizar** nuestro programa. (introducidas con Java 1.1)
- En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.
- Además `java.io` contiene otras clases que están fuera de estas jerarquías: `File`, `FileDescriptor`, `RandomAccessFile` y `StreamTokenizer`.

13.2. Clase File

Describe las propiedades de un objeto archivo.
Se utiliza para obtener o modificar la información asociada con un archivo de disco (permisos, hora, fecha y directorio) y para navegar por la jerarquía de directorios.

- Los objetos de esta clase son principalmente fuente o destino de datos en muchos programas.
- Un directorio se trata igual que un archivo con una propiedad adicional: una lista de nombres de archivo que se pueden examinar con el método `String[] list()`
- La clase `File` proporciona un aislamiento de la plataforma (por ej. si el separador de subdirectorios es `\` o `/`. El convenio en Java es usar `/`.
- Constructores:

  ```
  File(String filename)
  File(File folder, String filename)
  File(String folder, String filename)
  ```

  Ejemplos

  ```
  File f1 = new File("/");
  File f2 = new File("/","autoexec.bat");
  File f3 = new File(f1,"autoexec.bat");
  ```

- Métodos más comunes:
  - `boolean canRead()`:
  - `boolean canWrite()`:
  - `File createTempFile(String prefix,String suffix)`
    `File createTempFile(String prefix,String suffix,File folder)`:
    Crea un fichero temporal.
  - `boolean delete()`:
  - `void deleteOnExit()`: Marca el fichero para borrarlo cuando termine de ejecutarse la JVM. (útil para limpiar los ficheros temporales)

- `boolean exists()`: Devuelve `true` si el fichero existe.
- `String getAbsolutePath()`: Devuelve el path absoluto.
- `String getName()`: Devuelve el nombre del fichero. (parte del path que sigue a la última barra)
- `String getParent()`: Nombre de directorio donde está el fichero. (null si es el root)
- `File getParentFile()`: Similar a `getParent()` pero devuelve un objeto `File`.
- `boolean isAbsolute()`: Devuelve `true` si el fichero tiene un path absoluto y `false` si es un path relativo.
- `boolean isDirectory()`: Devuelve `true` si el objeto es un directorio.
- `boolean isFile()`: Devuelve `true` si el objeto es un fichero, y `false` si es un directorio.
- `long lastModified()`: Devuelve la fecha de última modificación.
- `String[] list()`
  `String[] list(FilenameFilter filter)`: Si el fichero es un directorio, devuelve un array con los nombres de los ficheros que contiene.
- `File[] listFiles()`
  `File[] listFiles(FileFilter filter)`
  `File[] listFiles(FilenameFilter filter)`: Similar a `list()`
- `File[] listRoots()`: Devuelve todos los directorios raiz del sistema.
- `boolean mkdir()`: Crea un directorio con el nombre de este `File`.
- `boolean setReadOnly()`
- `URL toURL()`: Devuelve el path del fichero como un objeto `URL` en la forma `file://pathname`.

Ejemplo de uso de clase File: JavaIO/P1/FileDemo.java

```
import java.io.File;
class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }
  public static void main(String args[]) {
    File f1 = new File("FileDemo.java");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

Ejemplo de uso de clase File con directorios:
JavaIO/P2/DirList.java

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "..";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```


13.3. Clases para flujos orientados a byte

- Estas dos jerarquías tienen una superclase abstracta para la entrada y otra para la salida.
- Las clases sombreadas en amarillo definen el origen o destino de los datos (dispositivo con que se conectan).
- Las demás clases añaden características particulares a la forma de enviar los datos.

13.3.1. Clases InputStream y OutputStream

- La mayoría de la E/S se hace con métodos definidos en `InputStream` y `OutputStream`.
- Las distintas subclases sobrecargan o sobreescriben algunos de estos métodos para circuntancias específicas.

Clase java.io.InputStream

- Superclase abstracta de todos los flujos orientados a entrada de bytes. Los métodos de esta clase lanzan una `IOException` si se producen condiciones de error.
- Constructor
  `InputStream()`
- Métodos
  - `int available()`: Devuelve los bytes disponibles en el flujo.
  - `void close()`: Cierra el flujo y libera los recursos que usa.
  - `void mark(int readlimit)`: Establece una marca en la posición actual del flujo, que permite regresar posteriormente a esa posición.
  - `boolean markSupported()`: Indica si el flujo permite poner marcas.
  - `int read()`
    `int read(byte[] buffer)`
    `int read(byte[] buffer, int offset, int length)`: Lee bytes del flujo de entrada.
  - `void reset()`: Reposiciona el flujo en la marca establecida previamente.
  - `long skip(long bytecount)`: Salta `bytecount` bytes devolviendo el número de bytes saltados.

Clase java.io.OutputStream

- Superclase abstracta de todos los flujos orientados a salida de bytes. Los métodos de esta clase devuelven void y lanzan una `IOException` si se producen condiciones de error.
- Constructor:
  `OutputStream()`
- Métodos
  - `void close()`: Cierra el flujo y libera los recursos que utiliza.
  - `void flush()`: Fuerza a escribir los posibles bytes que haya almacenados en un buffer de memoria.
  - `void write(int b)`
    `void write(byte[] bytebuffer)`
    `void write(byte bytebuffer[], int offset, int count)`:
    Escribe un byte o un array de bytes en el flujo.

13.3.2. Entrada/salida con ficheros: Clases FileInputStream y FileOutputStream

- Las clases `FileInputStream` y `FileOutputStream` tienen dos constructores. Uno recibe como parámetro un `String` y el otro un `File`:
  - Un `String` con el nombre del fichero.
  - Un objeto `File`. El uso de este constructor tiene como ventaja:
    Al crear el objeto `File` se pueden hacer ciertas comprobaciones previas (si un fichero de entrada existe, si es de sólo lectura, o de lectura-escritura)
- `FileOutputStream` tiene además el constructor:

  `FileOutputStream(String filePath, boolean append)`

  Si `append` es `true`, el fichero es abierto en modo *adición*.
- Los constructores de `FileInputStream` pueden lanzar la excepción `FileNotFoundException` cuando el fichero no existe.
- Los constructores de `FileOutputStream` lanzan un excepción `IOException` si no pueden crear el fichero (fichero era de sólo lectura).

Ejemplo de FileInputStream: JavaIO/P22/ShowFile.java

```
import java.io.*;

class ShowFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("File Not Found");
      return;
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Usage: ShowFile File");
      return;
    }
    do {
      i = fin.read();
      if(i != -1) System.out.print((char) i);
    } while(i != -1);

    fin.close();
  }
}
```

Otro ejemplo de uso de clase FileInputStream: JavaIO/P3/FileInputStreamDemo.java

Ejemplo de uso de clase FileOutputStream:
JavaIO/P4/FileOutputStreamDemo.java

```
import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) throws Exception {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();

    OutputStream f0 = new FileOutputStream("file1.txt");
    for (int i=0; i < buf.length; i += 2) {
      f0.write(buf[i]);
    }
    f0.close();

    OutputStream f1 = new FileOutputStream("file2.txt");
    f1.write(buf);
    f1.close();

    OutputStream f2 = new FileOutputStream("file3.txt");
    f2.write(buf,buf.length-buf.length/4,buf.length/4);
    f2.close();
  }
}
```

Otro ejemplo de uso de clase FileOutputStream:
JavaIO/P23/CopyFile.java

13.3.3. Lectura/escritura de matriz de bytes

Clase ByteArrayInputStream

- Flujo de entrada que usa una matriz de bytes como origen de los datos.
- Constructores:

```
ByteArrayInputStream(byte matriz[])
ByteArrayInputStream(byte matriz[],int position, int numBytes)
```

Ejemplo de uso de ByteArrayInputStream:
JavaIO/P5/ByteArrayInputStreamDemo.java

```
import java.io.*;
class ByteArrayInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
  }
}
```

Otro ejemplo de uso de ByteArrayInputStream: JavaIO/P6/ByteArrayInputStreamReset.java

```
import java.io.*;
class ByteArrayInputStreamReset {
  public static void main(String args[]) throws IOException {
    String tmp ="abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);
    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

Salida del programa

```
abc
ABC
```


Clase ByteArrayOutputStream

- Flujo de salida que usa una matriz de bytes como salida de los datos.
- Constructores:
  - `ByteArrayOutputStream()` Crea un búfer con 32 bytes.
  - `ByteArrayOutputStream(int numBytes)`: Crea un búfer con el tamaño especificado.

Ejemplo de uso de ByteArrayOutputStream: JavaIO/P7/ByteArrayOutputStreamDemo.java

```
import java.io.*;
class ByteArrayOutputStreamDemo {
  public static void main(String args[]) throws IOException {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();
    f.write(buf);
    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) {
      System.out.print((char) b[i]); }
    System.out.println("\nTo an OutputStream()");
    OutputStream f2 = new FileOutputStream("test.txt");
    f.writeTo(f2);
    f2.close();
    System.out.println("Doing a reset");
    f.reset();
    for (int i=0; i<3; i++) f.write('X');
    System.out.println(f.toString());
  }
}
```

Salida del programa

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

13.3.4. Flujos filtrados: FilterInputStream y FilterOutputStream

Existen varias clases que proporcionan una **funcionalidad adicional** añadiendo o sobreescribiendo métodos para preprocesar la salida antes de escribir los datos o postprocesar la entrada después de leer los datos.

- `BufferedInputStream` y `BufferedOutputStream`: Proporcionan lectura y escritura con búfer de memoria. (incrementan la eficiencia)
- `PushbackInputStream`: Permite devolver al flujo de entrada el último byte leido.
- `PrintStream`: Implementa métodos para mostrar los datos como texto. (ejemplo `print()` `println()`)
- `DataInputStream` y `DataOutputStream`: Transmiten datos de tipos primitivos en lugar de tratar el flujo como secuencias de bytes independientes.
- Todas estas clases extienden a `FilterInputStream` o `FilterOutputStream`.
- Para usarlas se crea un objeto `InputStream` o `OutputStream` que se pasa a su vez, al constructor correspondiente de `FilterInputStream` o `FilterOutputStream`.

13.3.5. Flujos con un búfer de bytes

Clase BufferedInputStream

- Constructores
  - `BufferedInputStream(InputStream ent)`: Crea el búfer con 512 bytes.
  - `BufferedInputStream(InputStream ent, int size)`: Crea el búfer con el tamaño especificado.

Ejemplo de uso de BufferedInputStream: JavaIO/P8/BufferedInputStreamDemo.java

```
import java.io.*;
class BufferedInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] =  s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marked = false;
    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;
        }
        else {
          marked = false; }
        break;
      case ';':
        if (marked) {
          marked = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
```

```
      case ' ':
        if (marked) {
          marked = false;
          f.reset();
          System.out.print("&");
        } else
          System.out.print((char) c);
        break;
      default:
        if (!marked)
          System.out.print((char) c);
        break;
      }
    }
  }
}
```

Salida del programa

```
This is a (c) copyright symbol but this is &copy not.
```


Clase BufferedOutputStream

Es similar a cualquier **OutputStream** con la excepción que se añade un método `flush()` que se usa para asegurar que los datos del búfer se escriben físicamente en el dispositivo de salida. El objetivo de esta clase es solamente incrementar la velocidad, reduciendo el número de veces que el sistema escribe datos.

- Constructores
  - `BufferedOutputStream(OutputStream sal)`: Crea el búfer con 512 bytes.
  - `BufferedOutputStream(OutputStream sal, int size)`: Crea el búfer con el tamaño especificado.

Clase PushbackInputStream

Permite devolver un byte al flujo de entrada con `void unread(int ch)`.

- Constructor:
  `PushbackInputStream(InputStream ent)`

Ejemplo de uso de PushbackInputStream: JavaIO/P9/PushbackInputStreamDemo.java

```
import java.io.*;
class PushbackInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    PushbackInputStream f = new PushbackInputStream(in);
    int c;
    while ((c = f.read()) != -1) {
      switch(c) {
      case '=':
        if ((c = f.read()) == '=')
          System.out.print(".eq.");
        else {
          System.out.print("<-");
          f.unread(c);
        }
        break;
     default: System.out.print((char) c);
       break;
      }
    }
  }
}
```

Salida del programa

```
if (a .eq. 4) a <- 0;
```


13.3.6. Clase PrintStream

Esta clase proporciona las capacidades para formatear la salida con `Sistem.out`.

- Constructores:

  `PrintStream(OutputStream outputStream)`
  `PrintStream(OutputStream outputStream, boolean flushOnNewLine)`

- Esta clase contiene los métodos `print()` y `println()` que soportan todos los tipos, incluido `Object`.

- Si un argumento no es un tipo simple, entonces los métodos de `PrintStream` llaman al método `toString()` del objeto, y luego imprimen el resultado.

- Los constructores de esta clase están obsoletos (*deprecated*) desde la versión Java 1.1 ya que `PrintStream` no maneja caracteres Unicode, lo que hace que los programas no puedan *internacionalizarse*.

- Los métodos de `PrintStream` no están *deprecated*, lo que nos permite usar métodos de esta clase, pero no crear objetos de ella. La razón de lo anterior es que **System.out** es de la clase `PrintStream` y `System.out` se usa mucho en el código existente.

  Sin embargo, para nuevos programas, es mejor restringir el uso de `System.out` para depuración o programas de ejemplo.
  Cualquier programa real que muestre salida por la consola debería usar la clase `PrintWriter` en la forma que veremos más adelante.

13.3.7. Clases DataInputStream y DataOutputStream

Estas clases derivan de `FilterInputStream` y `FilterOutputStream` respectivamente.

DataInputStream

Permite leer distintos tipos de datos primitivos, además de objectos `String`

- Todos los métodos empienzan con `read` tales como `readByte()`, `readFloat()`, etc.
- Esta clase, junto con la compañera **DataOutputStream** permite mover datos *primitivos* de un sitio a otro vía un flujo. Estos lugares vendrán determinados por las clases sombreadas en amarillo de la jerarquía de clases vista anteriormente.

DataOutputStream

Es el complemento de `DataInputStream`, que da formato a los tipos primitivos y objetos `String` convirtiéndolos en un flujo de forma que cualquier `DataInputStream`, de cualquier máquina, los pueda leer.

- Todos los métodos empiezan por `write`, como `writeByte()`, `writeFloat()`, etc.

Ejemplo de DataInputStream y DataOutputStream: JavaIO/P21/DataIODemo.java

```
import java.io.*;
public class DataIODemo {
    public static void main(String[] args) throws IOException {
        DataOutputStream out = new DataOutputStream(new
                               FileOutputStream("invoice1.txt"));
        double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
        int[] units = { 12, 8, 13, 29, 50 };
        String[] descs = { "Java T-shirt","Java Mug",
          "Duke Juggling Dolls","Java Pin", "Java Key Chain" };
        for (int i = 0; i < prices.length; i ++) {
            out.writeDouble(prices[i]);
            out.writeChar('\t');
            out.writeInt(units[i]);
            out.writeChar('\t');
            out.writeChars(descs[i]);
            out.writeChar('\n');
        }
        out.close();
        DataInputStream in = new DataInputStream(new
                                 FileInputStream("invoice1.txt"));
        double price;
        int unit;
        StringBuffer desc;
        double total = 0.0;
```

```
        try {
            while (true) {
              price = in.readDouble();
              in.readChar();       // throws out the tab
              unit = in.readInt();
              in.readChar();       // throws out the tab
              char chr;
              desc = new StringBuffer(20);
              char lineSep = System.getProperty(
                          "line.separator").charAt(0);
              while ((chr = in.readChar()) != lineSep)
                desc.append(chr);
              System.out.println("You've ordered " +  unit +
                   " units of " + desc + " at $" + price);
              total = total + unit * price;
            }
        } catch (EOFException e) {
        }
        System.out.println("For a TOTAL of: $" + total);
        in.close();
    }
}
```


13.3.8. Clase SequenceInputStream

- Permite concatenar múltiples flujos del tipo `InputStream`.
- Constructores:
  - `SequenceInputStream(InputStream primero, InputStream segundo)`
  - `SequenceInputStream(Enumeration enum)`

Ejemplo de uso de SequenceInputStream: JavaIO/P10/SequenceInputStreamDemo.java

```
import java.io.*;
import java.util.*;
class InputStreamEnumerator implements Enumeration {
    private Enumeration files;
    public InputStreamEnumerator(Vector files) {
        this.files = files.elements();
    }
    public boolean hasMoreElements() {
        return files.hasMoreElements();
    }
    public Object nextElement() {
        try {
            return new FileInputStream(
                       files.nextElement().toString());
        } catch (Exception e) {
            return null;
        }
    }
}
```

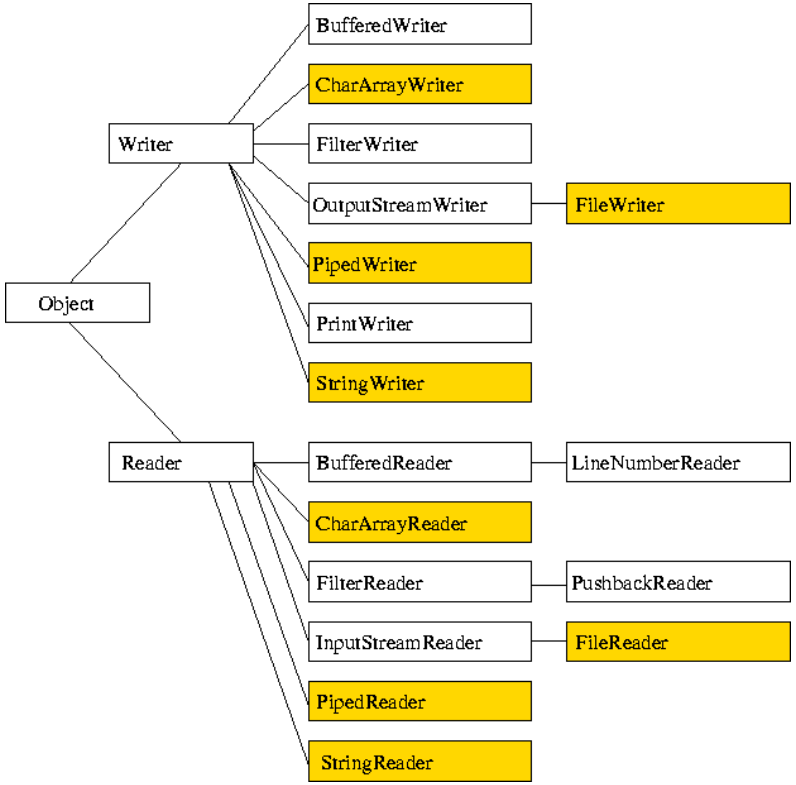
```
class SequenceInputStreamDemo {
    public static void main(String args[]) throws Exception {
        int c;
        Vector files = new Vector();
        files.addElement("SequenceInputStreamDemo.java");
        files.addElement("../P9/PushbackInputStreamDemo.java");
        InputStreamEnumerator e = new InputStreamEnumerator(files);
        InputStream input = new SequenceInputStream(e);
        while ((c = input.read()) != -1) {
            System.out.print((char) c);
        }
        input.close();
    }
}
```


13.4. Clase RandomAccessFile

- Encapsula un archivo de acceso aleatorio.
- No es derivada de `InputStream` ni `OutputStream`, sino que implementa las interfaces `DataInput` y `DataOuput`, (al igual que `DataInputStream` y `DataOutputStream`) que definen los métodos de E/S básicos.
- Constructores:
  - `RandomAccessFile(File objArch, String acceso)`
  - `RandomAccessFile(String nomArch, String acceso)` acceso puede ser `"r"` o `"rw"`.
- `void seek(long position)` permite establecer la posición.
- Contiene los métodos `close`, `read` y `write` como en `InputStream` y `OutputStream`.
- Además contiene los mismos métodos (`readBoolean()`, `readChar()`, `readByte()`, `readDouble()` ..., `writeBoolean()`, `writeChar()`, `writeByte()`, `writeDouble()` ...) para leer y escribir que `DataInputStream` y `DataOutputStream`

13.5. Clases para flujos orientados a carácter

- Diferencia con los flujos orientados a bytes son:
  - Convierten cada carácter de la codificación del sistema operativo nativo a código Unicode.
- La mayoría de las clases de flujos orientados a byte tienen su correspondiente clase de flujo orientado a carácter.
  `FileReader` es el equivalente a `FileInputStream`
  `FileWriter` es el equivalente a `FileOutputStream`



13.5.1. Clases Reader y Writer

Clase Reader

Es una clase abstracta que define el modelo de Java de entrada de caracteres.

Los métodos de esta clase lanzan una `IOException` cuando ocurren errores.

- Métodos
  - `abstract void close()`: Cierra el flujo y libera los recursos que usa.
  - `void mark(int readlimit)`: Establece una marca en la posición actual del flujo, que permite regresar posteriormente a esa posición.
  - `boolean markSupported()`: Indica si el flujo permite poner marcas.
  - `int read()`
    `int read(char[] buffer)`
    `abstract int read(char[] buffer, int offset, int length)`: Lee caracteres del flujo de entrada.
  - `void reset()`: Reposiciona el flujo en la marca establecida previamente.
  - `long skip(long charcount)`: Salta `charcount` caracteres devolviendo el número de caracteres realmente saltados.

Clase Writer

Es una clase abstracta que define el modelo de salida de caracteres.

Todos sus métodos devuelven `void` y lanzan una `IOException` en caso de error.

- Métodos
  - `abstract void close()`: Cierra el flujo y libera los recursos que utiliza.
  - `abstract void flush()`: Fuerza a escribir los posibles caracteres que haya almacenados en un buffer de memoria.

- `void write(int ch)`
  `void write(char[] charbuffer)`
  `void write(char charbuffer[], int offset, int count)`:
  Escribe un carácter o un array de caracteres en el flujo.
- `void write(String str)`
  `void write(String str, int offset, int count)`: Escribe un `String` completo o parte de él en el flujo.

13.5.2. Clases FileReader y FileWriter

Son las clases equivalentes a `FileInputStream` y `FileOutputStream`

- Constructores:
  `FileReader(File file)`
  `FileReader(FileDescriptor fd)`
  `FileReader(String fileName)`
  `FileWriter(File file)` `FileWriter(FileDescriptor fd)`
  `FileWriter(String fileName)`
  `FileWriter(String fileName, boolean append)`

Ejemplo de uso de FileReader: JavaIO/P11/FileReaderDemo.java

Ejemplo de uso de FileWriter: JavaIO/P12/FileWriterDemo.java

13.5.3. Clases InputStreamReader y OutputStreamWriter

- Permiten transformar programas construidos para versiones antiguas de Java (usando flujos orientados a bytes) a las nuevas versiones (con flujos orientados a carácter).
- Sirven de puente desde los flujos orientados a bytes hasta los orientados a carácter.
- Leen bytes y los transforman en caracteres Unicode.
- Constructor:
  `InputStreamReader(InputStream in)`
  `OutputStreamWriter(OutputStream out)`
- Ejemplo:

```
BufferedReader in
   = new BufferedReader(new InputStreamReader(System.in));
```

13.5.4. Clases CharArrayReader y CharArrayWriter

Son las clases equivalentes a `ByteArrayInputStream` y `ByteArrayOuputStream`.

- Permiten leer o escriber un array de caracteres como un objeto Reader o Writer.
- Constructores:
  `CharArrayReader(char[] buf)`
  `CharArrayReader(char[] buf, int offset, int length)`
  `CharArrayWriter()`
  `CharArrayWriter(int initialsize)`

Ejemplo de uso de CharArrayReader: JavaIO/P13/CharArrayReaderDemo.java

Ejemplo de uso de CharArrayWriter: JavaIO/P14/CharArrayWriterDemo.java

13.5.5. Clases BufferedReader y BuffererdWriter

Son las clases correspondientes a `BufferedInputStream` y `BufferedOutputStream`

- Constructores:

```
BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

Ejemplo de uso de BufferedReader:
JavaIO/P15/BufferedReaderDemo.java

13.5.6. Clase PushbackReader

Es la clase correspondiente a `PushbackInputStream`

- Constructores:

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

Ejemplo de uso de PushbackReader:
JavaIO/P16/PushbackReaderDemo.java

13.5.7. Clases StringReader y StringWriter

`StringReader` es la clase correspondiente a `StringBufferedInputStream` mientras que `StringWriter` no tiene una análoga en los flujos de bytes.

- Permiten leer (escribir) un **String** (**StringBuffer**) como un objecto Reader (Writer).
- Constructores:

```
StringReader(String s)
StringWriter()
StringWriter(int initialSize)
```

13.5.8. Clases PipedReader y PipedWriter

Son las clases correspondientes a `PipedInputStream` y `PipedOutputStream`.

- Permiten construir tuberías (especialmente útiles para comunicar hebras).

13.5.9. Clase PrintWriter

Es la clase orientada a caracteres equivalente a `PrintStream`

- Tiene los mismos métodos.
- Constructores:

```
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

13.6. Flujos predefinidos

- Dentro del paquete `java.lang` se define la clase `System` que encapsula algunos aspectos del entorno de ejecución (por ej. se puede obtener hora actual, valores asociados de algunos propiedades asociadas con el sistema).
- Además, contiene tres variables con flujos predefinidos: `in`, `out` y `err` declarados como `public static final`.
- `System.in`: Entrada estándar.
  - Es de la clase `java.io.BufferedInputStream`
- `System.out`: Salida estándar.
  - Es de la clase `java.io.PrintStream`
- `System.err`: Salida estándar de error.
  - Es de la clase `java.io.PrintStream`

13.6.1. Entrada por consola

- En Java 1.0, la única forma de realizar entrada por la consola era mediante un flujo de bytes.
- En Java 2, la forma preferida para realizar entrada por la consola es mediante un flujo de caracteres.
  Para ello haremos uso de la clase `BufferedReader` usando por ejemplo el constructor `BufferedReader(Reader inputReader)` en la siguiente forma:

```
BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
```

Ejemplo de lectura de caracteres: JavaIO/P17/BRRead.java

```
import java.io.*;
class BRRead {
  public static void main(String args[])
    throws IOException
  {
    char c;
    BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");
    // read characters
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Ejemplo de lectura de Strings: JavaIO/P18/BRReadLines.java

```
import java.io.*;
class BRReadLines {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str;
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```


13.6.2. Salida por consola

- La salida por consola se realiza con `print()` y `println()` de la clase `PrintStream`.
- También es posible el uso de `write()` de `OutputStream`.
- Aunque `System.out` es un flujo de bytes, su uso es aceptable para salida simple de programas. Pero existe una alternativa con flujos de caracteres con el uso de la clase `PrintWriter`.

Ejemplo de System.out: JavaIO/P19/WriteDemo.java

```
class WriteDemo {
  public static void main(String args[]) {
    int b;
    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

- Para escribir en la consola con un flujo de caracteres usaremos la clase `PrintWriter`. De esta forma el programa puede ser internacionalizado:

  `PrintWriter(OutputStream outputStream, boolean flushOnNewLine)`

Ejemplo de System.out con PrintWriter: JavaIO/P20/PrintWriterDemo.java

```
import java.io.*;
public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

13.7. Más ejemplos de utilización de flujos de E/S

wc (contador de palabras): JavaIO/P24/WordCount.java

```
import java.io.*;
class WordCount {
  public static int words = 0;
  public static int lines = 0;
  public static int chars = 0;
  public static void wc(InputStreamReader isr)
    throws IOException {
    int c = 0;
    boolean lastWhite = true;
    String whiteSpace = " \t\n\r";
    while ((c = isr.read()) != -1) {
      chars++;
      if (c == '\n') {
        lines++;
      }
      int index = whiteSpace.indexOf(c);
      if(index == -1) {
        if(lastWhite == true) {
          ++words;
        }
        lastWhite = false;
      }
      else {
        lastWhite = true;
      }
    }
    if(chars != 0) {
      ++lines;
    }
  }
```



```
  public static void main(String args[]) {
    FileReader fr;
    try {
      if (args.length == 0) { // We're working with stdin
        wc(new InputStreamReader(System.in));
      }
      else { // We're working with a list of files
        for (int i = 0; i < args.length; i++) {
          fr = new FileReader(args[i]);
          wc(fr);
        }
      }
    }
    catch (IOException e) {
      return;
    }
    System.out.println(lines + " " + words + " " + chars);
  }
}
```

13.8. Clase StreamTokenizer

- Permite buscar patrones en un flujo de entrada.
- Divide el flujo de entrada en tokens que están delimitados por conjuntos de caracteres.
- Constructor:

  `StreamTokenizer(Reader inStream)`
- Métodos:
  - `void resetSyntax()`: El conjunto de delimitadores por defecto están definidos para obtener los tokens de programas Java. Este método resetea los delimitadores por defecto.
  - `void eolIsSignificant(boolean eolFlag)`: Para que los caracteres de fin de línea se consideren como tokens.
  - `void wordChars(int start, int end)`: Para especificar el rango de caracteres que pueden usarse en palabras.
  - `void whitespaceChars(int start, int end)`: Los caracteres considerados como espacios en blanco (separadores).
  - `int nextToken()`: Obtiene el siguiente token del flujo de entrada. Devuelve el tipo de token (`TT_EOF`, `TT_EOL`, `TT_NUMBER`, `TT_WORD`).
- Existen tres variables de instancia en esta clase:
  - `double nval`: Guardará el valor del número que se acaba de leer con `nextToken()`.
  - `String sval`: Guardará la palabra que se acaba de leer.
  - `int ttype`: Indica el tipo de token que acaba de leerse (`TT_EOF`, `TT_EOL`, `TT_NUMBER`, `TT_WORD` o un carácter si el token era simplemente un carácter)

wc con StreamTokenizer: JavaIO/P25/WordCount.java

```
import java.io.*;
class WordCount {
  public static int words=0;
  public static int lines=0;
  public static int chars=0;
  public static void wc(Reader r) throws IOException {
    StreamTokenizer tok = new StreamTokenizer(r);
    tok.resetSyntax();
    tok.wordChars(33, 255);
    tok.whitespaceChars(0, ' ');
    tok.eolIsSignificant(true);
    while (tok.nextToken() != tok.TT_EOF) {
      switch (tok.ttype) {
        case tok.TT_EOL:
          lines++;
          chars++;
          break;
        case tok.TT_WORD:
          words++;
        default: // FALLSTHROUGH
          chars += tok.sval.length();
          break;
      }
    }
  }
```

```
  public static void main(String args[]) {
    if (args.length == 0) { // We're working with stdin
      try {
        wc(new InputStreamReader(System.in));
        System.out.println(lines + " " + words + " " + chars);
      } catch (IOException e) {};
    } else { // We're working with a list of files
      int twords = 0, tchars = 0, tlines = 0;
      for (int i=0; i<args.length; i++) {
        try {
          words = chars = lines = 0;
          wc(new FileReader(args[i]));
          twords += words;
          tchars += chars;
          tlines += lines;
          System.out.println(args[i] + ": " +
            lines + " " + words + " " + chars);
        } catch (IOException e) {
          System.out.println(args[i] + ": error.");
        }
      }
      System.out.println("total: " +
      tlines + " " + twords + " " + tchars);
    }
  }
}
```


13.9. Serialización de objetos

- La representación binaria de los tipos primitivos puede escribirse o leerse de un fichero con las clases `DataOutputStream` y `DataInputStream`.
- La serialización es un proceso por el que un objeto cualquiera se puede convertir en una secuencia de bytes para guardarlo en un fichero y posteriormente poder leerlo.
- Esto permite:
  - Transmitir objetos a través de la red.
  - Grabar objetos en ficheros que pueden leerse en ejecuciones posteriores del programa.
- Para serializar una clase, ésta debe implementar el interfaz `Serializable` que no define ningún método. Casi todas las clases de Java son serializables.
- Para escribir y leer objetos se usan las clases `ObjectInputStream` y `ObjectOutputStream` (que son subclases de `InputStream` y `OutputStream`) que contienen los métodos `writeObject` y `readObject`.

```
ObjectOutputStream objout = new ObjectOutputStream(
                                new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(
                                new FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

- Las clases `ObjectInputStream` y `ObjectOutputStream` implementan los interfaces `ObjectInput` y `ObjectOutput` respectivamente, los cuales a su vez extienden a los interfaces `DataInput` y `DataOutput`.
- Al serializar un objeto, se serializan todas sus variables y objetos miembro. A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables).
- Al reconstruirse se hace de igual manera.

- Variables **static** deben serializarse explícitamente en caso necesario.
- La palabra clave **transient** permite especificar que un objeto o variable no sea serializada con el resto del objeto.
- La clase `ObjectStreamField` nos da una forma alternativa para controlar en tiempo de ejecución qué campos se serializan.
  - Constructor: `ObjectStreamField(String n, Class clazz)`
  - Para ello añadimos el campo `private static final ObjectStreamField[] serialPersistentFields` a la clase a serializar, designando los campos que queremos serializar.
- Las clases serializables pueden definir los siguientes métodos para controlar la serialización:

```
private void writeObject(ObjectOutputStream stream)
             throws IOException
private void readObject(ObjectInputStream stream)
             throws ClassNotFoundException, IOException
```

- Obtendremos el comportamiento por defecto llamando a `stream.defaultWriteObject()` y `stream.defaultReadObject()` dentro de los anteriores métodos.
- En estos métodos podremos hacer uso de los métodos `readObject()`, `readChar()`, `readInt()`, `readDouble()`, ..., `writeObject()`, `writeChar()`, `writeInt()`, `writeDouble()`, ..., de `ObjectInputStream` y `ObjectOutputStream`

- Ejemplo de uso:

```
static double g = 9.8
private void writeObject(ObjectOutputStream stream)
             throws IOException {
  stream.defaultWriteObject();
  stream.writeDouble(g);
}
private void readObject(ObjectInputStream stream)
             throws ClassNotFoundException, IOException {
  stream.defaultReadObject();
  stream.readDouble(g);
}
```

Ejemplo de uso de clase Serializable: JavaIO/P26/ObjectSaver.java

```
import java.io.*;
class ObjectToSave implements Serializable {
   static final long serialVersionUID = 7482918152381158178L;
   private int i;
   private String s;
   private transient double d;
   public ObjectToSave( int i, String s, double d ) {
      this.i = i;
      this.s = s;
      this.d = d;
   }
   public String toString() {
      return "i = " + i + ", s = " + s + ", d = " + d;
   }
   private void readObject( ObjectInputStream ois )
         throws ClassNotFoundException, IOException {
      System.out.println( "deserializing..." );
      ois.defaultReadObject();
      System.out.println( "deserialized" );
   }
```

```
   private void writeObject( ObjectOutputStream oos )
         throws IOException {
      System.out.println( "serializing..." );
      oos.defaultWriteObject();
      System.out.println( "serialized" );
   }
}
public class ObjectSaver {
   private static final String FILE_NAME = "objects.ser";
   public static void main( String[] args ) {
      try {
         ObjectToSave ots = new ObjectToSave( 57, "pizza", 3.14 );
         File objectFile = new File( FILE_NAME );
         if ( objectFile.exists() ) {
            objectFile.delete();
         }
         FileOutputStream fos = new FileOutputStream( objectFile );
         ObjectOutputStream oos = new ObjectOutputStream( fos );
         oos.writeObject( ots );
         oos.close();
         FileInputStream fis = new FileInputStream( objectFile );
         ObjectInputStream ois = new ObjectInputStream( fis );
         ObjectToSave retrieved = (ObjectToSave) ois.readObject();
         ois.close();
         System.out.println( retrieved );
      }
      catch ( OptionalDataException x ) {
         System.out.println( x );
         x.printStackTrace();
      }
      catch ( ClassNotFoundException x ) {
         System.out.println( x );
         x.printStackTrace();
      }
```



```
      catch ( IOException x ) {
         System.out.println( x );
         x.printStackTrace();
      }
   }
}
```

13.9.1. Especificación del número de versión

- `serialVersionUID` se usa en Java para identificar la versión de una clase. (cambia cada vez que se hace cualquier modificación en la clase)
- La versión es grabada con los objetos serializados y comprobada al recuperarlos:
  Si no coincide se lanza `ClassNotFoundException`
- El siguiente comando permite ver la versión de una clase:
  `serialver clase`

13.9.2. Interfaz Externalizable

- La interfaz **Externalizable** extiende **Serializable**.
- A diferencia de **Serializable**, no serializa nada automáticamente.
- La clase que la implemente definirá los métodos:

```
public void writeExternal(ObjectOutput out)
                  throws IOException
public void readExternal(ObjectInput in)
                  throws IOException, ClassNotFoundException
```

- Antes de llamar a `readExternal` para leer el objeto serializado, Java llamará al constructor por defecto de la clase serializada.

Ejemplo de uso de clase Externalizable: JavaIO/P27/Blip3.java

```
import java.io.*;
import java.util.*;
class Blip3 implements Externalizable {
  int i;
  String s;
  public Blip3() {
    System.out.println("Blip3 Constructor");
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x; i = a;
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out) throws IOException {
    System.out.println("Blip3.writeExternal");
    out.writeObject(s); out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
              throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    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();
      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();
    }
  }
}
```

14. Nuevas características de Java 2, v5.0

14.1. Autoboxing and AutoUnboxing

14.1.1. Revisión de clases de envoltura

- Java usa tipos primitivos (**int**, **double**, etc) para guardar los tipos de datos básicos, por razones de eficiencia: los tipos primitivos no son parte de la jerarquía de clases.
- Java proporciona *clases de envoltura* (*wrappers*) que permiten encapsular un tipo primitivo en un objeto.

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

- Una vez encapsulado un tipo primitivo en una clase de envoltura, el valor puede obtenerse llamando a cualquier método definido en la clase de envoltura. Por ejemplo para las clases de envoltura numéricas:

```
byte byteValue()   double doubleValue()   float floatValue()
int intValue()     long longValue()       short shortValue()
```

- El proceso de encapsular un valor primitivo en un objeto se conoce como *boxing*. En versiones anteriores a Java 2, v5.0 todas las operaciones de encapsulamiento (*boxing*) las hacía manualmente el programador:

```
Integer iOb = new Integer(100);
```

- El proceso de extraer el valor de un objeto de una clase de envoltura se conoce como *unboxing*. De nuevo, antes de Java 2, v5.0, todos los *unboxing* las tenía que hacer el programador manualmente:

```
int i = iOb.intValue();
```

- Las operaciones manuales *boxing* y *unboxing* son tediosas y propensas a errores.

14.1.2. Fundamentos de Autoboxing/Unboxing

- *Autoboxing* es el proceso por el que un tipo primitivo, se encapsula automáticamente en la clase de envoltura equivalente cuando se necesita un objeto de esta clase: no es necesario construir explícitamente un objeto.

```
Integer iOb = 100;
```

- *Autounboxing* es el proceso por el que el valor de un objeto de una clase de envoltura es automáticamente extraido del objeto cuando se necesite: no es necesario llamar explícitamente a métodos tales como `intValue()` o `doubleValue()`.

```
int i = iOb;
```

Ejemplo de autoboxing/unboxing: P70/AutoBox.java

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {
    Integer iOb = 100; // autobox an int
    int i = iOb; // auto-unbox
    System.out.println(i + " " + iOb);  // displays 100 100
  }
}
```

14.1.3. Autoboxing en llamadas a métodos

- Autoboxing/unboxing se aplica también cuando un argumento se pasa a un método, o cuando un método devuelve un valor.

Ejemplo de autoboxing/unboxing en métodos: P71/AutoBox2.java

```
// Autoboxing/unboxing takes place with
// method parameters and return values.
class AutoBox2 {
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v ; // auto-unbox to int
  }
  public static void main(String args[]) {
    // Pass an int to m() and assign the return value
    // to an Integer.  Here, the argument 100 is autoboxed
    // into an Integer.  The return value is also autoboxed
    // into an Integer.
    Integer iOb = m(100);
    System.out.println(iOb);
  }
}
```


14.1.4. Autoboxing/Unboxing en expresiones

- En general Autoboxing/Unboxing se aplica siempre que sea necesaria una conversión entre un objeto y un tipo primitivo: esto se aplica en expresiones.

  Ejemplo de autoboxing/unboxing en expresiones: P72/AutoBox3.java

```
class AutoBox3 {
  public static void main(String args[]) {
    Integer iOb, iOb2;
    int i;
    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);
  }
}
```

- Podemos incluso mezclar diferentes tipos:

  Otro ejemplo de autoboxing/unboxing en expresiones: P73/AutoBox4.java

```
class AutoBox4 {
  public static void main(String args[]) {
    Integer iOb = 100;;
    Double dOb = 98.6;
    dOb = dOb + iOb;
    System.out.println("dOb after expression: " + dOb);
  }
}
```

14.1.5. Autoboxing/Unboxing de valores Boolean y Character

Ejemplo de autoboxing/unboxing con char y boolean: P74/AutoBox5.java

```
class AutoBox5 {
  public static void main(String args[]) {
```

```
    // Autobox/unbox a boolean.
    Boolean b = true;
    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");
    // Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char
    System.out.println("ch2 is " + ch2);
  }
}
```

14.1.6. Autoboxing/Unboxing ayuda a prevenir errores

Ejemplo de error en unboxing manual: P75/UnboxingError.java

```
class UnboxingError {
  public static void main(String args[]) {
    Integer iOb = 1000; // autobox the value 1000
    int i = iOb.byteValue(); // manually unbox as byte !!!
    System.out.println(i);  // does not display 1000 !
  }
}
```

14.1.7. Advertencia sobre el uso de Autoboxing/Unboxing

- Debemos restringir el uso de las clases de envoltura para los casos en que se necesite un objeto que represente el tipo primitivo, ya que puede hacer los programas más lentos:

  Ejemplo de mal uso de autoboxing/unboxing

  ```
  Double a, b, c;
  a = 10.0;
  b = 4.0;
  c = Math.sqrt(a*a + b*b);
  System.out.println("Hipotenusa es " + c);
  ```


14.2. Generics

14.2.1. ¿Qué son los generics?

- *Generic* significa *tipo parametrizado* (*parameterized type*): permite crear clases, interfaces y métodos en que el tipo de dato para el que trabajan se especifica con un parámetro.
- Una clase, interfaz o método que usa un tipo parametrizado se dice que es *generico*.
- En versiones previas de Java, se podían también crear clases, interfaces y métodos genéricos usando la clase **Object** pero podían presentarse problemas de seguridad de tipos.
- Los *generics* añaden seguridad de tipos ya que no hay que usar castings explícitos para pasar de **Object** al tipo concreto que se esté usando.

14.2.2. Un ejemplo simple

Ejemplo de generics: P76/GenDemo.java

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
class Gen<T> {
  T ob; // declare an object of type T
  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }

  // Show type of T.
  void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
  }
}

// Demonstrate the generic class.
class GenDemo {
  public static void main(String args[]) {
    // Create a Gen reference for Integers.
    Gen<Integer> iOb;
```

```
    // Create a Gen<Integer> object and assign its
    // reference to iOb.  Notice the use of autoboxing
    // to encapsulate the value 88 within an Integer object.
    iOb = new Gen<Integer>(88);

    // Show the type of data used by iOb.
    iOb.showType();

    // Get the value of in iOb. Notice that
    // no cast is needed.
    int v = iOb.getob();
    System.out.println("value: " + v);

    System.out.println();

    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");

    // Show the type of data used by strOb.
    strOb.showType();

    // Get the value of strOb. Again, notice
    // that no cast is needed.
    String str = strOb.getob();
    System.out.println("value: " + str);
  }
}
```

- **T** es el nombre del *parámetro tipo* (*type parameter*)
- En la función **main**, **Integer** y luego **String** se conocen como *argumentos tipo* (*type argument*)

- Los tipos genéricos con argumentos distintos son incompatibles en asignaciones:

  Ejemplo

  ```
  Gen<Integer> iOb;
  iOb = new Gen<Double>(88.0); // Error en compilación
  ```

  Otro ejemplo

  ```
  iOb = strOb; // Error en compilación
  ```

- Los generics funcionan sólo con objetos:

  ```
  // Error, no se pueden usar tipos primitivos
  Gen<int> strOb = new Gen<int>(53);
  ```

14.2.3. ¿Cómo mejoran los generics la seguridad de tipos?

- Podriamos construir una clase con la misma funcionalidad que la clase **Gen** anterior, pero sin usar genéricos, especificando **Object** como tipo de dato.

Una clase equivalente sin generics: P77/NonGenDemo.java

```
// NonGen is functionally equivalent to Gen
// but does not use generics.
class NonGen {
  Object ob; // ob is now of type Object

  // Pass the constructor a reference to
  // an object of type Object
  NonGen(Object o) {
    ob = o;
  }

  // Return type Object.
  Object getob() {
    return ob;
  }

  // Show type of ob.
  void showType() {
    System.out.println("Type of ob is " +
                       ob.getClass().getName());
  }
}

// Demonstrate the non-generic class.
class NonGenDemo {
  public static void main(String args[]) {
    NonGen iOb;

    // Create NonGen Object and store
    // an Integer in it. Autoboxing still occurs.
```

```
    iOb = new NonGen(88);

    // Show the type of data used by iOb.
    iOb.showType();

    // Get the value of iOb.
    // This time, a cast is necessary.
    int v = (Integer) iOb.getob();
    System.out.println("value: " + v);

    System.out.println();

    // Create another NonGen object and
    // store a String in it.
    NonGen strOb = new NonGen("Non-Generics Test");

    // Show the type of data used by strOb.
    strOb.showType();
    // Get the value of strOb.
    // Again, notice that a cast is necessary.
    String str = (String) strOb.getob();
    System.out.println("value: " + str);

    // This compiles, but is conceptually wrong!
    iOb = strOb;
    v = (Integer) iOb.getob(); // runtime error!
  }
}
```

- En el anterior ejemplo el compilador no puede conocer el tipo de dato que se almacena en **NonGen**, lo cual está mal por dos razones:
  - Se necesitan casting explícitos para recuperar los datos.
  - Muchos errores de *tipos incompatibles* no se pueden detectar hasta el tiempo de ejecución:

    ```
    iOb = strOb;
    v = (Integer) iOb.getob(); // runtime error!
    ```

- Usando generics, lo que antes eran errores en tiempo de ejecución se convierten en errores en tiempo de compilación.

14.2.4. Generics con dos parámetros tipo

Clase generics con dos parámetros: P78/SimpGen.java

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Pass the constructor a reference to
  // an object of type T.
  TwoGen(T o1, V o2) {
    ob1 = o1;
```



```
    ob2 = o2;
  }
  // Show types of T and V.
  void showTypes() {
    System.out.println("Type of T is " +
                       ob1.getClass().getName());

    System.out.println("Type of V is " +
                       ob2.getClass().getName());
  }
  T getob1() {
    return ob1;
  }
  V getob2() {
    return ob2;
  }
}

// Demonstrate TwoGen.
class SimpGen {
  public static void main(String args[]) {

    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Generics");

    // Show the types.
    tgObj.showTypes();

    // Obtain and show values.
    int v = tgObj.getob1();
    System.out.println("value: " + v);

    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

14.2.5. Tipos limitados

- A veces nos interesa crear una clase genérica que sólo sirva para tipos de una determinada clase y subclases

Ejemplo erroneo

```
// Stats attempts (unsuccessfully) to
// create a generic class that can compute
// the average of an array of numbers of
// any given type.
//
// The class contains an error!
class Stats<T> {
  T[] nums; // nums is an array of type T

  // Pass the constructor a reference to
  // an array of type T.
  Stats(T[] o) {
    nums = o;
  }

  // Return type double in all cases.
  double average() {
    double sum = 0.0;
```

```
    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue(); // Error de compilación

    return sum / nums.length;
  }
}
```

- Con *parámetros limitados* (*bounded types*) podemos crear un límite superior para la superclase de la que deben derivar obligatoriamente los argumentos tipo.

  `<T extends superclass>`

Clase con tipo limitado: P79/BoundsDemo.java

```
// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
  T[] nums; // array of Number or subclass

  // Pass the constructor a reference to
  // an array of type Number or subclass.
  Stats(T[] o) {
    nums = o;
  }

  // Return type double in all cases.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;
  }
}

// Demonstrate Stats.
class BoundsDemo {
  public static void main(String args[]) {

    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);

    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average();
    System.out.println("dob average is " + w);

    // This won't compile because String is not a
    // subclass of Number.
//    String strs[] = { "1", "2", "3", "4", "5" };
//    Stats<String> strob = new Stats<String>(strs);

//    double x = strob.average();
//    System.out.println("strob average is " + v);

  }
}
```

14.2.6. Uso de argumentos comodín

- A veces en una clase genérica necesitamos métodos con un parámetro genérico que puede ser compatible pero de distinto tipo al de la clase.
- Por ejemplo el método `saveAvg()` en:

```
Integer inums[] = {1, 2, 3, 4, 5};
Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

```
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if(iob.sameAvg(dob))
  System.out.println("Averages are the same.");
else
  System.out.println("Averages differ.");
```

- Una primera solución (**incorrecta**) para implementar `sameAvg()` sería:

```
// Determine if two averages are the same
boolean sameAvg(Stats<T> ob) {
  if(average() == ob.average())
    return true;
  return false;
}
```

- La solución **correcta** requiere del uso de un *argumento comodín* (*wildcard argument*) que representa un tipo desconocido (ver **P80/WildcardDemo.java**)

```
// Determine if two averages are the same
boolean saveAvg(Stats<?> ob) {
  if(average() == ob.average())
    return true;
  return false;
}
```


14.2.7. Argumentos comodín limitados

- Especifican un límite superior o bien un límite inferior para el argumento tipo, que permite restringir el tipo de objetos para los que puede usarse un determinado método.
  - `<? extends superclase>`: límite superior.
  - `<? super subclase>`: límite inferior.

Ejemplo de argumento comodín limitado: P81/BoundedWildcard.java

- Supongamos la siguiente jerarquía de clases:

```
// Two-dimensional coordinates.
class TwoD {
  int x, y;
  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}
// Three-dimensional coordinates.
class ThreeD extends TwoD {
  int z;
  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}
// Four-dimensional coordinates.
class FourD extends ThreeD {
  int t;
  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}
```

- Sea la siguiente clase genérica (array de **Coords**):

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}
```

Cada elemento de **Coords** puede ser de la clase **TwoD** o una clase derivada.

- Un método para mostrar las coordenadas X e Y de cada elemento del array (de la clase **TwoD** o una clase derivada) sería:

```
static void showXY(Coords<?> c) {
  System.out.println("X Y Coordinates:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y);
  System.out.println();
}
```

- Pero, si queremos un método (por ejemplo `showXYX()`) que funcione sólo para las clases **ThreeD** y **FourD** necesitaremos usar un *argumento comodín limitado*.

```
static void showXYZ(Coords<? extends ThreeD> c) {
  System.out.println("X Y Z Coordinates:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y + " " +
                       c.coords[i].z);
  System.out.println();
}
```

- Para especificar un límite inferior usaremos la sintaxis:

```
<? super subclase>
```

- Ahora sólo clases que sean superclases de *subclase* son argumentos permitidos.

14.2.8. Métodos genéricos

- Es posible crear **métodos genéricos** que usan uno o más parámetros tipo dentro de una *clase no genérica*.

```
<lista-parámetros-tipo> tipo-retorno nombre-método(
                                  lista-parámetros) { //...
```

Ejemplo de método genérico: P82/GenMethDemo.java

```
// Demonstrate a simple generic method.
class GenMethDemo {
  // Determine if an object is in an array.
  static <T, V extends T> boolean isIn(T x, V[] y) {

    for(int i=0; i < y.length; i++)
      if(x.equals(y[i])) return true;

    return false;
  }

  public static void main(String args[]) {

    // Use isIn() on Integers.
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if(isIn(2, nums))
      System.out.println("2 is in nums");

    if(!isIn(7, nums))
      System.out.println("7 is not in nums");

    System.out.println();

    // Use isIn() on Strings.
    String strs[] = { "one", "two", "three",
                      "four", "five" };

    if(isIn("two", strs))
      System.out.println("two is in strs");

    if(!isIn("seven", strs))
      System.out.println("seven is not in strs");

    // Opps! Won't compile! Types must be compatible.
//    if(isIn("two", nums))
//      System.out.println("two is in strs");
  }
}
```

Constructores genéricos

- Los constructores también pueden ser genéricos aunque su clase no lo sea.

Ejemplo de método genérico: P83/GenConsDemo.java

```
// Use a generic constructor.
class GenCons {
  private double val;
```

```
  <T extends Number> GenCons(T arg) {
    val = arg.doubleValue();
  }

  void showval() {
    System.out.println("val: " + val);
  }
}

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

    GenCons test = new GenCons(100);
    GenCons test2 = new GenCons(123.5F);

    test.showval();
    test2.showval();
  }
}
```

14.2.9. Interfaces genéricos

- Los interfaces también pueden ser genéricos al igual que las clases:

Ejemplo de método genérico: P84/GenIFDemo.java

```
// A generic interface example.

// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
  T min();
  T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
  T[] vals;

  MyClass(T[] o) { vals = o; }

  // Return the minimum value in vals.
  public T min() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0) v = vals[i];

    return v;
  }

  // Return the maximum value in vals.
  public T max() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) > 0) v = vals[i];

    return v;
  }
}

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



```
    Integer inums[] = {3, 6, 2, 8, 6 };
    Character chs[] = {'b', 'r', 'p', 'w' };

    MyClass<Integer> iob = new MyClass<Integer>(inums);
    MyClass<Character> cob = new MyClass<Character>(chs);

    System.out.println("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());

    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
  }
}
```

Puesto que **MinMax** requiere un tipo que extienda **Comparable**, la clase **MyClass** debe especificar el mismo límite.

Además no se debe especificar tal límite en la claúsula **implements**:

```
// Error
class MyClass<T extends Comparable<T>>
      implements MinMax<T extends Comparable <T>> {
```

- En general, si una clase implementa un interfaz genérico, entonces la clase debe ser genérica también:

  ```
  class MyClass implements MinMax<T> { // Error
  ```

  Puesto que la clase no declara un parámetro tipo, no hay forma de pasar uno a **MinMax**.

- Pero, si que se podría si la clase implementa un *tipo específico* de interfaz genérico:

  ```
  class MyClass implements MinMax<Integer> { // OK
  ```

14.2.10. Tipos rasos y código heredado

- Para que código antiguo (anterior a la versión 5.0) pueda funcionar con código genérico, Java permite que una clase genérica pueda usarse sin argumentos tipo: esto creará un *tipo raso* (*raw type*).
  - Por ejemplo, para crear un objeto de la clase **Gen** sustituyendo el tipo **T** por **Object**:

    ```
    Gen raw = new Gen(new Double(98.6));
    ```

  - El inconveniente es que se pierde la *seguridad de tipos* de los genéricos.

Ejemplo de raw tipe: P85/RawDemo.java

```
// Demonstrate a raw type.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Demonstrate raw type.
class RawDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");
```



```
    // Create a raw-type Gen object and give it
    // a Double value.
    Gen raw = new Gen(new Double(98.6));

    // Cast here is necessary because type is unknown.
    double d = (Double) raw.getob();
    System.out.println("value: " + d);

    // The use of a raw type can lead to runtime.
    // exceptions.  Here are some examples.

    // The following cast causes a runtime error!
//    int i = (Integer) raw.getob(); // runtime error

    // This assigment overrides type safety.
    strOb = raw; // OK, but potentially wrong
//    String str = strOb.getob(); // runtime error

    // This assignment also overrides type safety.
    raw = iOb; // OK, but potentially wrong
//    d = (Double) raw.getob(); // runtime error
  }
}
```

- El compilador de java **javac** muestra *unchecked warnings* cuando un raw type se usa de forma que puede hacer peligrar la seguridad de tipos.
- Deberiamos limitar el uso de raw types a los casos en que mezclamos código antiguo con nuevo.

14.2.11. Jerarquías de clases genéricas

- Las clases genéricas pueden formar parte de una jerarquía de clases al igual que una clase normal.
- La única particularidad es que cualquier argumento tipo necesitado en una superclase debe ser pasado a las subclases.

Superclases genéricas

Ejemplo de jerarquía de clases

```
// A simple generic class heirarchy.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}
```

Otro ejemplo de jerarquía de clases: P86/HierDemo.java

```
// A subclass can add its own type parameters.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }
  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
  V ob2;

  Gen2(T o, V o2) {
    super(o);
    ob2 = o2;
  }

  V getob2() {
```



```
    return ob2;
  }
}

// Create an object of type Gen2.
class HierDemo {
  public static void main(String args[]) {

    // Create a Gen2 object for String and Integer.
    Gen2<String, Integer> x =
      new Gen2<String, Integer>("Value is: ", 99);

    System.out.print(x.getob());
    System.out.println(x.getob2());
  }
}
```

Subclase genérica

- Una clase no genérica puede ser perfectamente superclase de una clase genérica.

Ejemplo de subclase genérica: P87/HierDemo2.java

```
// A nongeneric class can be the superclass
// of a generic subclass.

// A nongeneric class.
class NonGen {
  int num;

  NonGen(int i) {
    num = i;
  }
  int getnum() {
    return num;
  }
}

// A generic subclass.
class Gen<T> extends NonGen {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o, int i) {
    super(i);
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Create a Gen object.
class HierDemo2 {
  public static void main(String args[]) {

    // Create a Gen object for String.
    Gen<String> w = new Gen<String>("Hello", 47);

    System.out.print(w.getob() + " ");
```

```
    System.out.println(w.getnum());
  }
}
```


Comparación de tipos en tiempo de ejecución con una jerarquía genérica

- El operador **instanceof** puede aplicarse a objetos de clases genéricas.

Ejemplo de instanceof: P88/HierDemo3.java

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}

 // Demonstrate runtime type ID implications of generic class hierarchy.
class HierDemo3 {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    // See if iOb2 is some form of Gen2.
    if(iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 is instance of Gen2");

    // See if iOb2 is some form of Gen.
    if(iOb2 instanceof Gen<?>)
      System.out.println("iOb2 is instance of Gen");

    System.out.println();

    // See if strOb2 is a Gen2.
    if(strOb2 instanceof Gen2<?>)
      System.out.println("strOb is instance of Gen2");

    // See if strOb2 is a Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb is instance of Gen");

    System.out.println();

    // See if iOb is an instance of Gen2, which its not.
    if(iOb instanceof Gen2<?>)
```

```
      System.out.println("iOb is instance of Gen2");

    // See if iOb is an instance of Gen, which it is.
    if(iOb instanceof Gen<?>)
      System.out.println("iOb is instance of Gen");

    // The following can't be compiled because
    // generic type info does not exist at runtime.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
  }
}
```

Sobreescritura de métodos en una clase genérica

- Un método de una clase genérica puede ser sobreescrito al igual que en clases normales.

Ejemplo de sobreescritura de un método genérico: P89/OverrideDemo.java

```
// Overriding a generic method in a generic class.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }
  // Return ob.
  T getob() {
    System.out.print("Gen's getob(): " );
    return ob;
  }
}

// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
  // Override getob().
  T getob() {
    System.out.print("Gen2's getob(): ");
    return ob;
  }
}

// Demonstrate generic method override.
class OverrideDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    System.out.println(iOb.getob());
```



```
    System.out.println(iOb2.getob());
    System.out.println(strOb2.getob());
  }
}
```

14.2.12. Genéricos y colecciones

- Uno de los usos más importantes de los genéricos se encuentra en las clases de colección de Java (*Collection Framework*).
- En Java 2, v5.0 todas sus clases (por ejemplo **ArrayList**, **LinkedList**, **TreeSet**) e interfaces (por ejemplo **Iterator**) han sido reajustadas para usar genéricos.
- El parámetro tipo genérico de estas clases e interfaces especifica el tipo de objeto que la colección contiene y que el iterador obtiene.
- El uso de genéricos mejora la seguridad en los tipos en las clases de colección.

Ejemplo de uso de una colección en código pre-genérico

```
// Pre-generics example that uses a collection.
import java.util.*;

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

    // These lines store strings, but any type of object
    // can be stored.  In old-style code, there is no
    // convenient way restrict the type of objects stored
    // in a collection
    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    Iterator itr = list.iterator();
    while(itr.hasNext()) {

      // To retrieve an element, an explicit type cast is needed
      // because the collection stores only Object.
      String str = (String) itr.next(); // explicit cast needed here.

      System.out.println(str + " is " + str.length() + " chars long.");
    }
  }
}
```

- En código pre-genérico las colecciones almacenan referencias a **Object**, para poder guardar objetos de cualquier clase.
- Esto es una fuente de errores:

- El programador es el que debe asegurar que sólo se guardan objetos del tipo apropiado en la colección.
  Por ejemplo el compilador no daría error en:
  `list.add(new Integer(100));`
- Al recuperar un objeto de la colección hay que emplear una conversión de tipo explícita. Esto suele provocar errores en tiempo de ejecución.
  Por ejemplo la siguiente sentencia generaría un error en tiempo de ejecución:
  `Integer i = (Integer) itr.next();`

- Los genéricos solucionan los errores anteriores en el uso de colecciones.
  - Aseguran que sólo se puedan almacenar objetos de la clase adecuada.
  - Eliminan la necesidad de hacer casting para obtener objetos de la colección.
- Todo esto es posible porque las colecciones añaden un parámetro tipo que especifica el tipo de la colección. Por ejemplo:
  `class ArrayList<E>`

Ejemplo de uso de colección en versión genérica: P90/NewStyle.java

```
// Modern, generics version.
import java.util.*;

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

    // Now, list holds references of type String.
    ArrayList<String> list = new ArrayList<String>();

    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    // Notice that Iterator is also generic.
    Iterator<String> itr = list.iterator();

    // The following statement will now cause a compile-time eror.
//    Iterator<Integer> itr = list.iterator(); // Error!

    while(itr.hasNext()) {
      String str = itr.next(); // no cast needed

      // Now, the following line is a compile-time,
```



```
      // rather than runtime, error.
//    Integer i = itr.next(); // this won't compile

      System.out.println(str + " is " + str.length() + " chars long.");
    }
  }
}
```

14.2.13. Errores de ambigüedad

- La inclusión de genéricos puede provocar un nuevo tipo de error: *ambigüedad*.
- Estos errores ocurren cuando el proceso de *erasure* hace que dos declaraciones genéricas distintas se conviertan en el mismo tipo.

Ejemplo en sobrecarga de métodos

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // ...

  // These two overloaded methods are ambiguous.
  // and will not compile.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

14.2.14. Restricciones en el uso de genéricos

Los parámetros tipo no pueden instanciarse

Ejemplo

```
// Can't create an instance of T.
class Gen<T> {
  T ob;
  Gen() {
    ob = new T(); // Illegal!!!
  }
}
```

Restricciones en miembros estáticos

- Ningún miembro **static** puede usar un parámetro tipo declarado en la clase.

Ejemplo

```
class Wrong<T> {
  // Wrong, no static variables of type T.
  static T ob;

  // Wrong, no static method can use T.
  static T getob() {
    return ob;
  }

  // Wrong, no static method can access object
  // of type T.
  static void showob() {
    System.out.println(ob);
 }
}
```

- Si que es válido declarar métodos genéricos **static**, que definan sus propios parámetros tipo (ver ejemplo de sección **Métodos genéricos**).

Restricciones en arrays genéricos

- No se puede instanciar un array cuyo tipo base sea un parámetro tipo.
- No se puede declarar un array de referencias a genéricos de un tipo específico.

Ejemplo de generics y arrays: P91/GenArrays.java

```
// Generics and arrays.
class Gen<T extends Number> {
  T ob;

  T vals[]; // OK

  Gen(T o, T[] nums) {
    ob = o;

    // This statement is illegal.
    // vals = new T[10]; // can't create an array of T

    // But, this statement is OK.
    vals = nums; // OK to assign reference to existent array
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Can't create an array of type-specific generic references.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

    // This is OK.
    Gen<?> gens[] = new Gen<?>[10]; // OK
  }
}
```

- Si que se puede crear un array de referencias de un tipo genérico usando el caracter comodín:

  `Gen<?> gens[] = new Gen<?>[10]; // OK`

Restricciones en excepciones

- Una clase genérica no puede extender **Throwable**: por tanto no se pueden crear clases de excepción genéricas.

14.3. El bucle for-each

- Un bucle for-each permite recorrer una colección de objetos, tal como un array, de forma secuencial, de principio a fin.
- La forma de un bucle for-each en Java es:

  `for(tipo var-iter : objetoIterable) sentencias;`

  - `objetoIterable` debe implementar el interfaz **Iterable**.
- El bucle se repite hasta que se obtienen todos los elementos de la colección.

Ejemplo de bucle tradicional

```
int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

Versión usando bucle for-each

```
int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;
for(int x: nums) sum += x;
```

- El nuevo bucle automatiza algunos aspectos del bucle tradicional:
  - Elimina la necesidad de establecer un contador del bucle.
  - De especificar un valor de comienzo y final.
  - De indexar manualmente el array.

Ejemplo de bucle for-each: P92/ForEach.java

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // use for-each style for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

- Es posible finalizar el bucle antes de llegar al final usando **break**.

Ejemplo de uso de break en bucle for-each: P93/ForEach2.java

```
// Use break with a for-each style for.
class ForEach2 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Use for to display and sum the values.
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
      if(x == 5) break; // stop the loop when 5 is obtained
    }
    System.out.println("Summation of first 5 elements: " + sum);
  }
}
```

Iterando sobre arrays multidimensionales

- En Java los arrays multidimensionales son realmente *arrays de arrays*.
- En un array multidimensional, el bucle for-each obtiene un array de una dimensión menor en cada iteración.

Ejemplo de bucle for-each en array multidimensional: P94/ForEach3.java

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```


14.3.1. Bucle for-each en colecciones

- El bucle for-each permite iterar sobre los objetos de cualquier clase que implemente **Iterable**, por ejemplo las clases de colección.

  Ejemplo de bucle for-each en una colección: P95/AvgCollection.java

```
// Using a for-each for loop with a collection.
import java.util.*;

class AvgCollection {
  static double getAvg(ArrayList<Double> nums) {
    double sum = 0.0;

    for(double itr : nums)
      sum = sum + itr;

    return sum / nums.size();
  }

  public static void main(String args[]) {
    ArrayList<Double> list = new ArrayList<Double>();

    list.add(10.14);
    list.add(20.22);
    list.add(30.78);
    list.add(40.46);

    double avg = getAvg(list);

    System.out.println("List average is " + avg);

  }
}
```

- En el ejemplo, el bucle for-each sustituye a la siguiente versión tradicional:

```
Iterator<Double> itr = nums.iterator();
while(itr.hasNext()) {
  Double d = itr.next();
  sum = sum + d;
}
```

14.3.2. Creación de objetos iterables

- Cualquier clase que implemente **Iterable** puede emplearse en un bucle for-each.
- **Iterable** es un interfaz añadido en Java 2, v5.0, y declarado como:

```
interface Iterable<T>
```

- **Iterable** contiene sólo el método:

  `Iterator<T> iterator()`

- El interfaz **Iterator** se define como:

  `interface Iterator<E>`

- **Iterator** define los métodos:
  - **boolean hasNext()**: Devuelve **true** si quedan más elementos.
  - **E next()**: Devuelve el siguiente elemento. Lanza **NoSuchElementException** si no hay más elementos.
  - **void remove()**: Borra el elemento actual. Este método es opcional. Lanza **IllegalStateException** si la llamada a este método no está precedida de una llamada a **next()**. Lanza **UnsoportedOperationException** si el método no está implementado en la clase.

- El bucle for-each llama implícitamente a los métodos **hasNext()** o **next()**.

Ejemplo de clase iterable: P96/ForEachIterable.java

```
// Using a for-each for loop on an Iterable object.
import java.util.*;

// This class supports iteration of the
// characters that comprise a string.
class StrIterable implements Iterable<Character>,
                             Iterator<Character> {
  private String str;
  private int count = 0;

  StrIterable(String s) {
    str = s;
  }

  // The next three methods impement Iterator.
  public boolean hasNext() {
    if(count < str.length()) return true;
    return false;
  }

  public Character next() {
    if(count == str.length())
      throw new NoSuchElementException();

    count++;
    return str.charAt(count-1);
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
```



```
  // This method implements Iterable.
  public Iterator<Character> iterator() {
    return this;
  }
}

class ForEachIterable {
  public static void main(String args[]) {
    StrIterable x = new StrIterable("This is a test.");

    // Show each character.
    for(char ch : x)
      System.out.print(ch);

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

14.4. Varargs: Argumentos de longitud variable

- Un argumento de longitud variable se especifica usando tres puntos consecutivos (...). Por ejemplo:

  ```
  static void vaTest(int ... v) {
  ```

- Esto indica que **vaTest()** puede llamarse con cero o más argumentos de tipo **int**.
- El argumento **v** es implícitamente un array **int**[]

Ejemplo de argumento de longitud variable: P97/VarArgs.java

```
// Demonstrate variable-length arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {

    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);      // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();        // no args
  }
}
```


- Un método puede tener parámetros *normales* junto con un **único** parámetro de longitud variable que debe ser el **último**.

  ```
  int doIt(int a, int b, double c, int ... vals) {
  ```

Otro ejemplo de argumento de longitud variable: P98/VarArgs2.java

```
// Use varargs with standard arguments.
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

14.4.1. Sobrecarga de métodos vararg

- Los métodos vararg pueden también sobrecargarse.

Ejemplo de vararg y sobrecarga: P99/VarArgs3.java

```
// Varargs and overloading.
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest(1, 2, 3);
    vaTest("Testing: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

14.4.2. Varargs y ambigüedad

- Al sobrecargar métodos pueden ocurrir errores de ambigüedad que hacen que el programa no compile.

Ejemplo de vararg y ambigüedad: P100/VarArgs4.java

```
// Varargs, overloading, and ambiguity.
//
```



```
// This program contains an error and will
// not compile!
class VarArgs4 {
  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");
    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");
    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }


  public static void main(String args[])
  {
    vaTest(1, 2, 3);  // OK
    vaTest(true, false, false); // OK
    vaTest(); // Error: Ambiguous!
  }
}
```

- Otro ejemplo:

  ```
  static void vaTest(int ... v){ // ...
  static void vaTest(int n, int ... v){ // ...
  ```

  Aquí, el compilador no sabría a qué método llamar cuando se hace:

  ```
  vaTest(1);
  ```

14.5. Enumeraciones

14.5.1. Introducción

- En la forma más simple, es una lista de constantes con nombre.

Ejemplo

```
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
```

- Cada identificador **Jonathan**, **GoldenDel**, etc es una *constante de*

enumeración, que es implícitamente un miembro *public* y *static* de **Apple**.

- Las variables *enumeration* se declaran y usan en la misma forma en que se hace para variables de tipos primitivos:

  `Apple ap;`

Ejemplo de uso de enumeración: P101/EnumDemo.java

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
class EnumDemo {
  public static void main(String args[])
  {
    Apple ap;
    ap = Apple.RedDel;

    // Output an enum value.
    System.out.println("Value of ap: " + ap);
    System.out.println();

    ap = Apple.GoldenDel;

    // Compare two enum values.
    if(ap == Apple.GoldenDel)
      System.out.println("ap conatins GoldenDel.\n");

    // Use an enum to control a switch statement.
    switch(ap) {
      case Jonathan:
        System.out.println("Jonathan is red.");
        break;
      case GoldenDel:
        System.out.println("Golden Delicious is yellow.");
        break;
      case RedDel:
        System.out.println("Red Delicious is red.");
        break;
      case Winsap:
        System.out.println("Winsap is red.");
        break;
      case Cortland:
        System.out.println("Cortland is red.");
        break;
    }
  }
}
```

14.5.2. Métodos values() y valueOf()

- Las enumeraciones contienen dos métodos predefinidos:
  - `public static tipo-enum[] values()`: Devuelve un array con una lista de las constantes de enumeración.

  - `public static tipo-enum valueOf(String str)`:Devuelve la constante de enumeración cuyo valor corresponde al String pasado en *str*.

Ejemplo de uso de métodos: P102/EnumDemo2.java

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;
    System.out.println("Here are all Apple constants");
    // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);
    System.out.println();
    // use valueOf()
    ap = Apple.valueOf("Winsap");
    System.out.println("ap contains " + ap);
  }
}
```

14.5.3. Las enumeraciones son clases

- Las enumeraciones son clases realmente.
- Cada constante de enumeración es un objeto de la clase de enumeración.
- Las enumeraciones pueden tener constructores, variables de instancia y métodos:
  - Los constructores son llamados automáticamente cuando se declara cada constante de enumeración.
  - Cada constante de enumeración tiene su propia copia de las variables de instancia.

Ejemplo de uso de enumeración: P103/EnumDemo3.java

```
// Use an enum constructor, instance variable, and method.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winsap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }
```

```
  int getPrice() { return price; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
    Apple ap;

    // Display price of Winsap.
    System.out.println("Winsap costs " +
                       Apple.Winsap.getPrice() +
                       " cents.\n");

    // Display all apples and prices.
    System.out.println("All apple prices:");
    for(Apple a : Apple.values())
      System.out.println(a + " costs " + a.getPrice() +
                         " cents.");
  }
}
```


Otro ejemplo de enumeración: P111

```
public enum Coin {
    penny(1), nickel(5), dime(10), quarter(25);

    Coin(int value) { this.value = value; }

    private final int value;

    public int value() { return value; }
}

public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.values())
            System.out.println(c + ":   \t"
                  + c.value() +"cts \t" + color(c));
    }
    private enum CoinColor { copper, nickel, silver }
    private static CoinColor color(Coin c) {
        switch(c) {
          case penny:   return CoinColor.copper;
          case nickel:  return CoinColor.nickel;
          case dime:
          case quarter: return CoinColor.silver;
          default: throw new AssertionError("Unknown coin: " + c);
        }
    }
}
```

- Es posible incluir más de un constructor:

Otro ejemplo

```
// Use an enum constructor.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winsap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }

  // Overloaded constructor
  Apple() { price = -1; }

  int getPrice() { return price; }
}
```

- **Restricciones**:
  - Una enumeración no puede heredar de otra.
  - Una enumeración no puede ser una superclase.

14.5.4. Clase Enum

- Las enumeraciones heredan implícitamente de la clase **java.lang.Enum** (no es posible declarar un **enum** que herede de otra superclase).

  `abstract class Enum<E extends Enum<E>>`

- Los métodos más importantes son:
  - **final int ordinal()**: Devuelve un entero que indica la posición de la constante de enumeración en la lista de constantes.
  - **final int compararеTo(E e)**: Compara el valor ordinal de dos constantes de enumeración.
  - **final boolean equals(Object obj)**: Devuelve **true** si *obj* y el objeto llamante referencian la misma constante.
  - **String toString()**: Devuelve el nombre de la constante llamante. Este nombre puede ser distinto del usado en la declaración de la enumeración.

Ejemplo de uso de enumeración: P104/EnumDemo4.java

```
// Demonstrate ordinal(), compareTo(), and equals().

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all apple constants" +
                       " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());

    ap =  Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    System.out.println();
```



```
    // Demonstrate compareTo() and equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);

    System.out.println();

    if(ap.equals(ap2))
      System.out.println("Error!");

    if(ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);

    if(ap == ap3)
      System.out.println(ap + " == " + ap3);

  }
}
```

14.6. Static import

- Al usar una sentencia *import static* es posible usar los miembros **static** de una clase directamente sin tener que incluir delante el nombre de la clase.

Ejemplo de import static: P105/Hypot.java

```
// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Here, sqrt() and pow() can be called by themselves,
    // without their class name.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

14.6.1. Forma general de static import

- Hay dos formas para la sentencia **static import**:
  - `import static paquete.nombre-clase.miembro-static;`
  - `import static paquete.nombre-clase.*;`

14.6.2. Importar miembros static de nuestras propias clases

Ejemplo de import static: P106/MyMsg/Msg.java

```
package MyMsg;
public class Msg {
  public static final int UPPER = 1;
  public static final int LOWER = 2;
  public static final int MIXED = 3;
  private String msg;

  // Display a message in the specified case.
  public void showMsg(int how) {
    String str;
    switch(how) {
      case UPPER:
        str = msg.toUpperCase();
        break;
      case LOWER:
        str = msg.toLowerCase();
        break;
      case MIXED:
        str = msg;
        break;
      default:
        System.out.println("Invalid command.");
        return;
    }
    System.out.println(str);
  }
  public Msg(String s) { msg = s; }
}
```

Ejemplo de import static: P106/Test.java

```
// Static import user-defined static fields.
import MyMsg.*;

import static MyMsg.Msg.*;

class Test {
  public static void main(String args[]) {
    Msg m = new Msg("Testing static import.");

    m.showMsg(MIXED);
    m.showMsg(LOWER);
    m.showMsg(UPPER);
  }
}
```

14.6.3. Ambigüedad

- Si importamos dos clases o interfaces que contienen ambos un miembro static con el mismo nombre, tendremos *ambigüedad*.
- Por ejemplo, si el paquete **MyMsg** incluye también la clase:

  `package MyMsg;`

```
public class X {
  public static final int UPPER = 11;
  // ...
}
```

Si ahora importamos los miembros static de las dos clases, el compilador dara un error de ambigüedad:

```
import static MyMsg.Msg.*;
import static MyMsg.X.*;
```


14.7. Annotations (Metadata)

- Las anotaciones permiten asociar datos adicionales en el código fuente, a las clases, interfaces, métodos y campos.
- Estos datos adicionales pueden ser leidos del código fuente, de los ficheros .class o por el API de *reflexión* (*reflection*).
- La semántica del programa no cambia con las anotaciones.

Ejemplo de obtención de anotaciones en tiempo de ejecución: P107/Meta.java

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
class Meta {
  // Annotate a method.
  @MyAnno(str = "Annotation Example", val = 100)
  public static void myMeth() {
    Meta ob = new Meta();
    // Obtain the annotation for this method
    // and display the values of the members.
    try {
      // First, get a Class object that represents
      // this class.
      Class c = ob.getClass();
      // Now, get a Method object that represents
      // this method.
      Method m = c.getMethod("myMeth");

      // Next, get the annotation for this class.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Finally, display the values.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }
  public static void main(String args[]) {
    myMeth();
  }
}
```

14.8. Entrada/Salida formateada

- Java 2, v5.0 proporciona una nueva forma de formatear la salida similar al **printf** de C/C++, que se guardará en un **String**, fichero u otro destino.

- También añade la capacidad de leer entrada formateada, indepedientemente de si viene de un fichero, teclado, **String** u otra fuente.
- Java 2, v5.0 proporciona las clases **Formatter** y **Scanner** para la entrada/salida formateada.

Ejemplo de salida formateada: P108/FormatDemo.java

```
// A very simple example that uses Formatter.
import java.util.*;

class FormatDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);

    System.out.println(fmt);
  }
}
```

Ejemplo de salida formateada usando printf: P109/PrintDemo.java

```
// Demonstrate printf().

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();

    System.out.printf("Default floating-point format: %f\n",
                      1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                      1234567.123);
    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line-up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);

  }
}
```

Ejemplo de uso de Scanner: P110/AvgNums.java

```
// Use Scanner to compute an average of the values.
import java.util.*;
```



```
class AvgNums {
  public static void main(String args[]) {
    Scanner conin = new Scanner(System.in);

    int count = 0;
    double sum = 0.0;

    System.out.println("Enter numbers to average.");

    // Read and sum numbers.
    while(conin.hasNext()) {
      if(conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      }
      else {
        String str = conin.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Data format error.");
          return;
        }
      }
    }

    System.out.println("Average is " + sum / count);
  }
}
```