FORMACIÓN Y TECNOLOGÍAS JAVA

UNIVERSIDAD DE ALICANTE

Curso de Programación en Lenguaje Java

Apuntes

Grupo jTech@ua | Depto. Ciencia de la Computación e Inteligencia Artificial | Universidad de Alicante
Tfno: 965903900 | Fax: 965903902 | e-mail: jtech@dccia.ua.es | URL: http://www.jtech.ua.es

1 Introducción a Java 5
1.1 Compilación y ejecución de programas Java 5
1.1.1 Java como lenguaje interpretado 5
1.1.2 Interpretación y compilación con JDK 11
Compilar y ejecutar clases 12
1.2. Introduccion a la Programación Orientada a Objetos (POO) 14
1.2.1. Objetos y clases 15
1.2.2. Herencia y polimorfismo 16
1.2.4. Clases abstractas e interfaces 17
1.3 Sintaxis de Java 20
1.3.4. Programas Básicos en Java 24
1.4 Eclipse: un entorno gráfico para desarrollo Java 29
1.4.1 Instalación y ejecución 29
1.4.2 Configuración visual: perspectivas, vistas y editores 31
1.4.3 Configuración general 32
1.4.4 Espacio de trabajo 38
1.4.5 Proyectos Java 39
1.4.6 El editor de código 43
1.4.7 Plugins en Eclipse 44
1.5 Tipos de datos 45
1.5.1 Enumeraciones e iteradores 46
1.5.2 Colecciones 47
1.5.3 Wrappers de tipos básicos 55
1.5.4 Clases útiles 55
1.6 Algunos consejos 61
2. Características básicas 62
2.1. Excepciones 62
2.1.1. Tipos de excepciones 62
2.1.2. Captura de excepciones 63
2.1.3. Lanzamiento de excepciones 64
2.1.4. Creación de nuevas excepciones 66
2.2. Hilos 67
2.2.1. Creación de hilos 67
2.2.2. Estado y propiedades de los hilos 68
2.2.3. Sincronización de hilos 70
2.2.4. Grupos de hilos 71
2.3. Entrada/salida 72
2.3.1 Flujos de datos de entrada/salida 72
2.3.2. Entrada, salida y salida de error estándar 73
2.3.3. Acceso a ficheros 74
2.3.4. Lectura de tokens 76
2.3.5. Acceso a ficheros o recursos dentro de un JAR 78
2.3.6. Codificación de datos 79
2.3.7. Serialización de objetos 79
3. Interfaz Gráfica 81
3.1. AWT 81
3.1.1. Introducción a AWT 81
3.1.2. Gestores de disposición 83
3.1.3. Modelo de Eventos en Java 84
3.1.4. Pasos generales para construir una aplicación gráfica con AWT 88
3.2. Swing 91

3.2.1. Introducción a Swing 91
3.2.2. Características específicas de Swing 91

3.3. Applets 94
3.3.1. Applets Swing 95

3.4. Gráficos y animación 95
3.4.1. Gráficos en AWT 96
3.4.2 Contexto gráfico: Graphics 97
3.4.3 Animaciones 101
3.4.4 API de Java 2D 106
3.4.5 Modo a pantalla completa 108
3.4.6 Sonido y música. Java Sound 111

1 Introducción a Java

1.1 Compilación y ejecución de programas Java

1.1.1 Java como lenguaje interpretado

Java es un lenguaje de programación creado por *Sun Microsystems* para poder funcionar en distintos tipos de procesadores. Es un lenguaje orientado a objetos. Su sintaxis es muy parecida a la de C o C++, e incorpora como propias algunas características que en otros lenguajes son extensiones: gestión de hilos, ejecución remota, etc.

El código Java, una vez compilado, puede llevarse sin modificación alguna sobre cualquier sistema operativo (Windows, Linux, Mac OS X, IBM, ...), y ejecutarlo allí. Esto se debe a que el código se compila a un lenguaje intermedio (llamado bytecodes) independiente de la máquina. Este lenguaje intermedio es interpretado por el intérprete Java, denominado Java Virtual Machine (JVM), que deberá existir en la plataforma en la que queramos ejecutar el código. La siguiente figura ilustra el proceso.

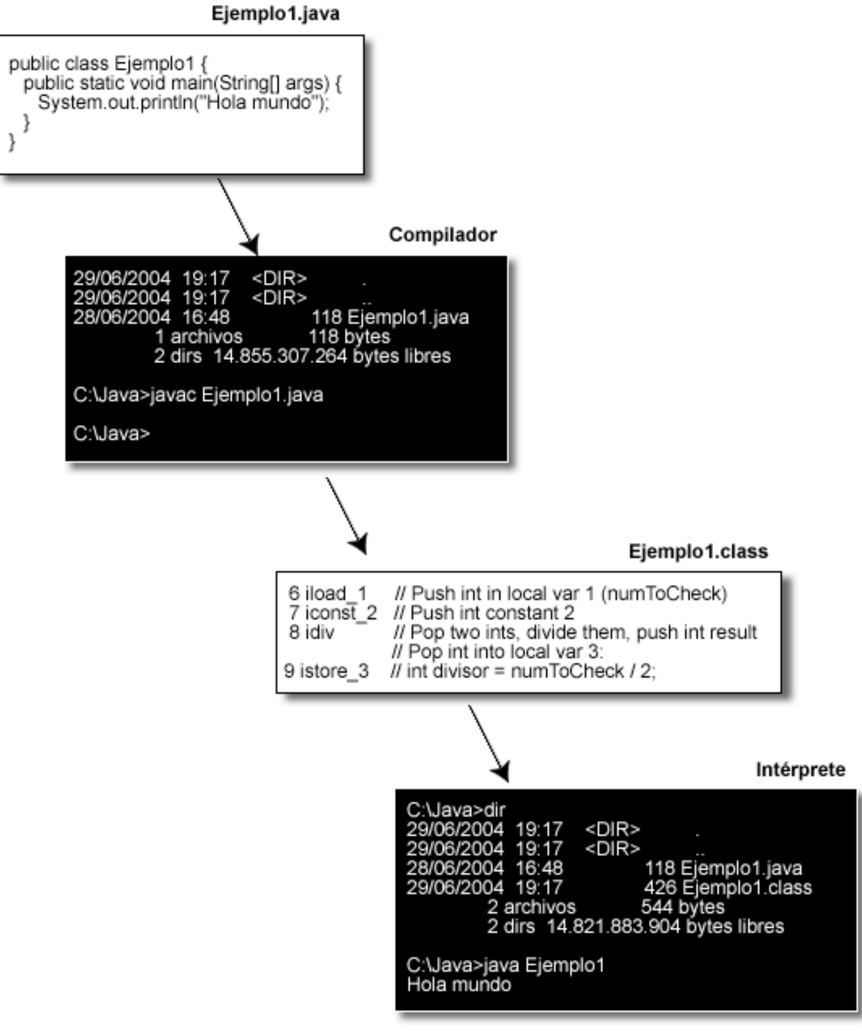


Figura 1.1.1.1 Proceso de compilación y ejecución de un programa Java

El hecho de que la ejecución de los programas Java sea realizada por un intérprete, en lugar de ser código nativo, ha generado la suposición de que los programas Java son más lentos que programas escritos en otros lenguajes compilados (como C o C++). Aunque esto es cierto en algunos casos, se ha avanzado mucho en la tecnología de interpretación de bytecodes y en cada nueva versión de Java se introducen optimizaciones en este funcionamiento. En la última versión de Java, 1.5 (ahora todavía en beta), se introduce una nueva JVM servidora que queda residente en el sistema. Esta máquina virtual permite ejecutar más de un programa Java al mismo tiempo, mejorando mucho el manejo de la memoria. Por último, es posible encontrar bastantes benchmarks en donde los programas Java son más rápidos que programas C++ en algunos aspectos.

Programas Java

Los ficheros fuente de Java tienen la extensión `.java`. Cada fichero `.java` define una clase Java pública (y, posiblemente, más de una clase privada usada por la clase pública). En el apartado siguiente realizaremos una introducción a la programación orientada a objetos (OO). Los ficheros bytecodes generados por la compilación tienen la extensión `.class`. Un fichero `.java` puede generar más de un fichero `.class`, si en el fichero `.java` se define más de una clase. El nombre del fichero .java debe corresponder con el nombre de la clase pública definida en él.

Las clases (ficheros `.class`) se organizan en paquetes. Un paquete contiene un conjunto de clases. A su vez, un paquete puede contener a otros paquetes. La estructura es similar a la de los directorios y ficheros. De hecho, puedes entender el nombre *package* como sinónimo de *directorio*. Los ficheros hacen el papel de las clases Java y los directorios hacen el papel de paquetes. La estructura de directorios en la que se organizan los ficheros `.class` (estructura física del sistema operativo) debe corresponderse con la estructura de paquetes definida en los ficheros fuente `.java`.

Por ejemplo, supongamos el siguiente fichero llamado `Punto2D.java`:

```
package simuladorRobot.geometria;
  public class Punto2D {
     double x;
     double y;

     Punto2D() {
         x = 0;
         y = 0;
     }
     Punto2D(double x, double y) {
         this.x = x;
         this.y = y;
     }
     void traslada(double incX, double incY) {
         x = x + incX;
         y = y + incY;
     }
  }
```

En él estamos declarando la clase Java `Punto2D` en el paquete `simuladorRobot.geometria`. Esto obliga a que el fichero `Punto2D.java` esté situado en el directorio `geometria` que, a su vez, deberá residir en el directorio `simuladorRobot`. Esto es, el fichero Punto2D.java debe tener como path:

```
simuladorRobot/geometria/Punto2D.java (en Linux)
simuladorRobot\geometria\Punto2D.java (en Windows)
```

¿Dónde colocar el directorio `simuladorRobot` para que el intérprete de la máquina virtual Java lo encuentre? El directorio se puede colocar en un directorio estándar de la distribución de Java en el que la máquina virtual busca las clases. O también se puede colocar en un directorio cualquiera e indicar a la máquina virtual que en ese directorio se encuentra un paquete Java. Para ello se utiliza la variable del entorno **CLASSPATH**.

Los directorios de paquetes y los ficheros de clases pueden compactarse en ficheros JAR (por ejemplo `simuladorRobot.jar`). Un fichero JAR es un fichero de archivo (como ZIP o TAR) que contiene comprimidos un conjunto de directorios y ficheros, o sea un conjunto de paquetes y clases. Es normal comprimir toda una librería de clases y paquetes comunes en un único fichero JAR. Ya veremos más adelante cómo crear ficheros JAR.

Para que el compilador y el intérprete pueda usar las clases de un fichero JAR, hay que incluir su camino (incluyendo el nombre del propio fichero JAR) en el **CLASSPATH**.

API de Java

Cuando se programa con Java, se dispone de antemano de un conjunto de clases ya implementadas. Estas clases (aparte de las que pueda hacer el usuario) forman parte del propio lenguaje (lo que se conoce como **API** (*Application Programming Interface*) de Java).

Una herramienta muy útil son las páginas HTML con la documentación del API de Java. Puedes encontrar estas páginas en los recursos del curso. Vamos a usar en concreto la versión 1.4.2 de Java.

Si consultamos la página principal de la documentación, veremos el enlace "Java 2 Platform API Specification" dentro del apartado "API & Language Documentation". Siguiendo ese enlace, aparece la siguiente página HTML. Es una página con tres frames. En la zona superior del lateral izquierdo se listan todos los paquetes de la versión 1.4.2 de Java. La zona inferior muestra una lista con todas las clases existentes en el API. La zona principal describe todos los paquetes existentes en la plataforma.

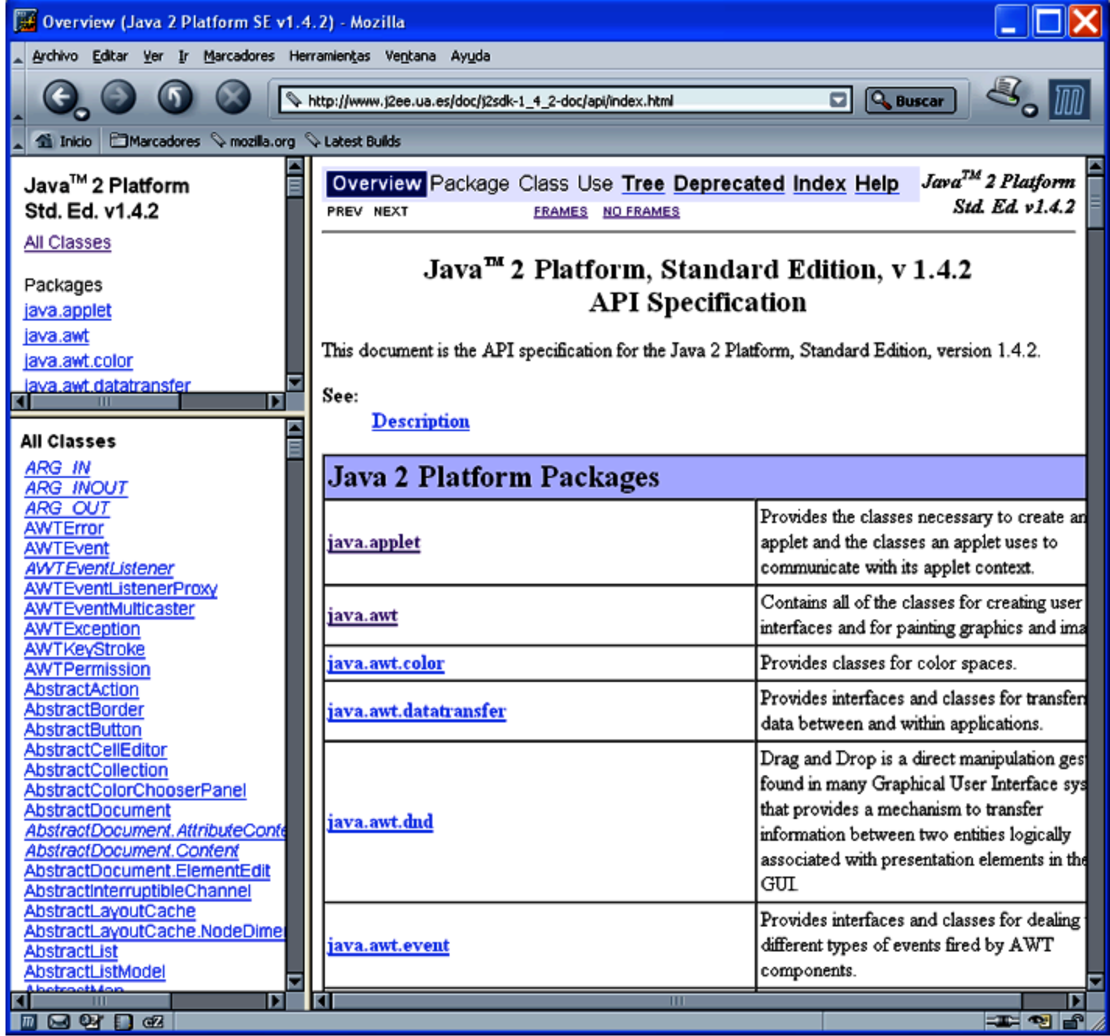


Figura 1.1.1.2 El API de Java 1.4.2

Si seleccionamos un paquete, por ejemplo `java.rmi`, aparece la siguiente página HTML. En el frame inferior izquierdo aparecen los elementos que constituyen el paquete: las clases, interfaces y excepciones definidas en el mismo. En el frame principal se describen con más detalle estos elementos. Todos los elementos están enlazados a la página en la que se detalla la clase, el interface o la excepción.

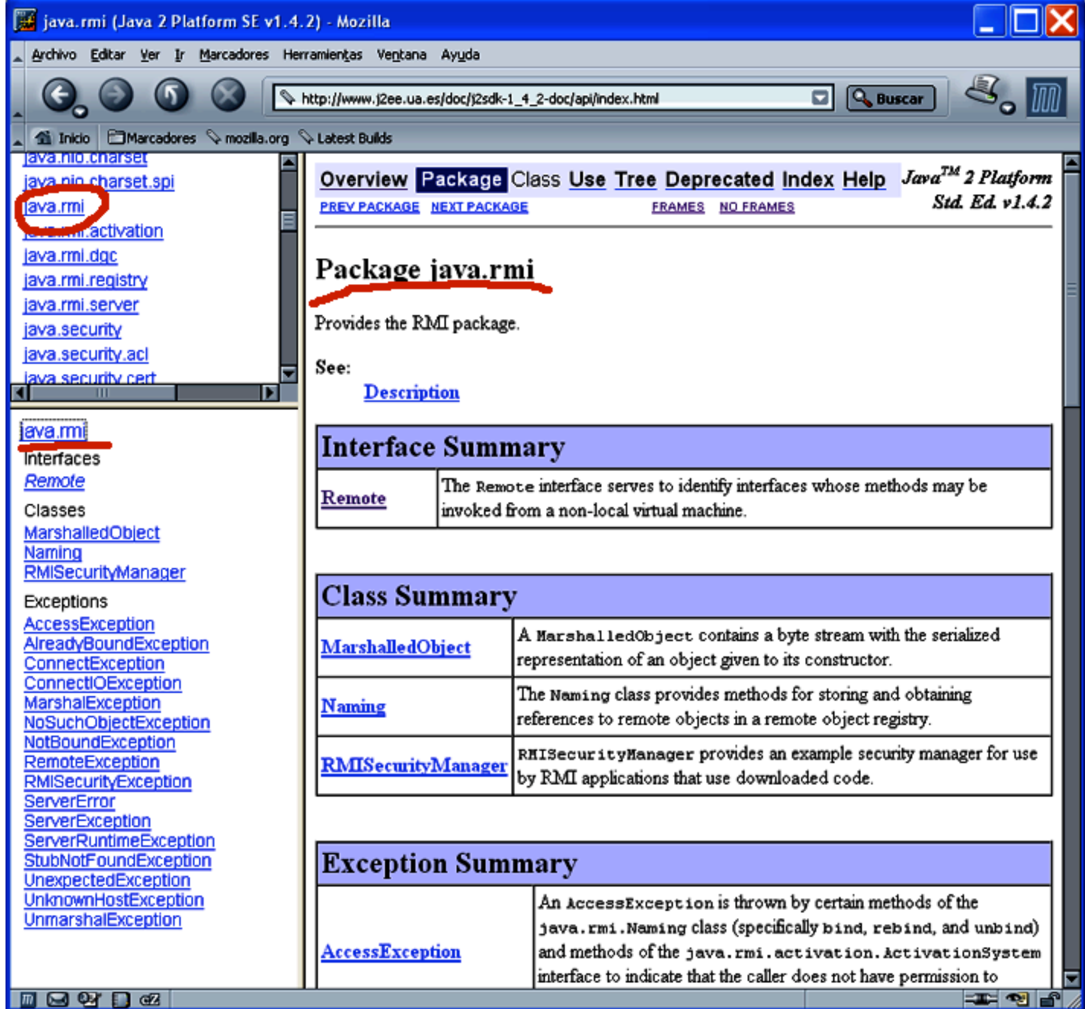


Figura 1.1.1.3 Descripción de un paquete

Cuando escogemos una clase, por ejemplo la clase `Integer` del paquete `java.lang`, aparece una página como la siguiente. En la ventana principal se muestra la jerarquía de la clase, todas las interfaces que implementa la clase y sus elementos constituyentes: campos, constructores y métodos (ver figura 1.1.1.5). En este caso, la clase `Integer` hereda de la clase `Number` (en el paquete `java.lang`), la cual hereda de la clase `Object` (también en el paquete `java.lang`). La clase `Integer` implementa la interfaz `Comparable` y la interfaz `Serializable` (porque es implementada por la clase `Number`).

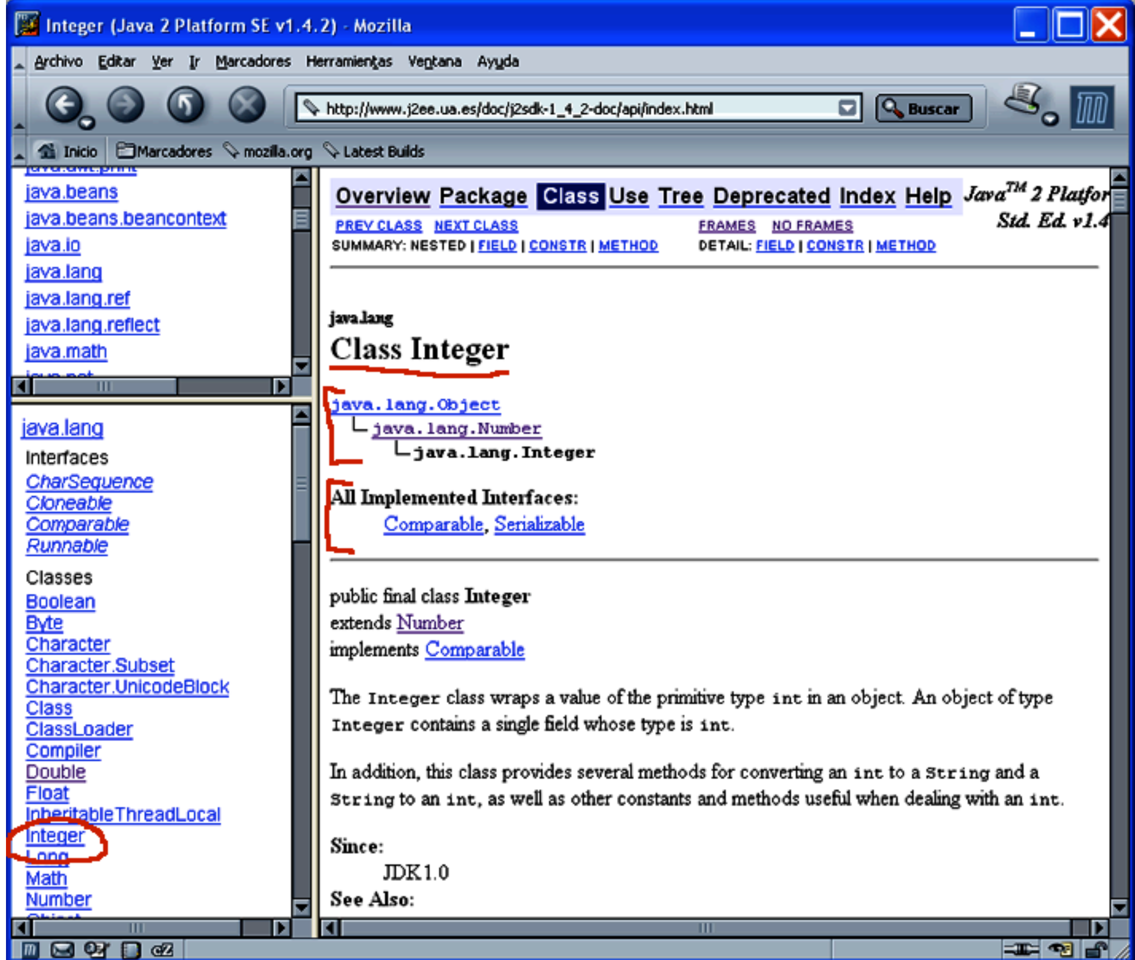


Figura 1.1.1.4 Descripción de una clase

En la figura siguiente se detallan algunos elementos que componen la clase `Integer`.

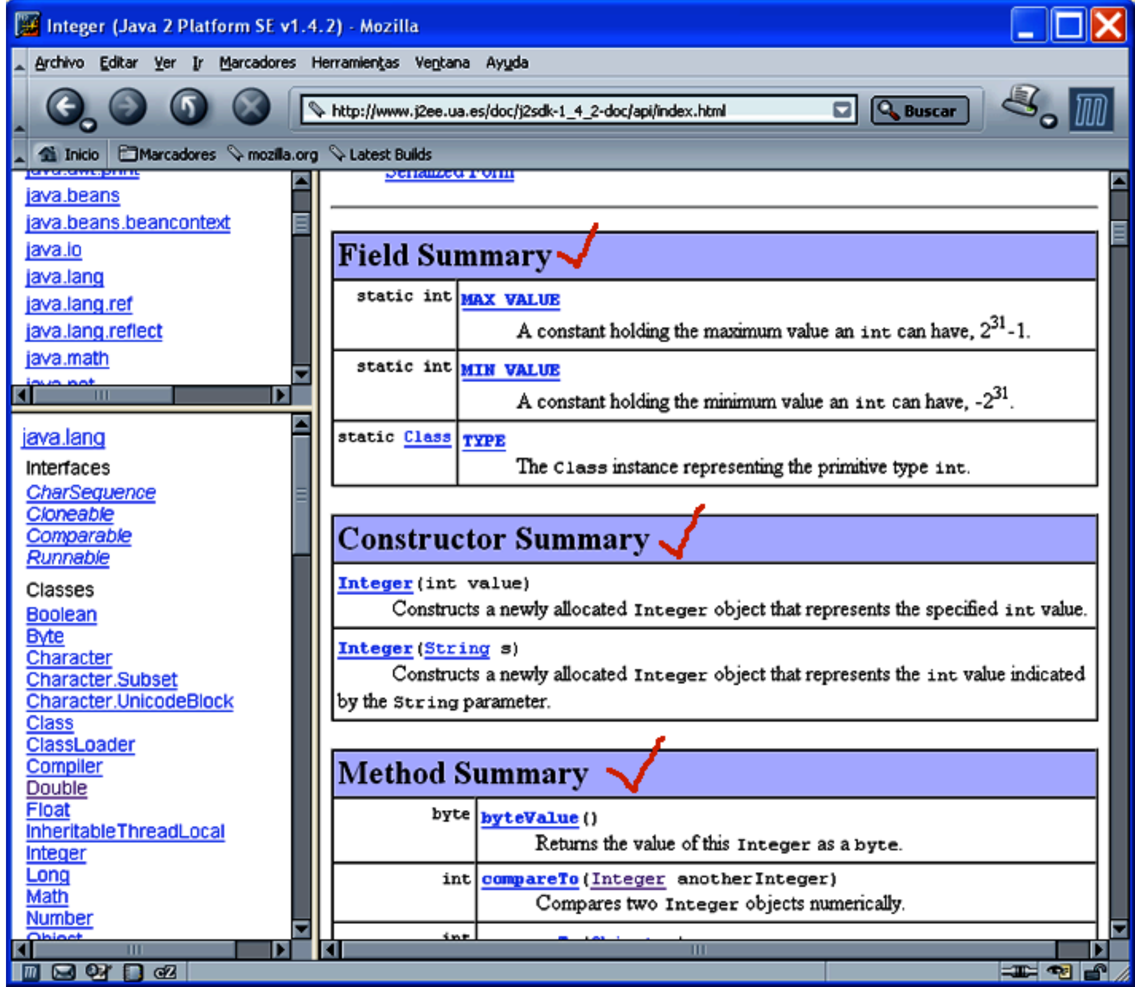


Figura 1.1.1.5 Elementos de una clase

1.1.2 Interpretación y compilación con JDK

Para compilar y ejecutar programas Java necesitamos la distribución **JDK** (*Java Development Kit*) de *Sun.* Es necesario tener instalada esta distribución para poder trabajar con otros entornos de desarrollo, puesto que dichos entornos se apoyan en la API de clases que viene con JDK.

La **instalación** es bastante sencilla (tanto en Windows como en Linux). En windows habrá que elegir el directorio donde instalar, y en Linux en general se descomprime en el lugar que se quiera. También es recomendable instalar (descomprimir) la **documentación** de la API

Variables de entorno

Para su correcto funcionamiento, Java necesita tener establecidas algunas variables de entorno: las variables **PATH** y **CLASSPATH**.

La variable de entorno del sistema **PATH** deberá contener la ruta donde se encuentren los programas para compilar y ejecutar con JDK (*javac* y *java*, respectivamente). Por ejemplo:

```
set PATH=%PATH%;C:\j2sdk1.4.2_02\bin (Windows)
export PATH=$PATH:/j2sdk1.4.2_02/bin (Linux)
```

Con la variable **CLASSPATH** indicamos en qué directorios debe buscar el intérprete de Java las clases compiladas. Por defecto, si CLASSPATH no está definido, las busca en el directorio actual. Puede haber más de un directorio, separando sus caminos por el separador del sistema operativo (";" en Windows y ":" en Linux). Por ejemplo, si las clases que queremos usar están en \misclases:

```
set CLASSPATH=.;C:\misclases (Windows)
export CLASSPATH=.:/misclases
```

Si las clases pertenecen a un paquete concreto, se debe apuntar al directorio a partir del cual comienzan los directorios del paquete. Por ejemplo, si la clase *MiClase* está en el paquete *unpaquete*, dentro de *\mispaquetes* (*\mispaquetes\unpaquete\MiClase.class*):

```
set CLASSPATH=.;C:\misclases;C:\mispaquetes (Windows)
export CLASSPATH=.:/misclases:/mispaquetes (Linux)
```

Si las clases están empaquetadas en un fichero *JAR*, se tendrá que hacer referencia a dicho fichero. Por ejemplo:

```
set CLASSPATH=.;C:\misclases\misclases.jar (Windows)
export CLASSPATH=.:/misclases/misclases.jar (Linux)
```

Para hacer estos cambios permanentes deberemos modificar los ficheros de autoarranque de cada sistema operativo, añadiendo las líneas correspondientes en *autoexec.bat* (para Windows) o *.profile* (para Linux).

La forma de establecer las variables cambia en función de la versión de Windows o Linux. Por ejemplo, en Windows 2000 o XP se pueden establecer variables de entorno directamente desde el panel de control. Y en versiones distintas de Linux se utilizan distintos shells con comandos de establecimiento distintos al *export* (ver información más detallada en el apéndice 2).

Compilar y ejecutar clases

Para compilar y ejecutra las clases Java se usan los programas **javac** y **java** que proporciona el SDK.

Veamos el siguiente programa Java. Se trata de un sencillo programa ejemplo en el que se define una clase `Persona` con varios métodos. Uno de ellos es el constructor (crea objetos de tipo `Persona`) y otro es el método estático `main` que hace que la clase sea ejecutable directamente por el intérprete Java.

```
/**
 * Ejemplo de clase Java
 */
public class Persona {
    public String nombre;
    int edad;

    /**
```

```
     * Constructor
     */
    public Persona() {
        nombre = "Pepe";
        edad = 33;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    public static void main(String[] args) {
        // Datos de la persona
        Persona p = new Persona();
        System.out.println("Nombre de persona: " +
p.getNombre());
        p.setNombre("Maria");
        System.out.println("Nombre de persona: " +
p.getNombre());
        System.out.println("Edad de persona: " + p.getEdad());
        // al ser el campo nombre publico, tambien puedo
        // cambiarlo y leerlo accediendo directamente
        p.nombre = "Pepa";
        System.out.println("Nombre de persona: " + p.nombre);
    }
}
```

Si queremos compilar este fichero debemos llamar a **javac**:

```
javac Persona.java
```

Tras haber compilado el ejemplo se tendrá un fichero `Persona.class` . Ejecutamos el programa con **java** :

```
java Persona
```

Si se quisieran pasar parámetros a un programa Java (no es el caso del ejemplo anterior), se pasan después de la clase:

```
java Persona 20 56 Hola
```

También podemos ejecutar un fichero JAR, si contiene una clase principal. Para ello pondremos:

```
java -jar Fichero.jar
```

Veamos otro ejemplo. Supongamos los siguientes ficheros:

```
./Persona.java
./animales/Elefante.class
./insectos/Mosca.class
./maspersonas.jar
```

Vemos que sólo tenemos el código fuente de una clase (`Persona.java`) y el resto son clases compiladas (la clase `Elefante` en el paquete `animales`, la clase `Mosca` en el paquete `insectos` y el fichero JAR `maspersonas.jar` que contiene la clase `OtraPersona`). Para compilar la clase `Persona.java` verás que no hace falta tener los códigos fuentes de las otras clases, sólo sus ficheros compilados. Hay que incluir en el CLASSPATH el directorio actual y el fichero JAR:

```
set CLASSPATH=.:.\maspersonas.jar (Windows)
export CLASSPATH=.:./maspersonas.jar (Linux)
```

Luego se compila y ejecuta igual que en el ejemplo anterior.

Notar que para compilar se pone la extensión del fichero (*.java*), pero para ejecutar no se pone la extensión *.class,* ya que para ejecutar una clase Java hay que pasarle al intérprete el nombre de una clase, no el nombre del fichero de bytecodes. Los nombres de ficheros que pasemos para compilar y ejecutar **deben coincidir en mayúsculas y minúsculas** con los nombres reales.

1.2. Introduccion a la Programación Orientada a Objetos (POO)

En Programación Orientada a Objetos (POO) un programa es un conjunto de objetos interactuando entre si. Cada objeto (también denominado *instancia*) guarda un estado (mediante sus campos, también llamados variables de instancia) y proporciona un conjunto de métodos con los que puede ejecutar una conducta. Tanto los métodos como los campos de un objeto vienen definidas en su clase.

Supongamos la clase Persona definida en el ejemplo anterior. En esa clase se definen los campos `nombre` y `edad`. También se definen los métodos `Persona` (es el constructor, que sirve para crear nuevos objetos de esta clase), `getNombre()` y `getEdad()` que devuelven la información del objeto y por último los métodos `setNombre(nombre)` y `setEdad(edad)` que modifican la información del objeto.

En POO debemos pensar que los objetos encapsulan (contienen) tanto los datos como los métodos que modifican estos datos. Así, una instancia de la clase `Persona` contiene los datos `nombre` y `edad` y los métodos `getNombre`, `getEdad`, `setNombre` y `setEdad`. El siguiente código crea un objeto de la clase `Persona` y lo guarda en la variable `unaPersona`. Después se llama al método `setNombre` y `setEdad` del objeto recién creado.

```
Persona unaPersona = new Persona();
unaPersona.setNombre("Juan P_rez"); unaPersona.setEdad(12);
```

Después de ejecutar el código, el objeto `unaPersona` tendrá como nombre el *String* "Juan Pérez" (otro objeto) y como edad el entero (*int*) 12. En Java

existen datos primitivos que no son objetos, como por ejemplo *double*, *int*, *char*, etc. Se pueden reconocer porque el nombre del tipo no comienza por mayúscula.

Un último punto a resaltar del ejemplo anterior. Fijémonos en la primera instrucción del ejemplo anterior:

```
Persona unaPersona = new Persona();
```

Hemos dicho que en esta instrucción el objeto de tipo `Persona` recién creado *se guarda en la variable* `unaPersona`. En POO también podemos ver la asignación de otra forma. Podemos ver esta instrucción como una forma de definir un identificador que va a designar el objeto recién creado. Estaríamos entonces diciendo que el identificador del objeto recién creado es `unaPersona`. En esta interpretación, entonces, las asignaciones se convierten en definiciones de identificadores (etiquetas) de objetos. Por ejemplo, si tuviéramos el código

```
Persona otraPersona = unaPersona;
```

estaríamos dando al objeto con el identificador "unaPersona" otro nombre adicional. Así, los nombres "otraPersona" y "unaPersona" se referirían al mismo objeto.

Esta interpretación de pensar en nombres (identificadores) de objetos, en lugar de en variables te será de mucha utilidad en el futuro, si te embarcas en proyectos de programación de componentes distribuidos con Java. Pero esto queda fuera del alcance de este curso.

1.2.1. Objetos y clases

- **Objeto**: conjunto de variables junto con los métodos relacionados con éstas. Contiene la **información** (las variables) y la forma de manipular la información (los métodos).
- **Clase**: prototipo que define las variables y métodos que va a emplear un determinado tipo de objeto.
- **Campos**: contienen la información relativa a la clase
- **Métodos**: permiten manipular dicha información.
- **Constructores**: reservan memoria para almacenar un objeto de esa clase.

La forma de especificar estos elementos en programa Java es la siguiente:

- **Paquetes**: equivalentes a los "include" de C, permiten utilizar clases en otras, y llamarlas de forma abreviada:

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

- **Clases**:

```
public class
MiClase
```

```
{
   ...
```

- **Campos**: Constantes, variables y en general elementos de información.

```
public int a;
Vector v;
```

- **Métodos**: Para las funciones que devuelvan algún tipo de valor, es imprescindible colocar una sentencia *return* en la función.

```
public void imprimirA()
public void insertarVector(String cadena)
```

- **Constructores**: Un tipo de método que siempre tiene el mismo nombre que la clase. Se pueden definir uno o varios.

```
public MiClase()
```

Así, podemos definir una **instancia** con **new**:

```
MiClase mc;
mc = new MiClase ();
mc.a++;
mc.insertarVector("hola");
```

No tenemos que preocuparnos de liberar la memoria del objeto al dejar de utilizarlo. Esto lo hace automáticamente el **garbage collector**. Aún así, podemos usar el método **finalize()** para liberar manualmente.

1.2.2. Herencia y polimorfismo

Con la **herencia** podemos definir una clase a partir de otra que ya exista, de forma que la nueva clase tendrá todas las variables y métodos de la clase a partir de la que se crea, más las variables y métodos nuevos que necesite. A la clase base a partir de la cual se crea la nueva clase se le llama **superclase**.

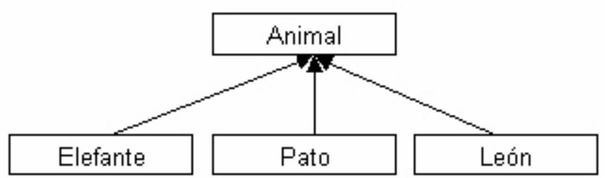


Figura 1.2.2.1 Ejemplo de herencia

Por ejemplo, tenemos una clase genérica *Animal*, y heredamos de ella para formar clases más específicas, como *Pato* , *Elefante*, o *León*. Si tenemos por ejemplo el método *dibuja(Animal a)*, podremos pasarle a este método como parámetro tanto un *Animal* como un *Pato*, *Elefante,* etc. Esto se conoce como **polimorfismo** .

- **Herencia**: Se utiliza la palabra **extends** para decir de qué clase se hereda. Para hacer que *Pato* herede de *Animal*:

```
class Pato extends Animal
```

- **this** se usa para hacer referencia al objeto que está ejecutando el método :

```
public class MiClase {
   int i;
   public MiClase (int i) {
        this.i = i;   // i de la clase = parametro i
   }
}
```

- **super** se usa para hacer referencia al objeto que está ejecutando el método *entendido como un objeto de la clase padre*. Si la clase *MiClase* tiene un método *Suma_a_i(...)*, lo llamamos con:

```
public class MiNuevaClase extends MiClase {
   public void Suma_a_i (int j) {
      i = i + (j / 2);
      super.Suma_a_i (j);
  }
}
```

1.2.4. Clases abstractas e interfaces

Mediante las **clases abstractas** y los **interfaces** podemos definir el esqueleto de una familia de clases, de forma que los subtipos de la clase abstracta o la clase que implemente la interfaz implementen ese esqueleto para dicho subtipo concreto. Por ejemplo, podemos definir en la clase `Animal` el método `dibuja()` y el método `imprime()`, y que `Animal` sea una clase abstracta o un interfaz.

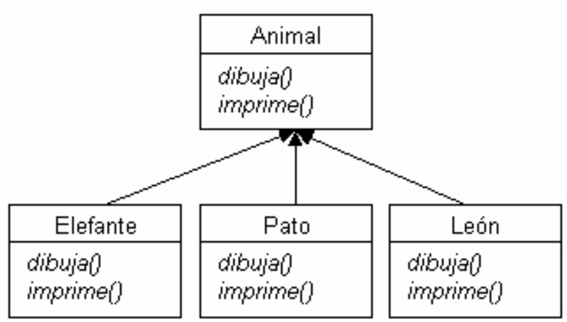


Figura 1.2.4.1 Ejemplo de interfaz y clase abstracta

Vemos la diferencia entre clase, clase abstracta e interfaz con este esquema:

- En una **clase**, al definir `Animal` tendríamos que implementar los métodos `dibuja()` e `imprime()`. Las clases hijas no tendrían por qué implementar

los métodos, a no ser que quieran adaptarlos a sus propias necesidades.

- En una **clase abstracta** podríamos implementar los métodos que nos interese, dejando sin implementar los demás (dejándolos como métodos abstractos). Dichos métodos tendrían que implementarse en las clases hijas.
- En un **interfaz** no podemos implementar ningún método en la clase padre, y cada clase hija tiene que hacer sus propias implementaciones de los métodos. Además, las clases hija podrían implementar otros interfaces.

La especificación en Java es como sigue.

Si queremos definir una clase (por ejemplo, `Animal`), como clase abstracta y otra clase (por ejemplo, `Pato`) que hereda de esta clase, debemos declararlo así:

```
public abstract class Animal
{
   abstract void dibujar ();
   void imprimir () { codigo; }
}
```

```
public class Pato extends Animal
{
   void dibujar() { codigo; }
}
```

Si en lugar de definir `Animal` como clase abstracta, lo definimos como **interfaz**, debemos declarar que la clase `Pato` **implementa la interfaz**, y debemos escribir el código de esa implementación en la clase `Pato`:

```
public interface Animal
{
   void dibujar ();
   void imprimir ();
}
```

```
public class Pato implements Animal
{
   void dibujar() { codigo; }
   void imprimir() { codigo; }
}
```

La diferencia fundamental es que la clase `Pato` puede implementar más de un interfaz, mientras que sólo es posible heredar de una clase padre (en Java no existe la herencia múltiple):

```
public class Pato implements Animal, Volador
{
   void dibujar() { codigo; }  // viene de la interfaz Animal
   void imprimir() { codigo; } // viene de la interfaz Animal
```

```
    void vuela() { codigo; }    // viene de la interfaz Volador
}
```

- **Modificadores**: en algunos elementos (campos, métodos, clases, etc) se utilizan algunos de estos modificadores al declararlos:

  - **public**: cualquier objeto puede acceder al elemento
  - **protected**: sólo pueden acceder las subclases de la clase.
  - **private**: sólo pueden ser accedidos desde dentro de la clase.
  - **abstract**: elemento base para la herencia (los objetos subtipo deberán definir este elemento).
  - **static**: elemento compartido por todos los objetos de la misma clase.

    Si se define **static** en un método estamos definiendo un método estático que puede ser llamado usando la misma clase sin crear ningún objeto (por ejemplo, el método `random()` de la clase `java.lang.Math`). Para ejecutar un método estático de una clase debemos usar la misma sintaxis que para ejecutar un método de un objeto (el operador punto "."), pero ahora sobre la propia clase, en lugar de sobre un objeto:

    ```
    hipotenusa = Math.sqrt (Math.pow(cat1,2),
                            Math.pow(cateto2,2));
    ```

    Si se define **static** en un campo, estamos declarando una variable de clase, accesible por todos los objetos (en el caso de ser public) o sólo por los objetos de la clase (en el caso de ser private).

  - **final**: objeto final, no modificable ni heredable.
  - **synchronized**: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución.

- **Paquetes**: la palabra **package** permite agrupar clases e interfaces. Los nombres de los paquetes son palabras separadas por puntos, y se almacenan en directorios que coinciden con esos nombres. Así, si definimos la clase `MiClase1_1` de la siguiente forma:

  ```
  package paquete1.subpaquete1;
  public class MiClase1_1
  ...
  ```

haremos que la clase *MiClase1_1* pertenezca al subpaquete *subpaquete1* del paquete *paquete1*. Para utilizar las clases de un paquete utilizamos **import**:

```
import java.Date;
import paquete1.subpaquete1.*;
import java.awt.*;
```

Para importar todas las clases del paquete se utiliza el asterisco * (aunque no vayamos a usarlas todas, si utilizamos varias de ellas puede ser útil simplificar

con un asterisco). Si sólo queremos importar una o algunas pocas, se pone un *import* por cada una, terminando el paquete con el nombre de la clase en lugar del asterisco (como pasa con *Date* en el ejemplo).

Al poner *import* podemos utilizar el nombre corto de la clase. Es decir, si ponemos:

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

Podemos hacer referencia a un objeto *Date* o a un objeto *Vector* (una clase del paquete *java.util*) con:

```
Date d = ...
Vector v = ...
```

Si no pusiéramos los *import*, deberíamos hacer referencia a los objetos con:

```
java.Date d = ...
java.util.Vector v = ...
```

Es decir, cada vez que queramos poner el nombre de la clase, deberíamos colocar todo el nombre, con los paquetes y subpaquetes.

1.3 Sintaxis de Java

Tipos de datos

Se tienen los siguientes tipos de datos simples. Además, se pueden crear complejos, todos los cuales serán subtipos de **Object**

Tipo	Tamaño/Formato	Descripción	Ejemplos
byte	8 bits, complemento a 2	Entero de 1 byte	210, 0x456
short	16 bits, complemento a 2	Entero corto	"
int	32 bits, complemento a 2	Entero	"
long	64 bits, complemento a 2	Entero largo	"
float	32 bits, IEEE 754	Real simple precisión	3.12, 3.2E13
double	64 bits, IEEE 754	Real doble precisión	"
char	16 bits, carácter	Carácter simple	'a'
String		Cadena de caracteres	"cadena"
boolean	true / false	verdadero / falso	true, false

Arrays

Se definen arrays o conjuntos de elementos de forma similar a como se hace en C. Hay 2 métodos:

```
int a[] = new int [10];
String s[] = {"Hola", "Adios"};
```

No pueden crearse arrays estáticos en tiempo de compilación (int a[8];), ni rellenar un array sin definir previamente su tamaño con el operador **new**. La función miembro **length** se puede utilizar para conocer la longitud del array:

```
int a [][] = new int [10] [3];
a.length;          // Devolvería 10
a[0].length;       // Devolvería 3
```

Los arrays empiezan a numerarse desde 0, hasta el tope definido menos uno (como en C).

Identificadores

Nombran variables, funciones, clases y objetos. Comienzan por una letra, carácter de subrayado '_' o símbolo '$'. El resto de caracteres pueden ser letras o dígitos (o '_'). Se distinguen mayúsculas de minúsculas, y no hay longitud máxima. Las variables en Java sólo son válidas desde el punto donde se declaran hasta el final de la sentencia compuesta (las llaves) que la engloba. No se puede declarar una variable con igual nombre que una de ámbito exterior.

En Java se tiene también un término NULL, pero si bien el de C es con mayúsculas (NULL), éste es con minúsculas (*null*):

```
String a = null;
...
if (a == null)...
```

Referencias

En Java no existen punteros, simplemente se crea otro objeto que referencie al que queremos "apuntar".

Código	Descripción
`MiClase mc = new MiClase(); MiClase mc2 = mc;`	mc2 y mc apuntan a la misma variable (al cambiar una cambiará la otra).
`MiClase mc = new MiClase(); MiClase mc2 = new MiClase();`	Tendremos dos objetos apuntando a elementos diferentes en memoria.

Comentarios

```
// comentarios para una sola línea

/* comentarios de
   una o más líneas */

/** comentarios de documentación para javadoc,
    de una o más líneas */
```

Operadores

Se muestra una tabla con los operadores en orden de precedencia

<table>
<tr><th>Operador</th><th>Ejemplo</th><th>Descripción</th></tr>
<tr><td>.</td><td><code>a.length</code></td><td>Campo o método de objeto</td></tr>
<tr><td>[ ]</td><td><code>a[6]</code></td><td>Referencia a elemento de array</td></tr>
<tr><td>( )</td><td><code>(a + b)</code></td><td>Agrupación de operaciones</td></tr>
<tr><td>++ , --</td><td><code>a++; b--</code></td><td>Autoincremento / Autodecremento de 1 unidad</td></tr>
<tr><td>!, ~</td><td><code>!a ; ~b</code></td><td>Negación / Complemento</td></tr>
<tr><td>instanceof</td><td><code>a instanceof TipoDato</code></td><td>Indica si a es del tipo TipoDato</td></tr>
<tr><td>*, /, %</td><td><code>a*b; b/c; c%a</code></td><td>Multiplicación, división y resto de división entera</td></tr>
<tr><td>+, -</td><td><code>a+b; b-c</code></td><td>Suma y resta</td></tr>
<tr><td><<, >></td><td><code>a>>2; b<<1</code></td><td>Desplazamiento de bits a izquierda y derecha</td></tr>
<tr><td><, >, <=, >=, ==, !=</td><td><code>a>b; b==c; c!=a</code></td><td>Comparaciones (mayor, menor, igual, distinto...)</td></tr>
<tr><td>&, |, ^</td><td><code>a&b; b|c</code></td><td>AND, OR y XOR lógicas</td></tr>
<tr><td>&&, ||</td><td><code>a&&b; b||c</code></td><td>AND y OR condicionales</td></tr>
<tr><td>?:</td><td><code>a?b:c</code></td><td>Condicional: si a entonces b , si no c</td></tr>
<tr><td>=, +=, -=, *=, /= ...</td><td><code>a=b; b*=c</code></td><td>Asignación. a += b equivale a (a = a + b)</td></tr>
</table>

Control de flujo

TOMA DE DECISIONES

Este tipo de sentencias definen el código que debe ejecutarse si se cumple una determinada condición. Se dispone de sentencias **if** y de sentencias **switch**:

<table>
<tr><th>Sintaxis</th><th>Ejemplos</th></tr>
<tr><td><pre>
if (condicion1) {
   sentencias;
} else if (condicion2) {
   sentencias;
   ...
} else if(condicionN) {
   sentencias;
} else {
   sentencias;
}
</pre></td><td><pre>
if
(a == 1) {
   b++;
} else if (b == 1) {
   c++;
} else if (c == 1) {
   d++;
}
</pre></td></tr>
<tr><td><pre>
switch (condicion) {
   case caso1: sentencias;
   case caso2: sentencias;
   case casoN: sentencias;
   default: sentencias;
}
</pre></td><td><pre>
switch (a) {
   case 1: b++;
           break;
   case 2: c++;
           break;
   default:b--;
           break;
}
</pre></td></tr>
</table>

BUCLES

Para repetir un conjunto de sentencias durante un determinado número de iteraciones se tienen las sentencias **for**, **while** y **do...while** :

Sintaxis	Ejemplo
`for(inicio;condicion; incremento) { sentencias; }`	`for (i=1;i<10;i++) { b = b+i; }`
`while (condicion){ sentencias; }`	`while (i < 10) { b += i; i++; }`
`do{ sentencias; } while (condicion);`	`do { b += i; i++; } while (i < 10);`

SENTENCIAS DE RUPTURA

Se tienen las sentencias **break** (para terminar la ejecución de un bloque o saltar a una etiqueta), **continue** (para forzar una ejecución más de un bloque o saltar a una etiqueta) y **return** (para salir de una función devolviendo o sin devolver un valor):

```
public int miFuncion(int n)
{
    int i = 0;
    while (i < n)
    {
            i++;
            if (i > 10)
                    // Sale del while
                    break;
            if (i < 5)
                    // Fuerza una iteracion mas
                    continue;
    }
    // Devuelve lo que valga i al llegar aquí
    return i;
}
```

1.3.4. Programas Básicos en Java

Veamos ahora algunos ejemplos de programas en Java.

Ejemplo 1

El siguiente ejemplo muestra un texto por pantalla (muestra "Mi programa Java"):

```
/**
  * Ejemplo que muestra un texto por pantalla
  */

public class Ejemplo1
{
   public static void main(String[] args)
   {
      System.out.println ("Mi programa Java");
   }
}
```

Ejemplo 2

El siguiente ejemplo toma dos números (un entero y un real) y devuelve su suma

```
/**
  * Ejemplo que suma dos numeros: un entero y un real
  */

public class Ejemplo2
{
    int n1;         // Primer numero (el entero)
    float n2;       // Segundo numero (el real)

    /**
      * Constructor
      */
    public Ejemplo2(int n1, float n2)
    {
       this.n1 = n1;
       this.n2 = n2;
    }

    public float suma()
    {
       return (n1 + n2);
    }

    /**
      * Main
      */
    public static void main(String[] args) {
       if (args.length < 2) {
          System.out.println ("Uso: java Ejemplo2 <n1> <n2>");
          System.exit(-1);
    }
```

```
    // Tomamos los dos parametros
    int n1 = Integer.parseInt(args[0]);
    float n2 = Float.parseFloat(args[1]);

    // Creamos un objeto Ejemplo2 y le pedimos
    // la suma de los valores

    Ejemplo2 e = new Ejemplo2(n1, n2);
    System.out.println ("Resultado: " + e.suma());
    }
}
```

Ejemplo 3

El siguiente ejemplo resuelve el teorema de pitágoras (obtiene una hipotenusa a partir de dos catetos)

```
/**
  * Este ejemplo resuelve el teorema de Pitágoras:
  * hipotenusa = raiz (cateto1 * cateto1 + cateto2 * cateto2)
  */

public class Ejemplo3
{
    // Primer Cateto
    public static final int CATETO1 = 20;

    // Segundo Cateto
    public static final int CATETO2 = 50;

    /**
      * Obtiene la hipotenusa de dos catetos que se
      * pasan como parametro
      */
    public static double hipotenusa(int cateto1, int cateto2)
    {
       return Math.sqrt(Math.pow(cateto1, 2) +
                        Math.pow(cateto2, 2));
    }

    /**
      * Main
      */
    public static void main(String[] args) {
       System.out.println
           ("La hipotenusa de los catetos indicados es:");
       double h = hipotenusa(CATETO1, CATETO2);
       System.out.println ("h = " + h);
    }
}
```

Ejemplo 4

El siguiente ejemplo devuelve todos los números primos que encuentra hasta un número determinado

```
/**
  * Este ejemplo devuelve los numeros primos encontrados
  * hasta un cierto valor
  */

public class Ejemplo4
{
    /**
      * Obtiene si un número es primo o no
      */
    public static boolean esPrimo (int valor)
    {
       int i = 2;
       while (i < valor) {
          if (valor % i == 0) return false;
          i++;
       }
       return true;
    }

    /**
      * Main
      */
    public static void main(String[] args) {
       System.out.println
          ("Numeros primos hasta el " + args[0] + ":");
       for (int i = 1; i < Integer.parseInt(args[0]); i++)
          if (esPrimo(i))
             System.out.print (" " + i);
          System.out.println ("\nFinalizado");
    }
}
```

Ejemplo 5

El siguiente ejemplo muestra cómo utilizar herencia y clases abstractas. Define una clase abstracta Persona, de la que hereda la clase Hombre. La clase Anciano a su vez hereda de la clase Hombre. En la clase *Herencia* se tiene el método main(), que muestra resultados de llamadas a todas las clases.

Fichero "Persona.java"

```
public abstract class Persona
{
    /**
      * Devuelve la clase a la que pertenecen las personas
      */
    public String clase() {
       return "mamiferos";
    }

    /**
      * Devuelve el genero de la persona
      */
    public abstract String genero();

    /**
      * Devuelve la edad de la persona
      */
    public abstract String edad();
}
```

Fichero "Hombre.java"

```
public class Hombre extends Persona {
    /*
      No hace falta definir el metodo clase(), porque ya esta
      definido en la clase padre. Lo tendríamos que definir si
      queremos devolver algo distinto a lo que devuelve allí
    */

    /**
     * Devuelve el genero de la persona (este metodo si hay que
     * definirlo porque es abstracto en la clase padre)
     */
    public String genero() {
       return "masculino";
    }

    /**
     * Devuelve la edad de la persona (este metodo si hay
      * que definirlo porque es abstracto en la clase padre)
      */
    public String edad() {
       return "40";
    }
}
```

Fichero "Anciano.java"

```
public class Anciano extends Hombre
{
    /*
    No hace falta definir ningun metodo, sólo aquellos en los
    que queramos devolver cosas distintas. En este caso,
    la edad
    */

    /**
      * Devuelve la edad de la persona
      */
    public String edad() {
       return "75";
    }
}
```

Fichero "Herencia.java"

```
public class Herencia
{
    /**
      * Main
      */
    public static void main(String[] args) {
      Hombre h = new Hombre();
      Anciano a = new Anciano();
      Persona p = (Persona)a;

      System.out.println ("Edad del hombre: " + h.edad());
      System.out.println ("Genero del anciano: " + a.genero());
      System.out.println ("Clase de la persona: " + p.clase());
    }
}
```

1.4 Eclipse: un entorno gráfico para desarrollo Java

Eclipse es una herramienta que permite integrar diferentes tipos de **aplicaciones**. La aplicación principal es el JDT (*Java Development Tooling*), un IDE para crear programas en Java. Otras aplicaciones, que no vienen con la distribución estándar de Eclipse, se añaden al mismo en forma de ***plugins***, y son reconocidos automáticamente por la plataforma.

Además, Eclipse tiene su propio mecanismo de gestión de **recursos**. Los recursos son ficheros en el disco duro, que se encuentran alojados en un espacio de trabajo (*workspace*), un directorio especial en el sistema. Así, si una aplicación de Eclipse modifica un recurso, dicho cambio es notificado al resto de aplicaciones de Eclipse, para que lo tengan en cuenta.

1.4.1 Instalación y ejecución

Para instalar Eclipse se **requiere**:

- Sistema operativo Windows, Linux, Solaris, QNX o Mac OS/X, con 256 MB de RAM preferiblemente.
- JDK o JRE versión 1.3 o posterior. Se recomienda al menos la versión 1.4.1.
- Los archivos de eclipse para instalar (en un archivo ZIP, o como vengan distribuidos)

Para la **instalación**, se siguen los pasos:

- Instalar JRE o JDK
- Descomprimir los archivos de Eclipse al lugar deseado del disco duro (p. ej, a *C:\eclipse*). El directorio en que se instale lo identificaremos de ahora en adelante como *ECLIPSE_HOME*. En Windows Eclipse detecta automáticamente un JRE o JDK instalado, aunque también se lo podemos proporcionar copiando el directorio *jre* en *ECLIPSE_HOME*. Otra opción es proporcionar, al ejecutar Eclipse, la ruta hacia JRE o JDK, mediante una opción *–vm*:

```
eclipse -vm ruta_jdk_jre
```

Para arrancar Eclipse se tiene el ejecutable *eclipse.exe* o *eclipse.sh* en *ECLIPSE_HOME*. La pantalla inicial de Eclipse aparecerá tras unos segundos:

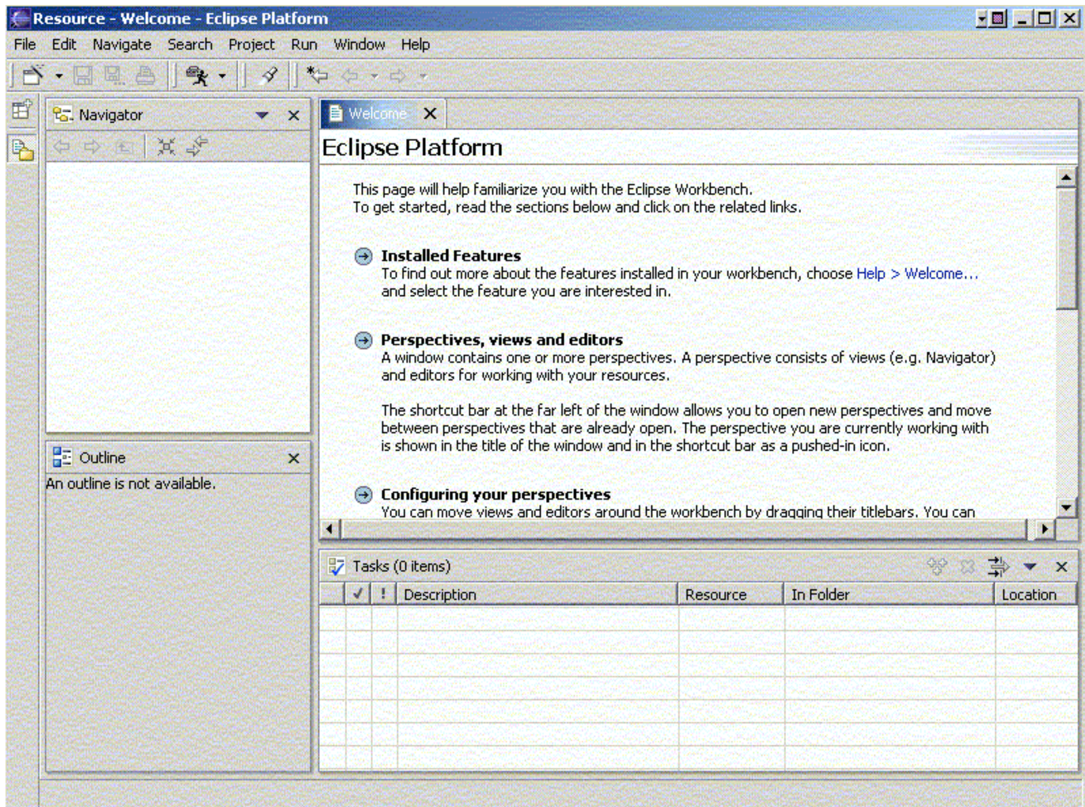


Figura 1.4.1.1 Pantalla inicial de Eclipse

Veremos las opciones principales con detalle más adelante. De los menús, entre otros, pueden resultar interesantes:

- **File**:
  - **New**: para crear nuevos proyectos, paquetes, clases Java, etc.
  - **Import / Export**: para importar o exportar recursos con un determinado formato (por ejemplo, exportar un proyecto como un fichero JAR).
- **Project**:
  - **Open / Close Project**: para abrir o cerrar el proyecto actual
  - **Rebuild Project**: recompila el proyecto actual
  - **Rebuild All**: recompila todos los proyectos
  - **Generate Javadoc**: genera el *javadoc* para las clases del proyecto
- **Run**:
  - **Run As**: permite indicar cómo queremos ejecutar un proyecto (por ejemplo, como una aplicación Java normal, como un applet, como un test de JUnit, etc).
  - **Run**: ejecuta el proyecto de la forma que hayamos indicado en *Run As*. Permite seleccionar la clase principal a ejecutar, los parámetros del *main(...)*, etc
- **Window**:
  - **Open Perspective**: para abrir una determinada perspectiva (por ejemplo, la perspectiva *Java*, que será la que nos interese normalmente).

- **Show View**: permite añadir/quitar vistas a la perspectiva actual. Lo veremos también más adelante.
- **Preferences**: opciones de configuración general. Algunas de ellas se explicarán con detalle más adelante.

1.4.2 Configuración visual: perspectivas, vistas y editores

El usuario trabaja con Eclipse mediante el entorno gráfico que se le presenta. Según la perspectiva que elija, se establecerá la apariencia de dicho entorno. Entendemos por ***perspectiva*** una colección de ***vistas*** y ***editores***, con sus correspondientes acciones especiales en menús y barras de herramientas. Algunas vistas muestran información especial sobre los recursos, y dependiendo de las mismas, en ocasiones sólo se mostrarán algunas partes o relaciones internas de dichos recursos. Un editor trabaja directamente sobre un recurso, y sólo cuando grabe los cambios sobre el recurso se notificará al resto de aplicaciones de Eclipse sobre estos cambios. Las vistas especiales se pueden conectar a editores (no a recursos), por ejemplo, la vista de estructura (*outline view*) se puede conectar al editor Java. De este modo, una de las características importantes de Eclipse es la flexibilidad para combinar vistas y editores.

Si queremos **abrir una determinada perspectiva**, vamos a *Window -> Open Perspective*. Eligiendo luego *Other* podemos elegir entre todas las perspectivas disponibles:

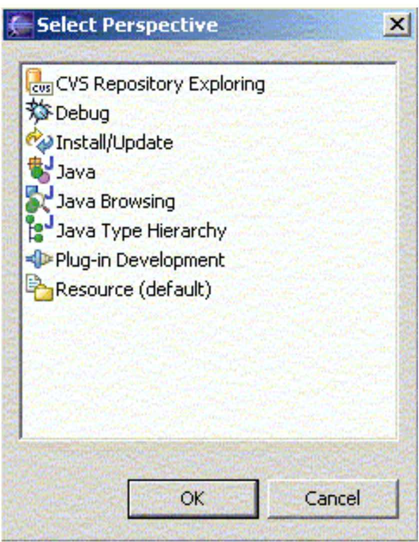


Figura 1.4.2.1 Abrir una perspectiva en Eclipse

Para **añadir vistas a una perspectiva**, primero abrimos la perspectiva, y luego vamos a *Window -> Show View* y elegimos la que queramos cargar:

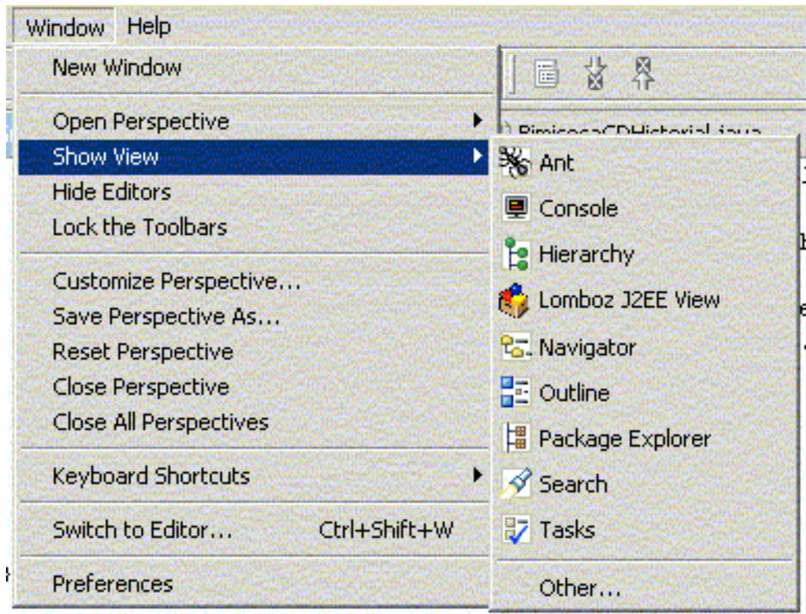


Figura 1.4.2.2 Elegir una vista en una perspectiva

Apariencia

Arrastrando la barra de título de una vista o editor, podemos moverlo a otro lugar de la ventana (lo podremos colocar en las zonas donde el cursor del ratón cambie a una flecha negra), o tabularlo con otras vistas o editores (arrastrando hasta el título de dicha vista o editor, el cursor cambia de aspecto, y se ve como una lista de carpetas, soltando ahí la vista o editor que arrastramos, se tabula con la(s) que hay donde hemos soltado).

1.4.3 Configuración general

Desde el menú **Window - Preferences** podemos establecer opciones de configuración de los distintos aspectos de Eclipse:

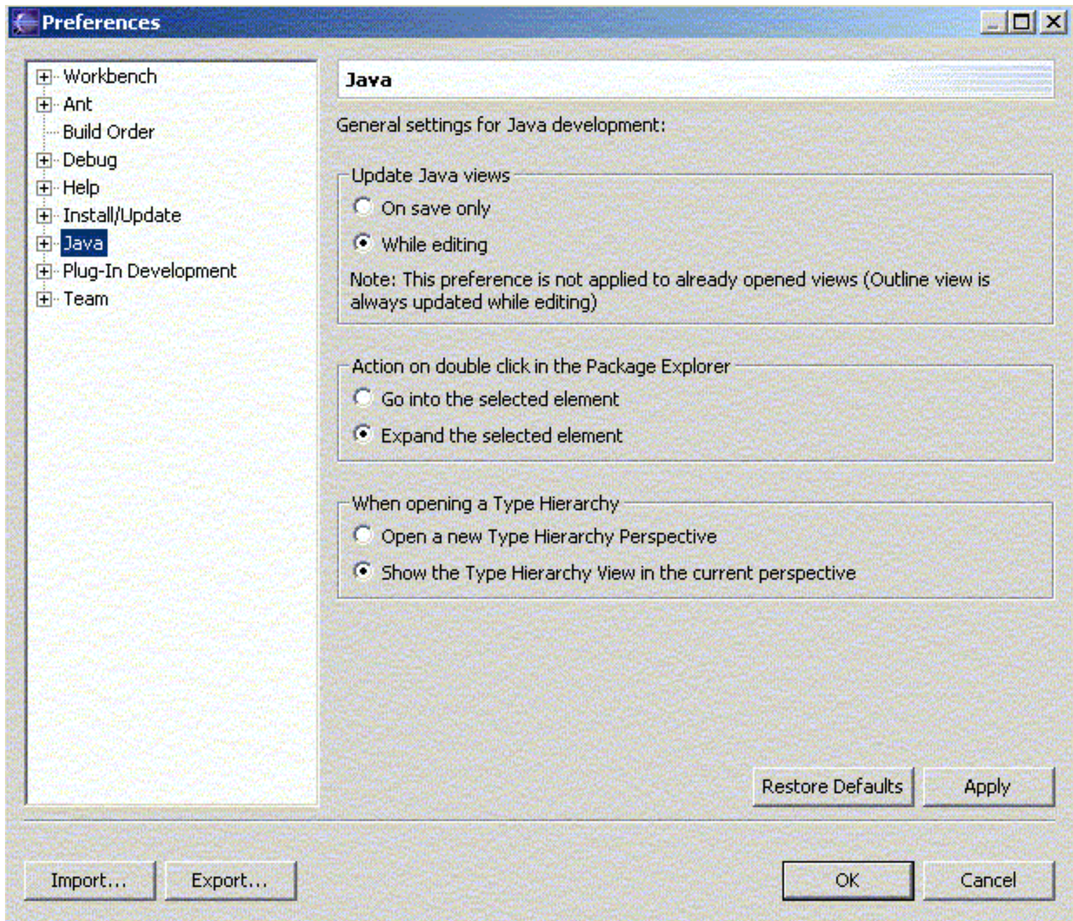


Figura 1.4.3.1 Configuración general de Eclipse

Establecer directorios para ficheros fuente o ficheros objeto

Podemos elegir entre tener nuestro código fuente en el mismo lugar que nuestras clases objeto compiladas, o bien elegir directorios diferentes para fuentes y objetos. Para ello tenemos, dentro del menú de configuración anterior, la opción *Java - New Project*. En el cuadro *Source and output folder* podremos indicar si queremos colocarlo todo junto (marcando *Project*) o indicar un directorio para cada cosa (marcando *Folders*, y eligiendo el subdirectorio adecuado para cada uno):

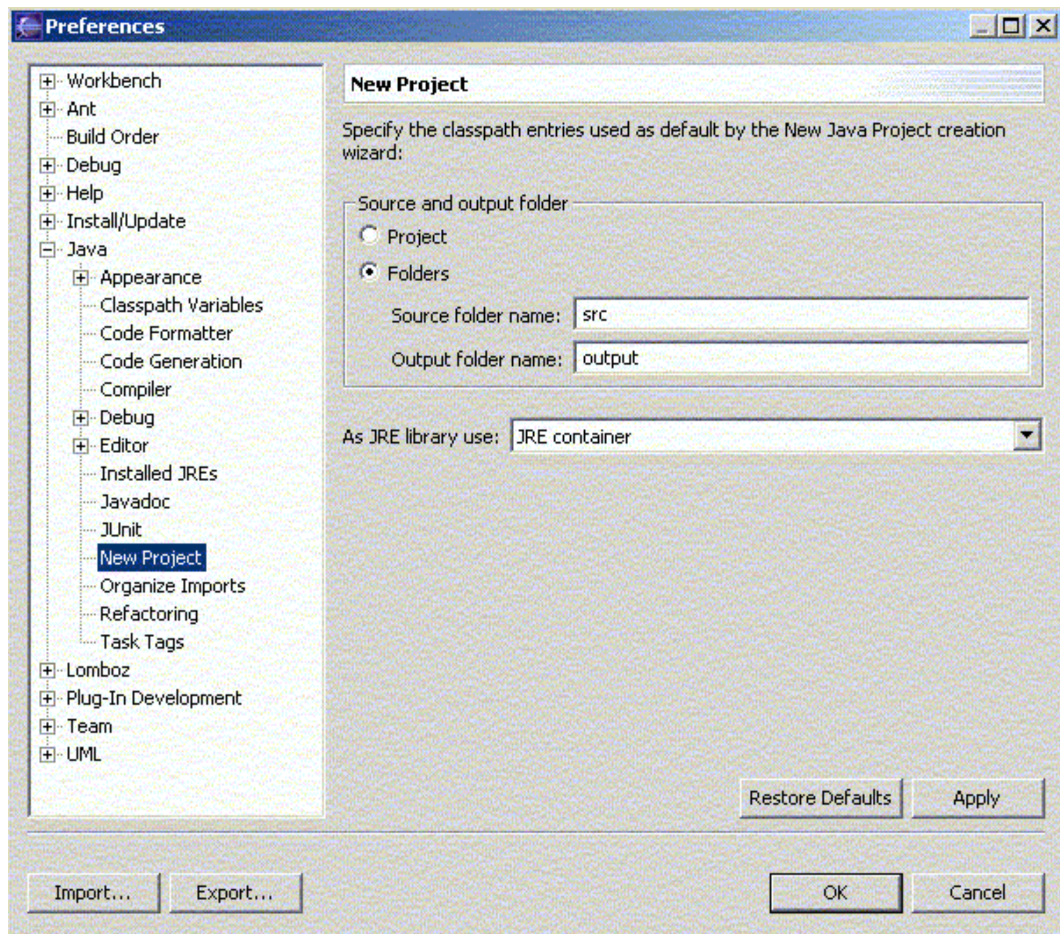


Figura 1.4.3.2. Establecimiento de los directorios fuente y objeto

Establecer la versión de JDK o JRE

Para cambiar el compilador a una versión concreta de Java, elegimos la opción de *Java* y luego *Compiler*. Pulsamos en la pestaña *Compliance and Classfiles* y elegimos la opción *1.4* (o la que sea) de la lista *Compiler compliance level*:

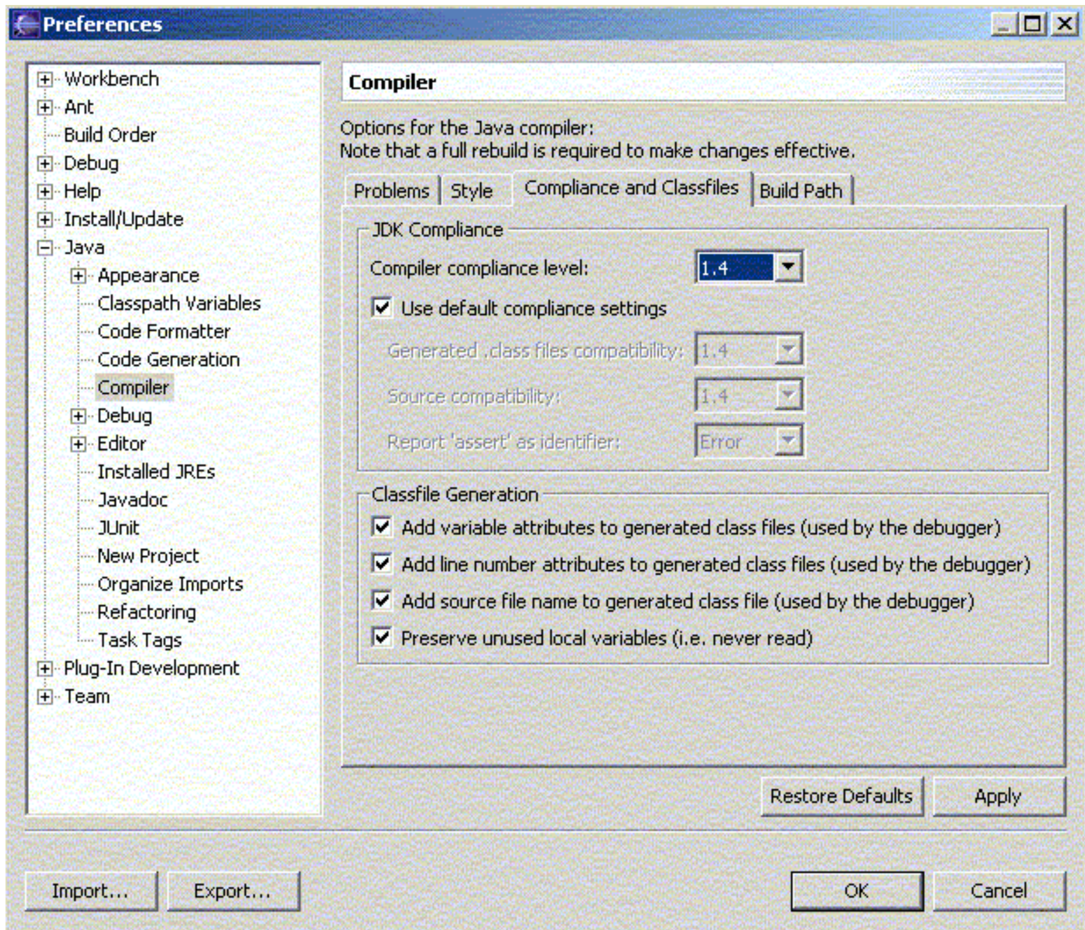


Figura 1.4.3.3 Establecer la versión del compilador

También podemos utilizar JDK en lugar de JRE para ejecutar los programas. Para ello vamos a *Java - Installed JREs*, elegimos la línea *Standard VM* y pulsamos en *Edit* o en *Add*, según si queremos modificar el que haya establecido, o añadir nuevas opciones.

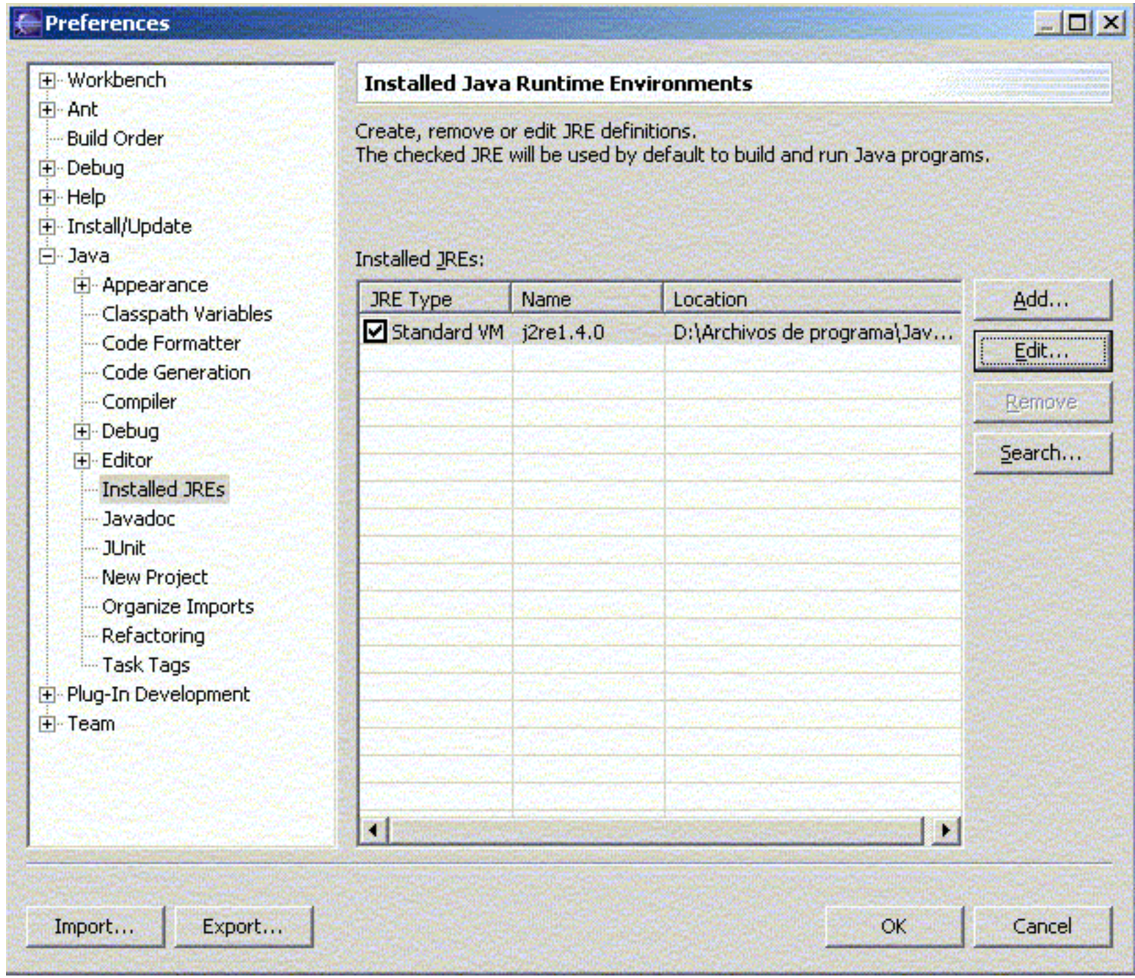


Figura 1.4.3.4 Establecer el compilador, o añadir nuevos

Se nos abre un cuadro de diálogo para editar valores. Pulsando en *Browse* elegimos el directorio de JDK (por ejemplo, *C:\j2sdk1.4.0*).

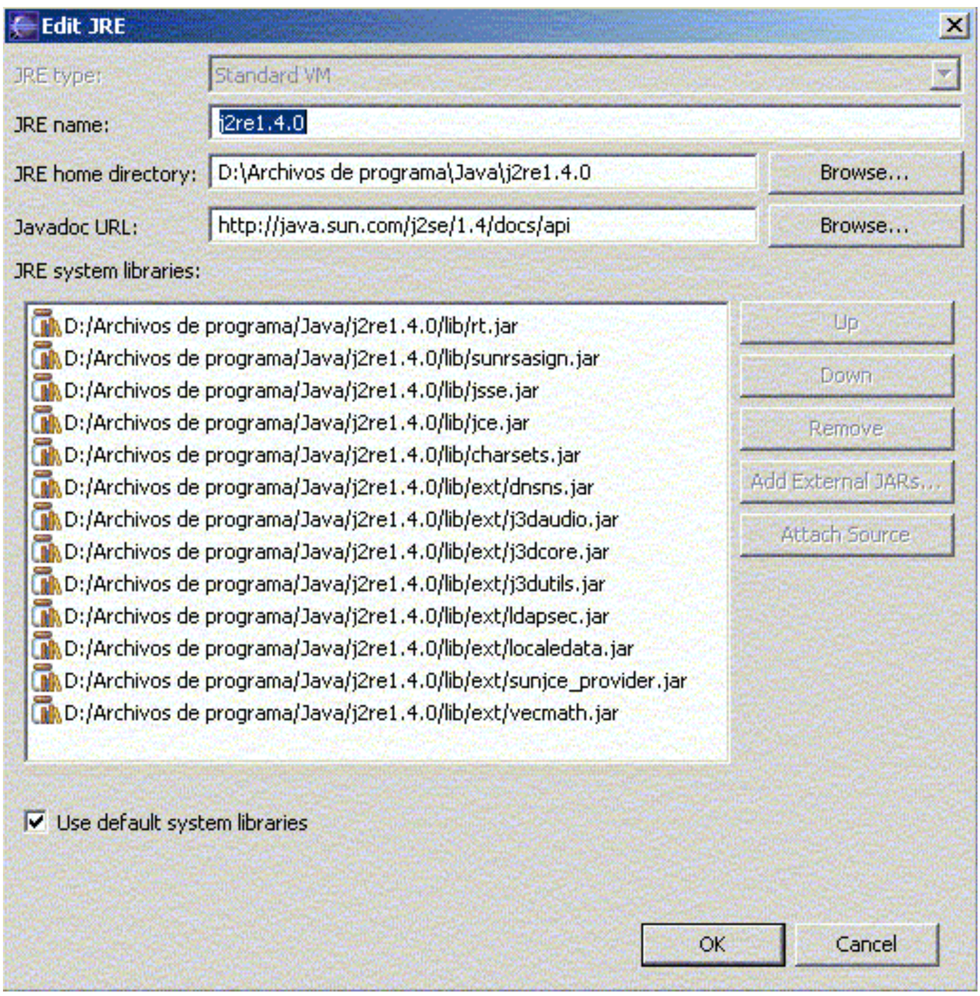


Figura 1.4.3.5 Editar los valores de JDK o JRE

Especificar variables de entorno (CLASSPATH)

Podemos añadir variables de entorno en Eclipse, cada una conteniendo un directorio, fichero JAR o fichero ZIP. Para añadir variables vamos a la opción *Java - Classpath Variables*.

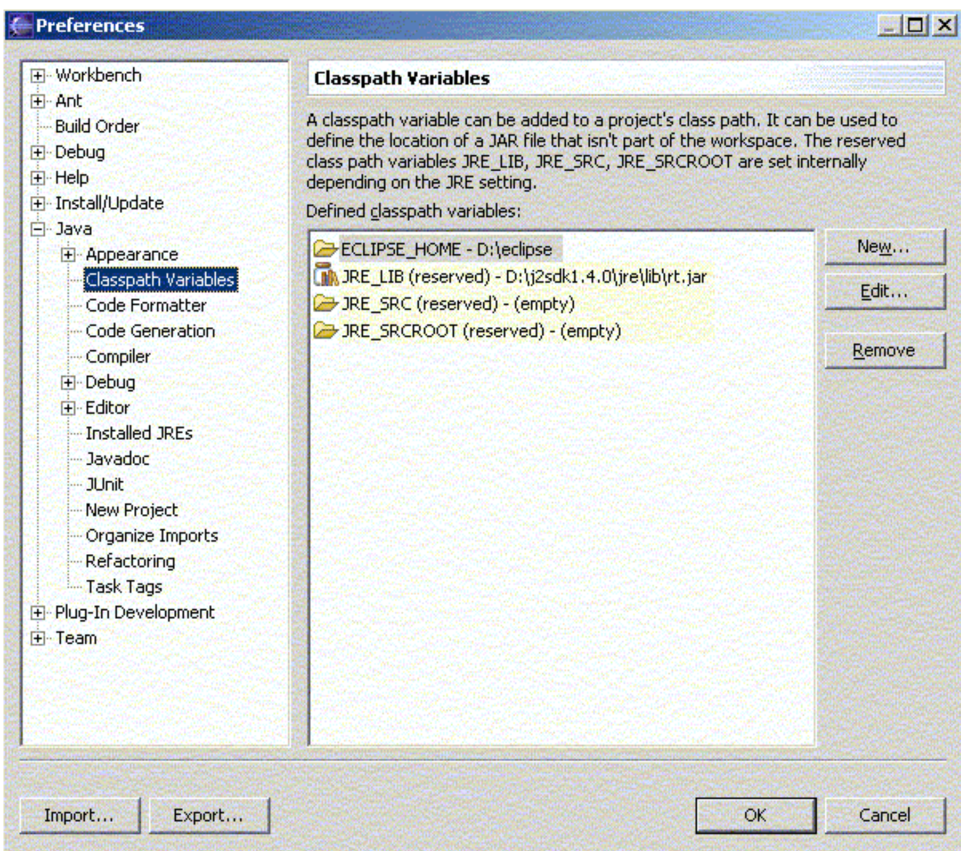


Figura 1.4.3.6 Variables de classpath

Pulsamos el botón de *New* para añadir una nueva, y le damos un nombre, y elegimos el fichero JAR o ZIP (pulsando en *File*) o el directorio (pulsando en *Folder*).

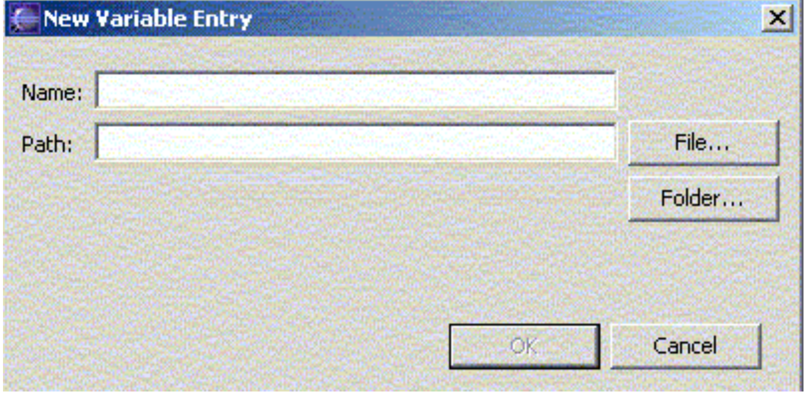


Figura 1.4.3.7 Establecer el valor de la nueva variable

1.4.4 Espacio de trabajo

Por defecto el espacio de trabajo (*workspace*) para Eclipse es el directorio *ECLIPSE_HOME/workspace*. Podemos elegir un directorio arbitrario lanzando eclipse con una opción *–data* que indique cuál es ese directorio, por ejemplo:

```
eclipse -data C:\misTrabajos
```

También podemos crear nuestros proyectos y trabajos fuera del *workspace* si queremos, podemos tomarlo simplemente como un directorio opcional donde organizar nuestros proyectos.

1.4.5 Proyectos Java

Para crear un nuevo proyecto Java vamos a *File -> New -> Project*. Después en el cuadro que aparece elegimos el proyecto que sea (normalmente, un *Java Project* en el grupo *Java*):

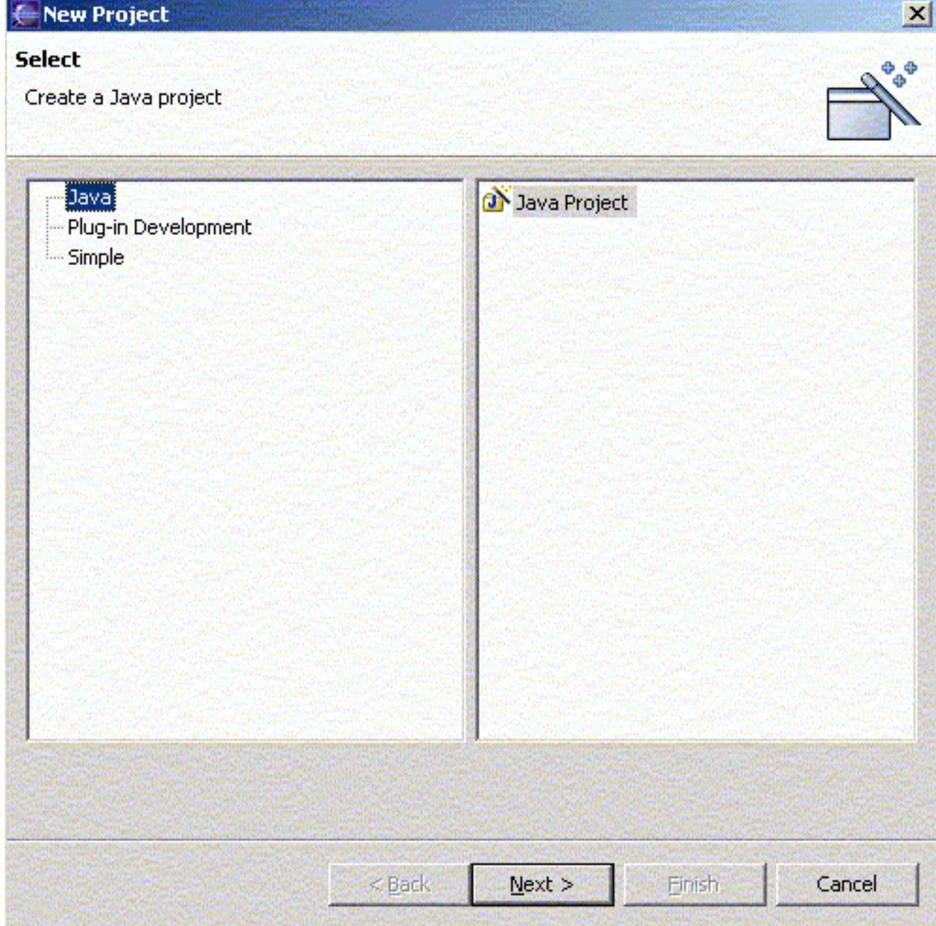


Figura 1.4.4.1 Crear un nuevo proyecto en Eclipse

Después nos aparece otra ventana para elegir el nombre del proyecto, y dónde guardarlo (por defecto en el espacio de trabajo de Eclipse):

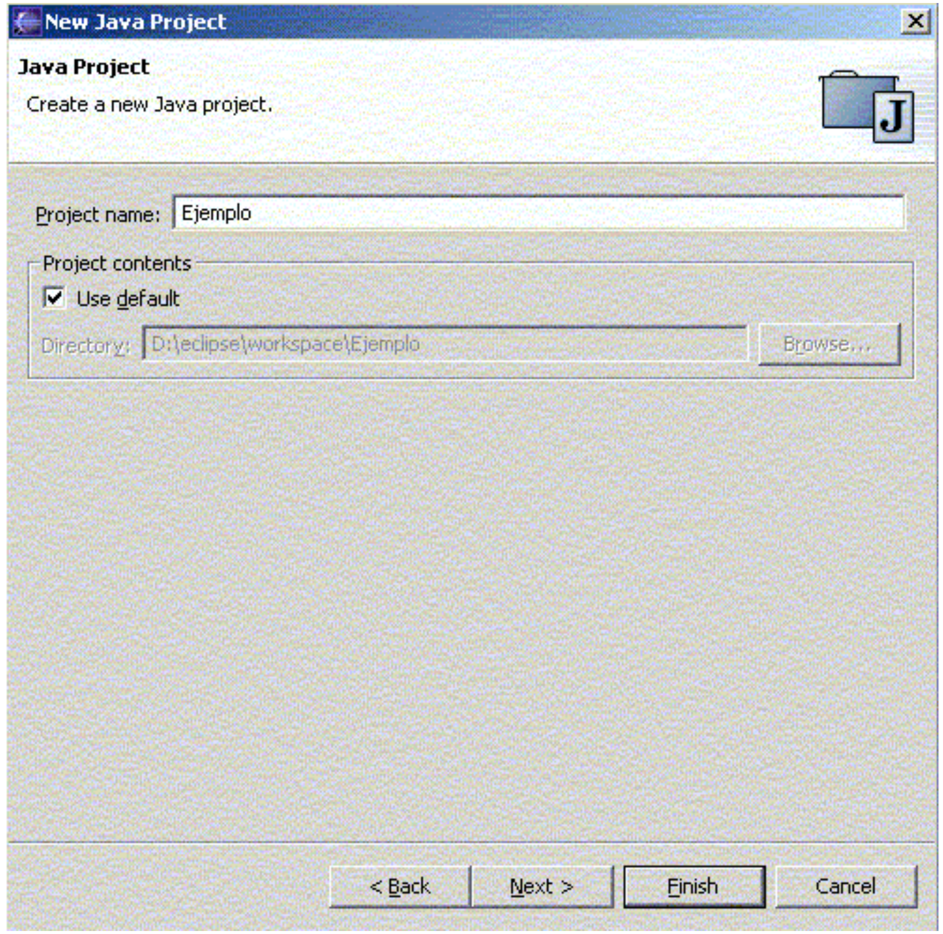


Figura 1.4.4.2 Asignar nombre y ubicación al proyecto

Si pulsamos en *Next* podemos especificar otras opciones del proyecto en otro panel:

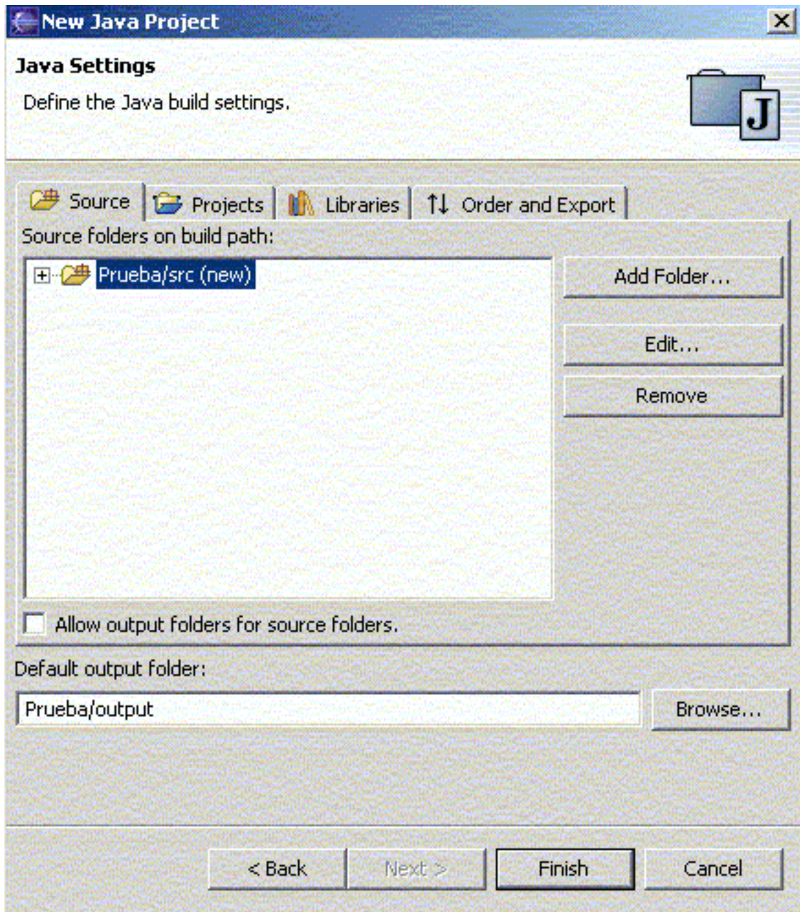


Figura 1.4.4.3 Otras opciones para el proyecto

donde podremos indicar qué carpetas tienen el código del proyecto (*Source*), el directorio donde sacar las clases compiladas (*Default output folder*), el classpath (*Libraries*), etc. Una vez rellenas las opciones a nuestro gusto, ya tendremos el proyecto creado:

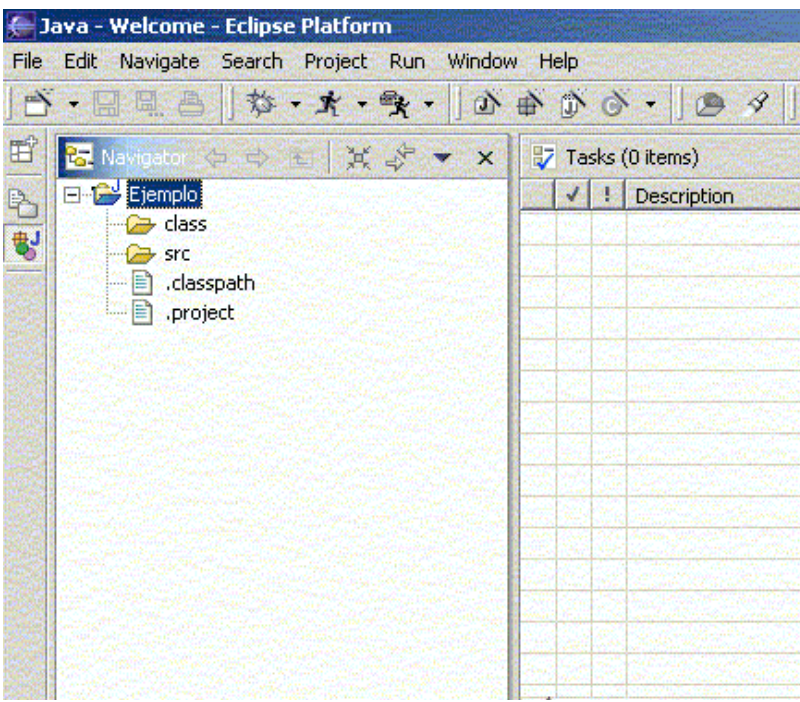


Figura 1.4.4.4 Proyecto creado

Buildpath de un proyecto

Desde el *buildpath* de un proyecto se establecen las clases que debe compilar, los recursos (directorios, ficheros JAR, etc) que debe tener en cuenta para compilarlo, etc. Para establecerlo, hacemos click con el botón derecho sobre el proyecto, y vamos a *Properties*.

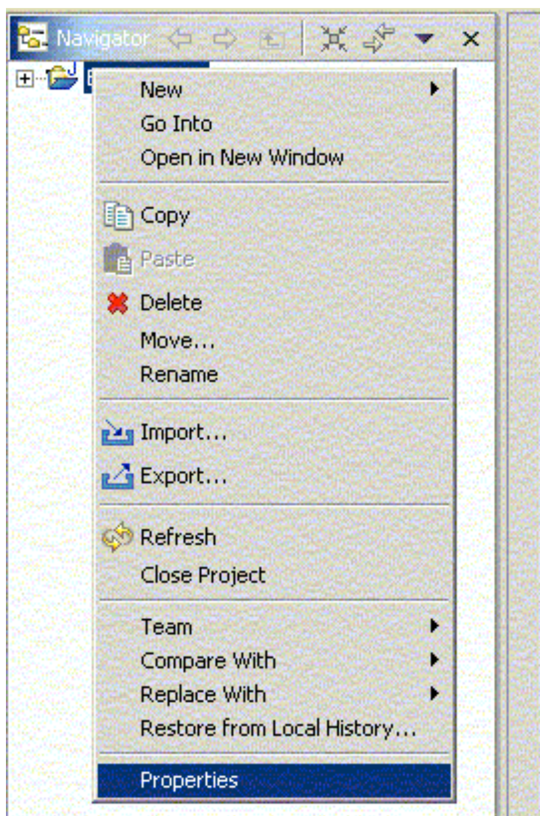


Figura 1.4.4.5 Establecer el buildpath de un proyecto

Nos aparecerá un cuadro con varias pestañas:

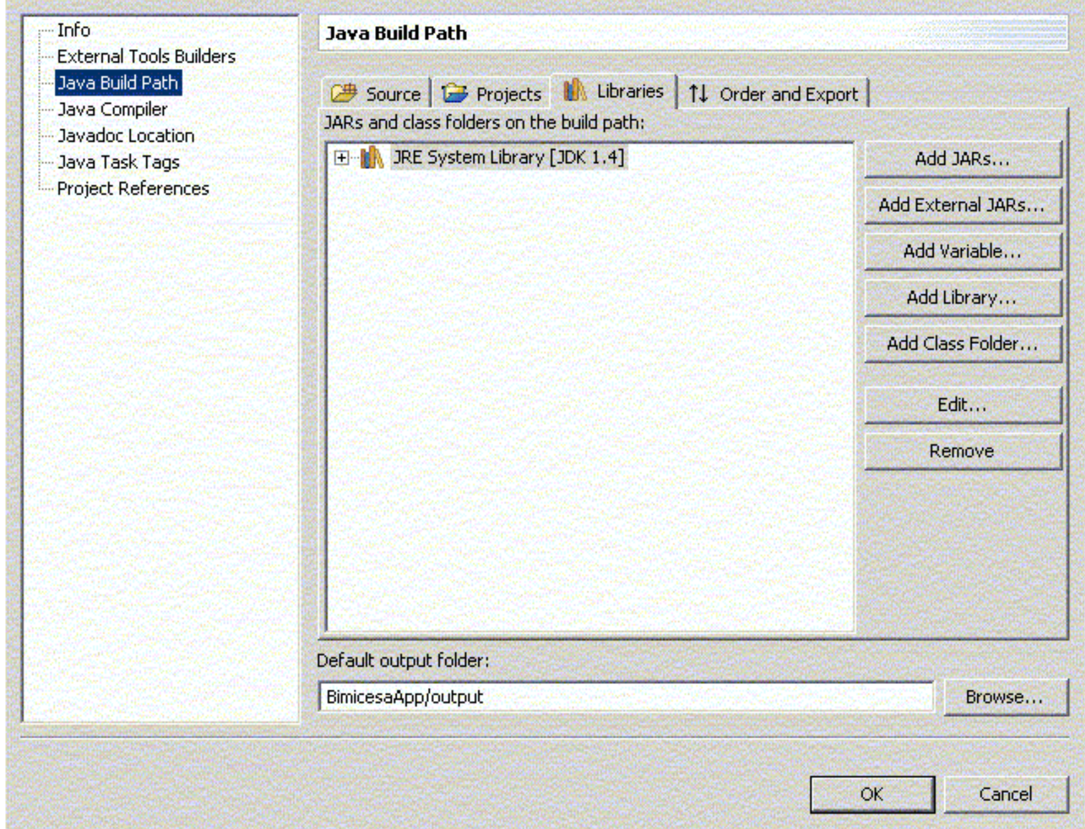


Figura 1.4.4.6 Opciones del buildpath

- En *Source* se indican las carpetas donde hay o puede haber código fuente del proyecto
- En *Projects* podemos indicar qué otros proyectos queremos tener en cuenta a la hora de compilar el actual (el compilador buscará clases en estos proyectos, si las necesita).
- En *Libraries* podemos añadir al classpath ficheros JAR que estén dentro del proyecto (*Add JARs*), ficheros JAR externos al proyecto (*Add External JARs*), o variables que contengan un directorio, JAR o ZIP determinado (*Add Variable*), que hayamos definido desde *Window -> Preferences*.
- En *Order and Export* podemos cambiar el orden en que se buscan las clases.
- En el cuadro inferior *Default output folder* indicamos dónde se colocarán las clases compiladas del proyecto.

1.4.6 El editor de código

El editor de código de Eclipse es bastante sencillo de usar, y dispone de una ayuda contextual que permite autocompletar las sentencias de código que vayamos escribiendo. Por ejemplo, si escribimos un nombre de campo, tras el punto nos mostrará las opciones que podemos escribir después:

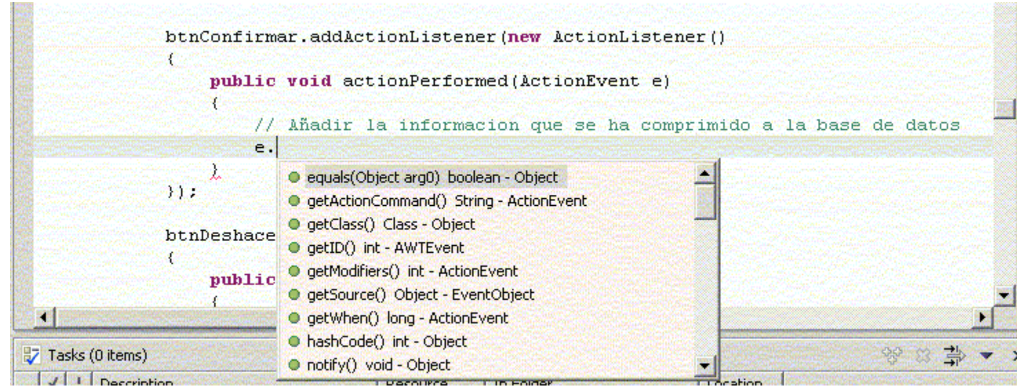


Figura 1.4.6.1 Editor de código de Eclipse

1.4.7 Plugins en Eclipse

Para **instalar nuevos plugins**, simplemente hay que copiarlos en el directorio *ECLIPSE_HOME/plugins.* Después habrá que reiniciar Eclipse para que pueda tomar los nuevos plugins instalados.

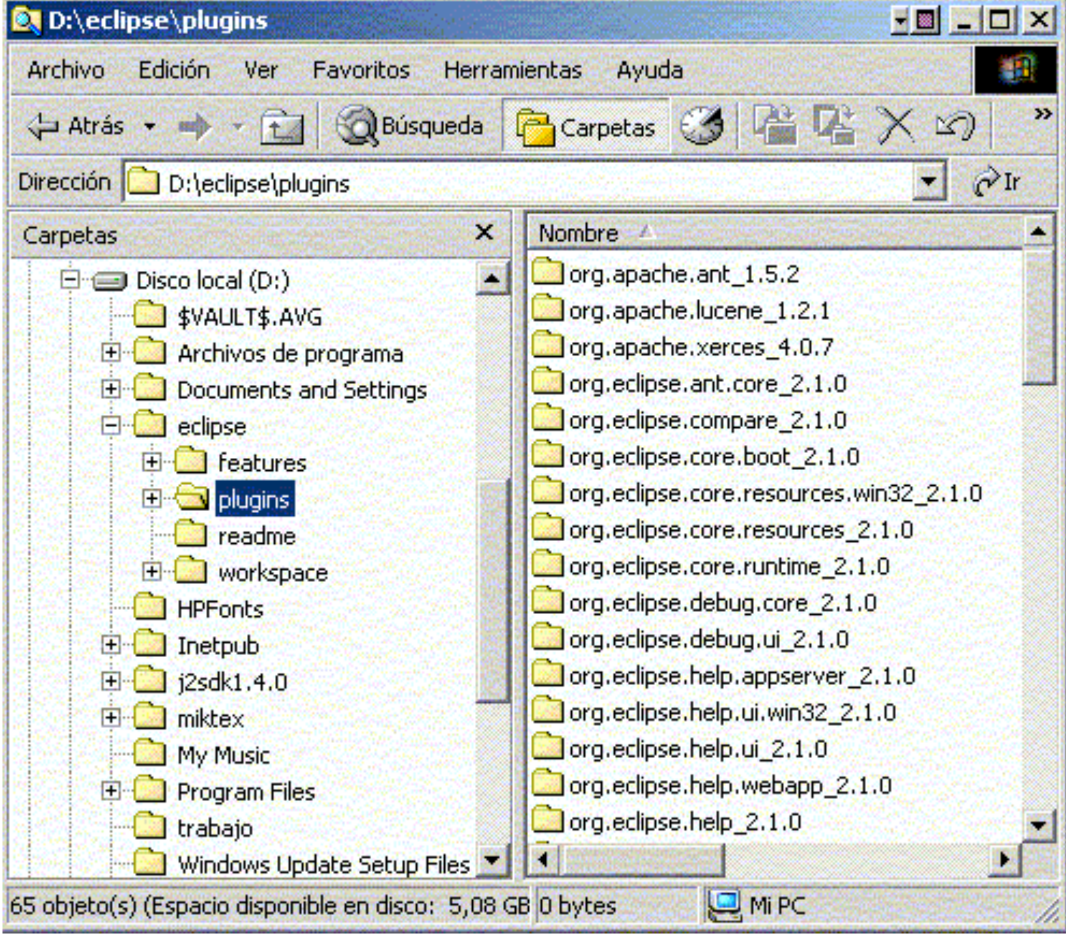


Figura 1.4.7.1 Plugins en Eclipse

Existen gran cantidad de plugins desarrollados. Por ejemplo el plugin **EclipseUML** de Omondo para realizar diseño UML (diagramas de clases, de paquetes, etc) en un determinado proyecto, o el plugin **Lomboz** para realizar proyectos J2EE.

Eclipse no es único entorno integrado para trabajar con Java. Otro entorno de código abierto igualmente recomendable es **NetBeans**, proporcionado por Sun.

1.5 Tipos de datos

La plataforma Java nos proporciona un amplio conjunto de clases dentro del que podemos encontrar tipos de datos que nos resultarán muy útiles para realizar la programación de aplicaciones en Java. Estos tipos de datos nos ayudarán a generar código más limpio de una forma sencilla.

Se proporcionan una serie de operadores para acceder a los elementos de estos tipos de datos. Decimos que dichos operadores son *polimórficos*, ya que un mismo operador se puede emplear para acceder a distintos tipos de datos. Por ejemplo, un operador *add* utilizado para añadir un elemento, podrá ser empleado tanto si estamos trabajando con una lista enlazada, con un array, o con un conjunto por ejemplo.

Este *polimorfismo* se debe a la definición de interfaces que deben implementar los distintos tipos de datos. Siempre que el tipo de datos contenga una colección de elementos, implementará la interfaz **Collection**. Esta interfaz proporciona métodos para acceder a la colección de elementos, que podremos utilizar para cualquier tipo de datos que sea una colección de elementos, independientemente de su implementación concreta.

Podemos encontrar los siguientes elementos dentro del marco de colecciones de Java:

- Interfaces para distintos tipos de datos: Definirán las operaciones que se pueden realizar con dichos tipos de datos. Podemos encontrar aquí la interfaz para cualquier colección de datos, y de manera más concreta para listas (secuencias) de datos, conjuntos, etc.
- Implementaciones de tipos de datos reutilizables: Son clases que implementan tipos de datos concretos que podremos utilizar para nuestras aplicaciones, implementando algunas de las interfaces anteriores para acceder a los elementos de dicho tipo de datos. Por ejemplo, dentro de las listas de elementos, podremos encontrar distintas implementaciones de la lista como puede ser listas enlazadas, o bien arrays de capacidad variable, pero al implementar la misma interfaz podremos acceder a sus elementos mediante las mismas operaciones (polimorfismo).
- Algoritmos para trabajar con dichos tipos de datos, que nos permitan realizar una ordenación de los elementos de una lista, o diversos tipos de búsqueda de un determinado elemento por ejemplo.

1.5.1 Enumeraciones e iteradores

Antes de ver los tipos de datos vamos a ver dos elementos utilizados comunmente en Java para acceder a colecciones de datos.

Las enumeraciones, definidas mediante la interfaz **Enumeration**, nos permiten consultar los elementos que contiene una colección de datos. Muchos métodos de clases Java que deben devolver múltiples valores, lo que hacen es devolvernos una enumeración que podremos consultar mediante los métodos que ofrece dicha interfaz.

La enumeración irá recorriendo secuencialmente los elementos de la colección. Para leer cada elemento de la enumeración deberemos llamar al método:

```
Object item = enum.nextElement();
```

Que nos proporcionará en cada momento el siguiente elemento de la enumeración a leer. Además necesitaremos saber si quedan elementos por leer, para ello tenemos el método:

```
enum.hasMoreElements()
```

Normalmente, el bucle para la lectura de una enumeración será el siguiente:

```
while (enum.hasMoreElements()) {
  Object item = enum.nextElement();
  // Hacer algo con el item leido
}
```

Vemos como en este bucle se van leyendo y procesando elementos de la enumeración uno a uno mientras queden elementos por leer en ella.

Otro elemento para acceder a los datos de una colección son los iteradores. La diferencia está en que los iteradores además de leer los datos nos permitirán eliminarlos de la colección. Los iteradores se definen mediante la interfaz **Iterator**, que proporciona de forma análoga a la enumeración el método:

```
Object item = iter.next();
```

Que nos devuelve el siguiente elemento a leer por el iterador, y para saber si quedan más elementos que leer tenemos el método:

```
iter.hasNext()
```

Además, podemos borrar el último elemento que hayamos leido. Para ello tendremos el método:

```
iter.remove();
```

Por ejemplo, podemos recorrer todos los elementos de una colección utilizando un iterador y eliminar aquellos que cumplan ciertas condiciones:

```
while (iter.hasNext())
{
    Object item = iter.next();
    if(condicion_borrado(item))
            iter.remove();
}
```

Las enumeraciones y los iteradores no son tipos de datos, sino elementos que nos servirán para acceder a los elementos dentro de los tipos de datos que veremos a continuación.

1.5.2 Colecciones

Las colecciones representan grupos de objetos, denominados elementos. Podemos encontrar diversos tipos de colecciones, según si sus elementos están ordenados, o si permitimos repetición de elementos o no.

Es el tipo más genérico en cuanto a que se refiere a cualquier tipo que contenga un grupo de elementos. Viene definido por la interfaz **Collection**, de la cual heredará cada subtipo específico. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

```
boolean add(Object o)
```

Añade un elemento (objeto) a la colección. Nos devuelve *true* si tras añadir el elemento la colección ha cambiado, es decir, el elemento se ha añadido correctamente, o *false* en caso contrario.

```
void clear()
```

Elimina todos los elementos de la colección.

```
boolean contains(Object o)
```

Indica si la colección contiene el elemento (objeto) indicado.

```
boolean isEmpty()
```

Indica si la colección está vacía (no tiene ningún elemento).

```
Iterator iterator()
```

Proporciona un iterador para acceder a los elementos de la colección.

```
boolean remove(Object o)
```

Elimina un determinado elemento (objeto) de la colección, devolviendo *true* si dicho elemento estaba contenido en la colección, y *false* en caso contrario.

```
int size()
```

Nos devuelve el número de elementos que contiene la colección.

```
Object [] toArray()
```

Nos devuelve la colección de elementos como un array de objetos. Si sabemos de antemano que los objetos de la colección son todos de un determinado tipo (como por ejemplo de tipo **String**) podremos obtenerlos en un array del tipo adecuado, en lugar de usar un array de objetos genéricos. En este caso NO podremos hacer una conversión cast descendente de array de objetos a array de un tipo más concreto, ya que el array se habrá instanciado simplemente como array de objetos:

```
String [] cadenas = (String []) coleccion.toArray();
                                   // Esto no se puede hacer!!!
```

Lo que si podemos hacer es instanciar nosotros un array del tipo adecuado y hacer una conversión cast ascendente (de tipo concreto a array de objetos), y utilizar el siguiente método:

```
String [] cadenas = new String[coleccion.size()];
coleccion.toArray(cadenas);                   // Esto si que
funcionará
```

Esta interfaz es muy genérica, y por lo tanto no hay ningún tipo de datos que la implemente directamente, sino que implementarán subtipos de ellas. A continuación veremos los subtipos más comunes.

1.5.2.1 Listas de elementos

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la interfaz **List**, que además de los métodos generales de las colecciones, nos ofrece los siguientes para trabajar con los índices:

```
void add(int indice, Object obj)
```

Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

```
Object get(int indice)
```

Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

```
int indexOf(Object obj)
```

Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

```
Object remove(int indice)
```

Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

```
Object set(int indice, Object obj)
```

Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

ArrayList

Implementa una lista de elementos mediante un array de tamaño variable. Conforme se añaden elementos el tamaño del array irá creciendo si es necesario. El array tendrá una capacidad inicial, y en el momento en el que se rebase dicha capacidad, se aumentará el tamaño del array.

Las operaciones de añadir un elemento al final del array (*add*), y de establecer u obtener el elemento en una determinada posición (*get*/*set*) tienen un coste temporal constante. Las inserciones y borrados tienen un coste lineal $O(n)$, donde n es el número de elementos del array.

Hemos de destacar que la implementación de **ArrayList** no está sincronizada, es decir, si múltiples hilos acceden a un mismo **ArrayList** concurrentemente podriamos tener problemas en la consistencia de los datos. Por lo tanto, deberemos tener en cuenta cuando usemos este tipo de datos que debemos controlar la concurrencia de acceso. También podemos hacer que sea sincronizado como veremos más adelante.

Vector

El **Vector** es una implementación similar al **ArrayList**, con la diferencia de que el **Vector** si que **está sincronizado**. Este es un caso especial, ya que la implementación básica del resto de tipos de datos no está sincronizada.

Esta clase existe desde las primeras versiones de Java, en las que no existía el marco de las colecciones descrito anteriormente. En las últimas versiones el **Vector** se ha acomodado a este marco implementando la interfaz **List**.

Sin embargo, si trabajamos con versiones previas de JDK, hemos de tener en cuenta que dicha interfaz no existía, y por lo tanto esta versión previa del vector no contará con los métodos definidos en ella. Los métodos propios del vector para acceder a su contenido, que han existido desde las primeras versiones, son los siguientes:

```
void addElement(Object obj)
```

Añade un elemento al final del vector.

```
Object elementAt(int indice)
```

Devuelve el elemento de la posición del vector indicada por el índice.

```
void insertElementAt(Object obj, int indice)
```

Inserta un elemento en la posición indicada.

```
boolean removeElement(Object obj)
```

Elimina el elemento indicado del vector, devolviendo *true* si dicho elemento estaba contenido en el vector, y *false* en caso contrario.

```
void removeElementAt(int indice)
```

Elimina el elemento de la posición indicada en el índice.

```
void setElementAt(Object obj, int indice)
```

Sobrescribe el elemento de la posición indicada con el objeto especificado.

```
int size()
```

Devuelve el número de elementos del vector.

Por lo tanto, si programamos para versiones antiguas de la máquina virtual Java, será recomendable utilizar estos métodos para asegurarnos de que nuestro programa funcione. Esto será importante en la programación de Applets, ya que la máquina virtual incluida en muchos navegadores corresponde a versiones antiguas.

Sobre el vector se construye el tipo pila (**Stack**), que apoyándose en el tipo vector ofrece métodos para trabajar con dicho vector como si se tratase de una pila, apilando y desapilando elementos (operaciones *push* y *pop* respectivamente). La clase **Stack** hereda de **Vector**, por lo que en realidad será un vector que ofrece métodos adicionales para trabajar con él como si fuese una pila.

LinkedList

En este caso se implementa la lista mediante una lista doblemente enlazada. Por lo tanto, el coste temporal de las operaciones será el de este tipo de listas. Cuando realicemos inserciones, borrados o lecturas en los extremos inicial o final de la lista el tiempo será constante, mientras que para cualquier operación en la que necesitemos localizar un determinado índice dentro de la lista deberemos recorrer la lista de inicio a fin, por lo que el coste será lineal con el tamaño de la lista *O(n)*, siendo *n* el tamaño de la lista.

Para aprovechar las ventajas que tenemos en el coste temporal al trabajar con los extremos de la lista, se proporcionan métodos propios para acceder a ellos en tiempo constante:

```
void addFirst(Object obj) / void addLast(Object obj)
```

Añade el objeto indicado al principio / final de la lista respectivamente.

```
Object getFirst() / Object getLast()
```

Obtiene el primer / último objeto de la lista respectivamente.

```
Object removeFirst() / Object removeLast()
```

Extrae el primer / último elemento de la lista respectivamente, devolviéndonos dicho objeto y eliminándolo de la lista.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una pila o de una cola. En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

1.5.2.2 Conjuntos

Los conjuntos son grupos de elementos en los que no encontramos ningún elemento repetido. Consideramos que un elemento está repetido si tenemos dos objetos *o1* y *o2* iguales, comparandolos mediante el operador *o1.equals(o2)*. De esta forma, si el objeto a insertar en el conjunto estuviese repetido, no nos dejará insertarlo. Recordemos que el método add devolvía un valor *booleano*, que servirá para este caso, devolviendonos *true* si el elemento a añadir no estaba en el conjunto y ha sido añadido, o *false* si el elemento ya se encontraba dentro del conjunto. Un conjunto podrá contener a lo sumo un elemento *null*.

Los conjuntos se definen en la interfaz **Set**, a partir de la cuál se construyen diferentes implementaciones:

HashSet

Los objetos se almacenan en una tabla de dispersión (*hash*). El coste de las operaciones básicas (inserción, borrado, búsqueda) se realizan en tiempo constante siempre que los elementos se hayan dispersado de forma adecuada. La iteración a través de sus elementos es más costosa, ya que necesitará recorrer todas las entradas de la tabla de dispersión, lo que hará que el coste esté en función tanto del número de elementos insertados en el conjunto como del número de entradas de la tabla. El orden de iteración puede diferir del orden en el que se insertaron los elementos.

LinkedHashSet

Es similar a la anterior pero la tabla de dispersión es doblemente enlazada. Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con la carga adicional que supone tener que gestionar los enlaces. Sin embargo habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados. En este caso, al haber enlaces entre los elementos, estos enlaces definirán el orden en el que se insertaron en el conjunto, por lo que el orden de iteración será el mismo orden en el que se insertaron.

TreeSet

Utiliza un árbol para el almacenamiento de los elementos. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico con el número de elementos que tenga el conjunto *O(log n)*.

1.5.2.3 Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no heredan de la interfaz **Collection**.

Los mapas se definen en la interfaz **Map**. Un mapa es un objeto que relaciona una clave (*key*) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. En versiones anteriores este mapeado entre claves y valores lo hacía la clase **Dictionary**, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Los métodos básicos para trabajar con estos elementos son los siguientes:

```
Object get(Object clave)
```

Nos devuelve el valor asociado a la clave indicada

```
Object put(Object clave, Object valor)
```

Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o *null* si la clave no estaba en la tabla todavía.

```
Object remove(Object clave)
```

Elimina una clave, devolviendonos el valor que tenía dicha clave.

```
Set keySet()
```

Nos devuelve el conjunto de claves registradas

```
int size()
```

Nos devuelve el número de parejas (clave,valor) registradas.

Encontramos distintas implementaciones de los mapas:

HashMap

Utiliza una tabla de dispersión para almacenar la información del mapa. Las operaciones básicas (*get* y *put*) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. Es coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa. No se garantiza que se respete el orden de las claves.

TreeMap

Utiliza un árbol rojo-negro para implementar el mapa. El coste de las operaciones básicas será logarítmico con el número de elementos del mapa *O(log n)*. En este caso los elementos se encontrarán ordenados por orden ascendente de clave.

Hashtable

Es una implementación similar a **HashMap**, pero con alguna diferencia. Mientras las anteriores implementaciones no están sincronizadas, esta si que lo está. Además en esta implementación, al contrario que las anteriores, no se permitirán claves nulas (*null*). Este objeto extiende la obsoleta clase **Dictionary**, ya que viene de versiones más antiguas de JDK. Ofrece otros métodos además de los anteriores, como por ejemplo el siguiente:

```
Enumeration keys()
```

Este método nos devolverá una enumeración de todas las claves registradas en la tabla.

1.5.2.4 Algoritmos

Como hemos comentado anteriormente, además de las interfaces y las implementaciones de los tipos de datos descritos en los apartados previos, el marco de colecciones nos ofrece una serie de algoritmos utiles cuando trabajamos con estos tipos de datos, especialmente para las listas.

Estos algoritmos los podemos encontrar implementados como métodos estáticos en la clase **Collections**. En ella encontramos métodos para la ordenación de listas (*sort*), para la búsqueda binaria de elementos dentro de una lista (*binarySearch*) y otras operaciones que nos serán de gran utilidad cuando trabajemos con colecciones de elementos.

1.5.2.5 Wrappers

A parte de los algoritmos comentados en el apartado anterior, la clase **Collections** aporta otros métodos para cambiar ciertas propiedades de las listas. Estos métodos nos proporcionan los denominados *wrappers* de los

distintos tipos de colecciones. Estos *wrappers* son objetos que 'envuelven' al objeto de nuestra colección, pudiendo de esta forma hacer que la colección esté sincronizada, o que la colección pase a ser de solo lectura.

Como dijimos anteriormente, todos los tipos de colecciones no están sincronizados, excepto el **Vector** que es un caso especial. Al no estar sincronizados, si múltiples hilos utilizan la colección concurrentemente, podrán estar ejecutándose simultáneamente varios métodos de una misma colección que realicen diferentes operaciones sobre ella. Esto puede provocar inconsistencias en los datos. A continuación veremos un posible ejemplo de inconsistencia que se podría producir:

1. Tenemos un **ArrayList** de nombre *letras* formada por los siguiente elementos: ["A", "B", "C", "D"]
2. Imaginemos que un hilo de baja prioridad desea eliminar el objeto "C". Para ello hará una llamada al método *letras.remove("C")*.
3. Dentro de este método primero deberá determinar cuál es el índice de dicho objeto dentro del array, para después pasar a eliminarlo.
4. Se encuentra el objeto "C" en el índice 2 del array (recordemos que se empieza a numerar desde 0).
5. El problema viene en este momento. Imaginemos que justo en este momento se le asigna el procesador a un hilo de mayor prioridad, que se encarga de eliminar el elemento "A" del array, quedándose el array de la siguiente forma: ["B", "C", "D"]
6. Ahora el hilo de mayor prioridad es sacado del procesador y nuestro hilo sigue ejecutándose desde el punto en el que se quedó.
7. Ahora nuestro hilo lo único que tiene que hacer es eliminar el elemento del índice que había determinado, que resulta ser ¡el índice 2!. Ahora el índice 2 está ocupado por el objeto "D", y por lo tanto será dicho objeto el que se elimine.

Podemos ver que haciendo una llamada a *letras.remove("C")*, al final se ha eliminado el objeto "D", lo cual produce una inconsistencia de los datos con las operaciones realizadas, debido al acceso concurrente.

Este problema lo evitaremos sincronizando la colección. Cuando una colección está sincronizada, hasta que no termine de realizarse una operación (inserciones, borrados, etc), no se podrá ejecutar otra, lo cual evitará estos problemas.

Podemos conseguir que las operaciones se ejecuten de forma sincronizada envolviendo nuestro objeto de la colección con un *wrapper*, que será un objeto que utilice internamente nuestra colección encargándose de realizar la sincronización cuando llamemos a sus métodos. Para obtener estos *wrappers* utilizaremos los siguientes métodos estáticos de **Collections**:

```
Collection synchronizedCollection(Collection c)
List synchronizedList(List l)
Set synchronizedSet(Set s)
Map synchronizedMap(Map m)
```

```
SortedSet synchronizedSortedSet(SortedSet ss)
SortedMap synchronizedSortedMap(SortedMap sm)
```

Como vemos tenemos un método para envolver cada tipo de datos. Nos devolverá un objeto con la misma interfaz, por lo que podremos trabajar con él de la misma forma, sin embargo la implementación interna estará sincronizada.

Podemos encontrar también una serie de *wrappers* para obtener versiones de sólo lectura de nuestras colecciones. Se obtienen con los siguientes métodos:

```
Collection unmodifiableCollection(Collection c)
List unmodifiableList(List l)
Set unmodifiableSet(Set s)
Map unmodifiableMap(Map m)
SortedSet unmodifiableSortedSet(SortedSet ss)
SortedMap unmodifiableSortedMap(SortedMap sm)
```

1.5.3 Wrappers de tipos básicos

Hemos visto que en Java cualquier tipo de datos es un objeto, excepto los tipos de datos básicos: *boolean, int, long, float, double, byte, short, char*.

Cuando trabajamos con colecciones de datos los elementos que contienen éstas son siempre objetos, por lo que en un principio no podríamos insertar elementos de estos tipos básicos. Para hacer esto posible tenemos una serie de objetos que se encargarán de envolver a estos tipos básicos, permitiéndonos tratarlos como objetos y por lo tanto insertarlos como elementos de colecciones. Estos objetos son los llamados wrappers, y las clases en las que se definen tienen nombre similares al del tipo básico que encapsulan, con la diferencia de que comienzan con mayúscula: **Boolean, Integer, Long, Float, Double, Byte, Short, Character**.

Estas clases, además de servirnos para encapsular estos datos básicos en forma de objetos, nos proporcionan una serie de métodos e información útiles para trabajar con estos datos. Nos proporcionarán métodos por ejemplo para convertir cadenas a datos numéricos de distintos tipos y viceversa, así como información acerca del valor mínimo y máximo que se puede representar con cada tipo numérico.

1.5.4 Clases útiles

En esta sección vamos a ver una serie de clases que conviene conocer ya que nos serán de gran utilidad para realizar nuestros programas:

Object

Esta es la clase base de todas las clases en Java, toda clase hereda en última instancia de la clase **Object**, por lo que los métodos que ofrece estarán disponibles en cualquier objeto Java, sea de la clase que sea.

En Java es importante distinguir claramente entre lo que es una variable, y lo que es un objeto. Las variables simplemente son referencias a objetos,

mientras que los objetos son las entidades instanciadas en memoria que podrán ser manipulados mediante las referencias que tenemos a ellos (mediante variable que apunten a ellos) dentro de nuestro programa. Cuando hacemos lo siguiente:

```
new MiClase()
```

Se está instanciando en memoria un nuevo objeto de clase MiClase y nos devuelve una referencia a dicho objeto. Nosotros deberemos guardarnos dicha referencia en alguna variable con el fin de poder acceder al objeto creado desde nuestro programa:

```
MiClase mc = new MiClase();
```

Es importante declarar la referencia del tipo adecuado (en este caso tipo MiClase) para manipular el objeto, ya que el tipo de la referencia será el que indicará al compilador las operaciones que podremos realizar con dicho objeto. El tipo de esta referencia podrá ser tanto el mismo tipo del objeto al que vayamos a apuntar, o bien el de cualquier clase de la que herede o interfaz que implemente nuestro objeto. Por ejemplo, si MiClase se define de la siguiente forma:

```
public class MiClase extends Thread implements List {
    ...
}
```

Podremos hacer referencia a ella de diferentes formas:

```
MiClase mc = new MiClase();
Thread t = new MiClase();
List l = new MiClase();
Object o = new MiClase();
```

Esto es así ya que al heredar tanto de **Thread** como de **Object**, sabemos que el objeto tendrá todo lo que tienen estas clases más lo que añada **MiClase**, por lo que podrá comportarse como cualquiera de las clases anteriores. Lo mismo ocurre al implementar una interfaz, al forzar a que se implementen sus métodos podremos hacer referencia al objeto mediante la interfaz ya que sabemos que va a contener todos esos métodos. Siempre vamos a poder hacer esta asignación 'ascendente' a clases o interfaces de las que deriva nuestro objeto.

Si hacemos referencia a un objeto **MiClase** mediante una referencia **Object** por ejemplo, sólo podremos acceder a los métodos de **Object**, aunque el objeto contenga métodos adicionales definidos en **MiClase**. Si conocemos que nuestro objeto es de tipo **MiClase**, y queremos poder utilizarlo como tal, podremos hacer una asignación 'descendente' aplicando una conversión cast al tipo concreto de objeto:

```
Object o = new MiClase();
...
MiClase mc = (MiClase) o;
```

Si resultase que nuestro objeto no es de la clase a la que hacemos cast, ni hereda de ella ni la implementa, esta llamada resultará en un **ClassCastException** indicando que no podemos hacer referencia a dicho objeto mediante esa interfaz debido a que el objeto no la cumple, y por lo tanto podrán no estar disponibles los métodos que se definen en ella.

Una vez hemos visto la diferencia entre las variables (referencias) y objetos (entidades) vamos a ver como se hará la asignación y comparación de objetos. Si hiciesemos lo siguiente:

```
MiClase mc1 = new MiClase();
MiClase mc2 = mc1;
```

Puesto que hemos dicho que las variables simplemente son referencias a objetos, la asignación estará copiando una referencia, no el objeto. Es decir, tanto la variable *mc1* como *mc2* apuntarán a un mismo objeto.

Si lo que queremos es copiar un objeto, teniendo dos entidades independientes, deberemos invocar el método **clone** del objeto a copiar:

```
MiClase mc2 = (MiClase)mc1.clone();
```

El método **clone** es un método de la clase **Object** que estará disponible para cualquier objeto Java, y nos devuelve un **Object** genérico, ya que al ser un método que puede servir para cualquier objeto nos debe devolver la copia de este tipo. De él tendremos que hacer una conversión cast a la clase de la que se trate como hemos visto en el ejemplo.

Por otro lado, para la comparación, si hacemos lo siguiente:

```
mc1 == mc2
```

Estaremos comparando referencias, por lo que estaremos viendo si las dos referencias apuntan a un mismo objeto, y no si los objetos a los que apuntan son iguales. Para ver si los objetos son iguales, aunque sean entidades distintas, tenemos:

```
mc1.equals(mc2)
```

Este método también es propio de la clase **Object**, y será el que se utilice para comparar internamente los objetos.

Tanto **clone** como **equals**, deberán ser redefinidos en nuestras clases para adaptarse a éstas. Deberemos especificar dentro de ellos como se copia nuestro objeto y como se compara si son iguales:

```
public class Punto2D {

    public int x, y;


    ...
```

```
    public boolean equals(Object o) {
            Punto2D p = (Punto2D)o;
            // Compara objeto this con objeto p
            return (x == p.x && y == p.y);
    }

    public Object clone() {
            Punto2D p = new Punto2D();
            // Construye nuevo objeto p
            // copiando los atributos de this
            p.x = x;
            p.y = y;
            return p;
    }
```

Un último método interesante de la clase **Object** es **toString**. Este método nos devuelve una cadena (**String**) que representa dicho objeto. Por defecto nos dará un identificador del objeto, pero nosotros podemos sobrescribirla en nuestras propias clases para que genere la cadena que queramos. De esta manera podremos imprimir el objeto en forma de cadena de texto, mostrandose los datos con el formato que nosotros les hayamos dado en **toString**. Por ejemplo, si tenemos una clase **Punto2D**, sería buena idea hacer que su conversión a cadena muestre las coordenadas *(x,y)* del punto:

```
public class Punto2D {

    public int x,y;


    ...


    public String toString() {
            String s = "(" + x + "," + y + ")";
            return s;
    }
}
```

Properties

Esta clase es un subtipo de **Hastable**, que se encarga de almacenar una serie de propiedades asociando un valor a cada una de ellas. Estas propiedades las podremos utilizar para registrar la configuración de nuestra aplicación. Además esta clase nos permite cargar o almacenar esta información en algún dispositivo, como puede ser en disco, de forma que sea persistente.

Puesto que hereda de **Hashtable**, podremos utilizar sus métodos, pero también aporta métodos propios para añadir propiedades:

```
Object setProperty(Object clave, Object valor)
```

Equivalente al método *put*.

```
Object getProperty(Object clave)
```

Equivalente al método *get*.

```
Object getProperty(Object clave, Object default)
```

Esta variante del método resulta útil cuando queremos que determinada propiedad devuelva algún valor por defecto si todavía no se le ha asignado ningún valor.

Además, como hemos dicho anteriormente, para hacer persistentes estas propiedades de nuestra aplicación, se proporcionan métodos para almacenarlas o leerlas de algún dispositivo de E/S:

```
void load(InputStream entrada)
```

Lee las propiedades del flujo de entrada proporcionado. Este flujo puede por ejemplo referirse a un fichero del que se leerán los datos.

```
void store(OutputStream salida, String cabecera)
```

Almacena las información de las propiedades escribiendolas en el flujo de salida especificado. Este flujo puede por ejemplo referirse a un fichero en disco, en el que se guardará nuestro conjunto de propiedades, pudiendo especificar una cadena que se pondrá como cabecera en el fichero, y que nos permite añadir algún comentario sobre dicho fichero.

System

Esta clase nos ofrece una serie de métodos y campos útiles del sistema. Esta clase no se debe instanciar, todos estos métodos y campos son estáticos.

Podemos encontrar los objetos que encapsulan la entrada, salida y salida de error estándar, así como métodos para redireccionarlas, que veremos con más detalle en el tema de entrada/salida.

También nos permite acceder al gestor de seguridad instalado, como veremos en el tema sobre seguridad.

Otros métodos útiles que encontramos son:

```
void exit(int estado)
```

Finaliza la ejecución de la aplicación, devolviendo un código de estado. Normalmente el código 0 significa que ha salido de forma normal, mientras que con otros códigos indicaremos que se ha producido algún error.

```
void gc()
```

Fuerza una llamada al colector de basura para limpiar la memoria. Esta es una operación costosa. Normalmente no lo llamaremos explicitamente, sino que dejaremos que Java lo invoque cuando sea necesario.

```
long currentTimeMillis()
```

Nos devuelve el tiempo medido en el número de milisegundos transcurridos desde el 1 de Enero de 1970 a las 0:00.

```
void arraycopy(Object fuente, int pos_fuente,
                           Object destino, int pos_dest, int n)
```

Copia n elementos del array fuente, desde la posición pos_fuente, al array destino a partir de la posición pos_dest.

Runtime

Toda aplicación Java tiene una instancia de la clase **Runtime** que se encargará de hacer de interfaz con el entorno en el que se está ejecutando. Para obtener este objeto debemos utilizar el siguiente método estático:

```
Runtime rt = Runtime.getRuntime();
```

Una de las operaciones que podremos realizar con este objeto, será ejecutar comandos como si nos encontrásemos en la línea de comandos del sistema operativo. Para ello utilizaremos el siguiente método:

```
rt.exec(comando);
```

De esta forma podremos invocar programas externos desde nuestra aplicación Java.

Math

La clase **Math** nos será de gran utilidad cuando necesitemos realizar operaciones matemáticas. Esta clase no necesita ser instanciada, ya que todos sus métodos son estáticos. Entre estos métodos podremos encontrar todas las operaciones matemáticas básicas que podamos necesitar, como logaritmos, exponenciales, funciones trigonométricas, generación de números aleatorios, conversión entre grados y radianes, etc. Además nos ofrece las constantes de los números *PI* y *E*.

Otras clases

Si miramos dentro del paquete **java.util**, podremos encontrar una serie de clases que nos podrán resultar útiles para determinadas aplicaciones.

Entre ellas tenemos la clase **Calendar**, que nos servirá cuando trabajemos con fechas y horas, para realizar operaciones con fechas, comparar fechas, u obtener distintas representaciones para mostrar la fecha en nuestra aplicación.

Encontramos también la clase **Currency** con información monetaria. La clase **Locale** almacena información sobre una determinada región del mundo, por lo que podremos utilizar esta clase junto a las anteriores para obtener la moneda

de una determinada zona, o las diferencias horarias y de representación de fechas.

1.6 Algunos consejos

Hemos visto que Java nos permite escribir facilmente un código limpio y mantenible. Sin embargo, en muchas ocasiones además nos interesará que el código sea rápido en determinadas funciones críticas. A continuación damos una serie de consejos para optimizar el código Java:

- No instanciar más objetos de los necesarios. Es una buena práctica para la eficiencia temporal del código reutilizar los objetos que tenemos ya instanciados siempre que sea posible, ya que consume tiempo tanto instanciar nuevos objetos, como después limpiar de la memoria los objetos que ya no se necesiten por parte del colector de basura.
- Minimizar el número de llamadas a métodos. La llamada a un método para obtener una determinada propiedad de un objeto es más costoso computacionalmente que consultar la propiedad directamente (en el caso de que sea pública). Si necesitamos utilizar el valor repetidas veces es buena idea leer el valor en una variable local y utilizar dicha variable.
- Es más rápido acceder a un campo de un objeto directamente que llamar a un método para obtener el valor de dicho campo. Acceder directamente a los campos va en contra de la encapsulación, pero puede resultar conveniente en determinados casos. Si desarrollamos una librería con una serie de clases, podemos usar variables protegidas en lugar de privadas, para dentro de nuestra librería no tener que llamar a métodos para consultar o modificar dicha información. Esto hará más rápidas las llamadas internas a la librería.
- Sustituir tipos de datos complejos por tipos de datos básicos. Esto va en contra de la legibilidad del código, pero en caso de ser la velocidad un factor crítico puede ser conveniente hacer este cambio. Una vez comprobado que el programa funciona, si necesitamos más velocidad podemos cambiar tipos de datos como Vectores por un array básico cuyo acceso resulta más rápido.
- Cuando trabajemos con cadenas grandes, es conveniente utilizar la clase **StringBuffer** en lugar de **String**, ya permite ser modificada sin necesidad de instanciar nuevos objetos, lo cual hará la manipulación de estas cadenas mucho más eficiente.

2. Características básicas

2.1. Excepciones

Las excepciones son eventos que ocurren durante la ejecución de un programa y hacen que éste salga de su flujo normal de instrucciones. Este mecanismo permite tratar los errores de una forma elegante, ya que separa el código para el tratamiento de errores del código normal del programa. Se dice que una excepción es *lanzada* cuando se produce un error, y esta excepción puede ser *capturada* para tratar dicho error.

2.1.1. Tipos de excepciones

Tenemos diferentes tipos de excepciones dependiendo del tipo de error que representen. Todas ellas descienden de la clase **Throwable**, la cual tiene dos descendientes directos:

- **Error**: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **Exception**: Representa errores que no son críticos y por lo tanto pueden ser tratados y continuar la ejecución de la aplicación. La mayoría de los programas Java utilizan estas excepciones para el tratamiento de los errores que puedan ocurrir durante la ejecución del código.

Dentro de **Exception**, cabe destacar una subclase especial de excepciones denominada **RuntimeException**, de la cual derivarán todas aquellas excepciones referidas a los errores que comúnmente se pueden producir dentro de cualquier fragmento de código, como por ejemplo hacer una referencia a un puntero *null*, o acceder fuera de los límites de un *array*.

Estas **RuntimeException** se diferencian del resto de excepciones en que no son de tipo *checked*. Una excepción de tipo *checked* debe ser capturada o bien especificar que puede ser lanzada de forma obligatoria, y si no lo hacemos obtendremos un error de compilación. Dado que las **RuntimeException** pueden producirse en cualquier fragmento de código, sería impensable tener que añadir manejadores de excepciones y declarar que éstas pueden ser lanzadas en todo nuestro código.

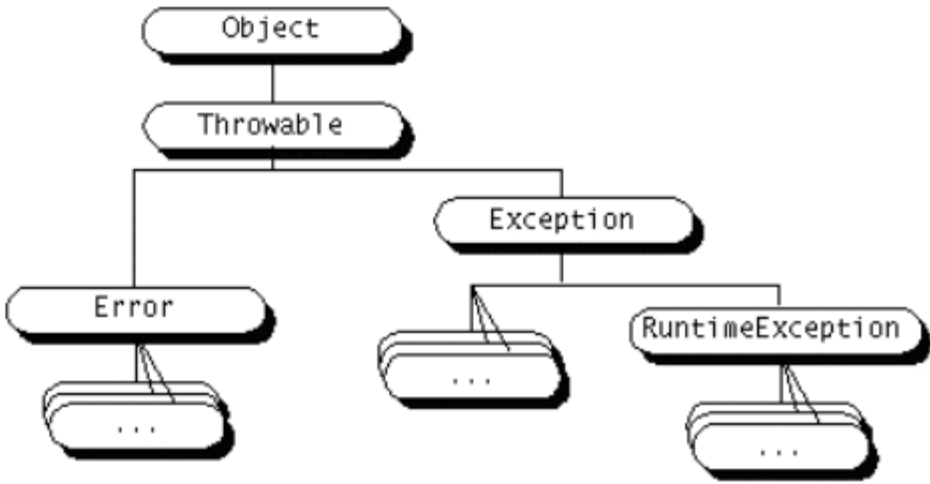


Figura 1. Tipos de excepciones

Dentro de estos grupos principales de excepciones podremos encontrar tipos concretos de excepciones o bien otros grupos que a su vez pueden contener más subgrupos de excepciones, hasta llegar a tipos concretos de ellas. Cada tipo de excepción guardará información relativa al tipo de error al que se refiera, además de la información común a todas las excepciones. Por ejemplo, una **ParseException** se suele utilizar al procesar un fichero. Además de almacenar un mensaje de error, guardará la línea en la que el *parser* encontró el error.

2.1.2. Captura de excepciones

Cuando un fragmento de código sea susceptible de lanzar una excepción y queramos tratar el error producido o bien por ser una excepción de tipo *checked* debamos capturarla, podremos hacerlo mediante la estructura *try-catch-finally*, que consta de tres bloques de código:

- Bloque *try*: Contiene el código regular de nuestro programa que puede producir una excepción en caso de error.
- Bloque *catch*: Contiene el código con el que trataremos el error en caso de producirse.
- Bloque *finally*: Este bloque contiene el código que se ejecutará al final tanto si se ha producido una excepción como si no lo ha hecho. Este bloque se utiliza para, por ejemplo, cerrar algún fichero que haya podido ser abierto dentro del código regular del programa, de manera que nos aseguremos que tanto si se ha producido un error como si no este fichero se cierre.

Notar que podemos poner un bloque *try* sin *catch* pero con *finally* (cuando por ejemplo no queremos capturar la excepción, pero sí que hay cosas que queremos hacer tanto si se produce como si no), o podemos poner un bloque *try* con *catch* y sin *finally* (cuando queremos capturar la excepción, y no hay ningún código que necesitemos que se ejecute inexorablemente).

Para el bloque *catch* además deberemos especificar el tipo o grupo de excepciones que tratamos en dicho bloque, pudiendo incluir varios bloques *catch*, cada uno de ellos para un tipo/grupo de excepciones distinto. La forma de hacer esto será la siguiente:

```
try {
        // Código regular del programa
        // Puede producir excepciones
} catch(TipoDeExcepcion1 e1) {
        // Código que trata las excepciones de tipo
        // TipoDeExcepcion1 o subclases de ella.
        // Los datos sobre la excepción los encontraremos
        // en el objeto e1.
} catch(TipoDeExcepcion2 e2) {
        // Código que trata las excepciones de tipo
        // TipoDeExcepcion2 o subclases de ella.
        // Los datos sobre la excepción los encontraremos
        // en el objeto e2.
...
} catch(TipoDeExcepcionN eN) {
        // Código que trata las excepciones de tipo
        // TipoDeExcepcionN o subclases de ella.
        // Los datos sobre la excepción los encontraremos
        // en el objeto eN.
} finally {
        // Código de finalización (opcional)
}
```

Si como tipo de excepción especificamos un grupo de excepciones este bloque se encargará de la captura de todos los subtipos de excepciones de este grupo. Por lo tanto, si especificamos **Exception** capturaremos cualquier excepción, ya que está es la superclase común de todas las excepciones.

En el bloque *catch* pueden ser útiles algunos métodos de la excepción (que podemos ver en la API de la clase padre *Exception*):

```
String getMessage()
void printStackTrace()
```

con **getMessage()** obtenemos una cadena descriptiva del error (si la hay). Con **printStackTrace()** se muestra por la salida estándar la traza de errores que se han producido (en ocasiones la traza es muy larga y no puede seguirse toda en pantalla con algunos sistemas operativos).

Un ejemplo de uso:

```
try
{
        ... // Aqui va el codigo que puede lanzar una excepcion
} catch (Exception e) {
        System.out.println ("El error es: " + e.getMessage());
        e.printStackTrace();
}
```

2.1.3. Lanzamiento de excepciones

Hemos visto cómo capturar excepciones que se produzcan en el código, pero en lugar de capturarlas también podemos hacer que se propaguen al método de nivel superior (desde el cual se ha llamado al método actual). Para esto, en el método donde se vaya a lanzar la excepción, se siguen 2 pasos:

- Indicar en el método que determinados tipos de excepciones o grupos de ellas pueden ser lanzados, cosa que haremos de la siguiente forma, por ejemplo:

```
public void lee_fichero()
throws IOException, FileNotFoundException
{
    // Cuerpo de la función
}
```

Podremos indicar tantos tipos de excepciones como queramos en la claúsula ***throws***. Si alguna de estas clases de excepciones tiene subclases, también se considerará que puede lanzar todas estas subclases.

- Para lanzar la excepción utilizamos la instrucción ***throw***, proporcionándole un objeto correspondiente al tipo de excepción que deseamos lanzar. Por ejemplo:

```
throw new IOException(mensaje_error);
```

- Juntando estos dos pasos:

```
public void lee_fichero()
throws IOException, FileNotFoundException
{
    ...
    throw new IOException(mensaje_error);
    ...
}
```

Podremos lanzar así excepciones en nuestras funciones para indicar que algo no es como debiera ser a las funciones llamadoras. Por ejemplo, si estamos procesando un fichero que debe tener un determinado formato, sería buena idea lanzar excepciones de tipo **ParseException** en caso de que la sintaxis del fichero de entrada no sea correcta.

```
public void leeFich()
throws ParseException
{
    ...
    throw new ParseException("Error al procesar el fichero");
    ...
}

...

public void otroMetodo()
{
    try
    {
```

```
        leeFich();
    } catch (ParseException e) {
        System.out.println ("Se ha producido error al leer fichero");
        System.out.println ("El mensaje es: " + e.getMessage());
    }
}
```

NOTA: para las excepciones que no son de tipo *checked* no hará falta la cláusula *throws* en la declaración del método, pero seguirán el mismo comportamiento que el resto, si no son capturadas pasarán al método de nivel superior, y seguirán así hasta llegar a la función principal, momento en el que si no se captura provocará el error correspondiente.

2.1.4. Creación de nuevas excepciones

Además de utilizar los tipos de excepciones contenidos en la distribución de Java, podremos crear nuevos tipos que se adapten a nuestros problemas.

Para crear un nuevo tipo de excepciones simplemente deberemos crear una clase que herede de **Exception** o cualquier otro subgrupo de excepciones existente. En esta clase podremos añadir métodos y propiedades para almacenar información relativa a nuestro tipo de error. Por ejemplo:

```
public class MiExcepcion extends Exception
{
        public MiExcepcion (String mensaje)
        {
                super(mensaje);
        }
}
```

Además podremos crear subclases de nuestro nuevo tipo de excepción, creando de esta forma grupos de excepciones. Para utilizar estas excepciones (capturarlas y/o lanzarlas) hacemos lo mismo que lo explicado antes para las excepciones que se tienen definidas en Java. Por ejemplo:

```
public void unMetodo()
throws MiException
{
    ...
    throw new MiException("Error en el metodo");
    ...
}

...

public void otroMetodo()
{
    try
    {
        unMetodo();
    } catch (MiException e) {
        ...
    }
}
```

2.2. Hilos

Un hilo es un flujo de control dentro de un programa que permite realizar una tarea separada. Es decir, creando varios hilos podremos realizar varias tareas simultáneamente. Cada hilo tendrá sólo un contexto de ejecución (contador de programa, pila de ejecución). Es decir, a diferencia de los procesos UNIX, no tienen su propio espacio de memoria sino que acceden todos al mismo espacio de memoria común, por lo que será importante su sincronización cuando tengamos varios hilos accediendo a los mismos objetos.

2.2.1. Creación de hilos

En Java los hilos están encapsulados en la clase **Thread**. Para crear un hilo tenemos dos posibilidades:

- Heredar de **Thread** redefiniendo el método *run()*.
- Crear una clase que implemente la interfaz **Runnable** que nos obliga a definir el método *run()*.

En ambos casos debemos definir un método *run()* que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método *run()* será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método *run()*.

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
{
    public void run() {
        // Código del hilo
    }
}
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

```
Thread t = new EjemploHilo();
t.start();
```

Al llamar al método *start* del hilo, comenzará ejecutarse su método *run*. Crear un hilo heredando de **Thread** tiene el problema de que al no haber herencia múltiple en Java, si heredamos de **Thread** no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz **Runnable** para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación:

```
public class EjemploHilo implements Runnable
{
    public void run() {
        // Código del hilo
    }
}
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

```
Thread t = new Thread(new EjemploHilo());
t.start();
```

Esto es así debido a que en este caso **EjemploHilo** no deriva de una clase **Thread**, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método *run()*. Con esto lo que haremos será proporcionar esta clase al constructor de la clase **Thread**, para que el objeto **Thread** que creemos llame al método *run()* de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

2.2.2. Estado y propiedades de los hilos

Un hilo pasará por varios estados durante su ciclo de vida.

```
Thread t = new Thread(this);
```

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de *Nuevo hilo*.

```
t.start();
```

Cuando invoquemos su método *start()* el hilo pasará a ser un hilo *vivo*, comenzándose a ejecutar su método *run()*. Una vez haya salido de este método pasará a ser un hilo *muerto*.

La única forma de parar un hilo es hacer que salga del método *run()* de forma natural. Podremos conseguir esto haciendo que se cumpla una condición de salida de *run()* (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos haciendo). Las funciones para parar, pausar y reanudar hilos están desaprobadas en las versiones actuales de Java.

Mientras el hilo esté *vivo*, podrá encontrarse en dos estados: *Ejecutable* y *No ejecutable*. El hilo pasará de *Ejecutable* a *No ejecutable* en los siguientes casos:

- Cuando se encuentre dormido por haberse llamado al método *sleep()*, permanecerá *No ejecutable* hasta haber transcurrido el número de milisegundos especificados.
- Cuando se encuentre bloqueado en una llamada al método *wait()* esperando que otro hilo lo desbloquee llamando a *notify()* o *notifyAll()*. Veremos cómo utilizar estos métodos más adelante.

- Cuando se encuentre bloqueado en una petición de E/S, hasta que se complete la operación de E/S.

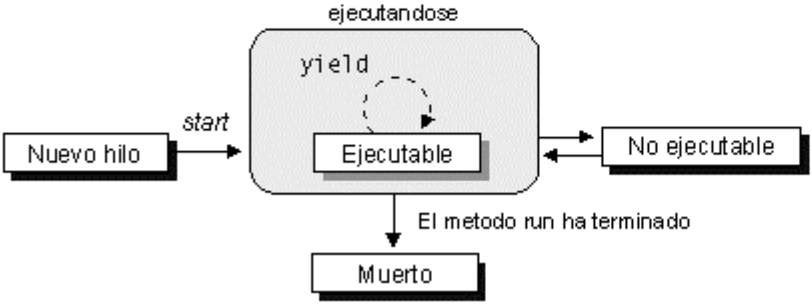


Figura 2. Ciclo de vida de los hilos

Lo único que podremos saber es si un hilo se encuentra vivo o no, llamando a su método *isAlive()*.

Prioridades de los hilos

Además, una propiedad importante de los hilos será su prioridad. Mientras el hilo se encuentre vivo, el *scheduler* de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede forzar la salida de un hilo de la CPU llamando a su método *yield()*. También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga *Ejecutable*, o cuando el tiempo que se le haya asignado expire.

Para cambiar la prioridad de un hilo se utiliza el método *setPriority()*, al que deberemos proporcionar un valor de prioridad entre *MIN_PRIORITY* y *MAX_PRIORITY* (tenéis constantes de prioridad disponibles dentro de la clase *Thread*, consultad el API de Java para ver qué valores de constantes hay).

Hilo actual

En cualquier parte de nuestro código Java podemos llamar al método *currentThread* de la clase *Thread*, que nos devuelve un objeto hilo con el hilo que se encuentra actualmente ejecutando el código donde está introducido ese método. Por ejemplo, si tenemos un código como:

```
public class EjemploHilo implements Runnable
{
    public EjemploHilo()
    {
        ...
        int i = 0;
        Thread t = Thread.currentThread();
               t.sleep(1000);
    }
}
```

La llamada a *currentThread* dentro del constructor de la clase nos devolverá el hilo que corresponde con el programa principal (puesto que no hemos creado

ningún otro hilo, y si lo creáramos, no ejecutaría nada que no estuviese dentro de un método *run.*

Sin embargo, en este otro caso:

```
public class EjemploHilo implements Runnable
{
    public EjemploHilo()
    {
        Thread t1 = new Thread(this);
        Thread t2 = new Thread(this);
        t1.start();
        t2.start();
    }

    public void run()
    {
        int i = 0;
        Thread t = Thread.currentThread();
               t.sleep(1000);
    }
}
```

Lo que hacemos es crear dos hilos auxiliares, y la llamada a *currentThread* se produce dentro del *run*, con lo que se aplica a los hilos auxiliares, que son los que ejecutan el *run:* primero devolverá un hilo auxiliar (el que primero entre, t1 o t2), y luego el otro (t2 o t1).

Dormir hilos

Como hemos visto en los ejemplos anteriores, una vez obtenemos el hilo que queremos, el método *sleep* nos sirve para dormirlo, durante los milisegundos que le pasemos como parámetro (en los casos anteriores, dormían durante 1 segundo). El tiempo que duerme el hilo, deja libre el procesador para que lo ocupen otros hilos. Es una forma de no sobrecargar mucho de trabajo a la CPU con muchos hilos intentando entrar sin descanso.

2.2.3. Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de comunicación es la variable cerrojo incluida en todo objeto **Object**, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los métodos declarados como *synchronized* utilizan el cerrojo del objeto al que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void seccion_critica()
{
    // Código sección crítica
}
```

Todos los métodos *synchronized* de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
synchronized (objeto_con_cerrojo)
{
    // Código sección crítica
}
```

de esta forma sincronizaríamos el código que escribiésemos dentro, con el código *synchronized* del objeto *objeto_con_cerrojo*.

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función *wait()*, para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método *synchronized*, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará *notifyAll()*, o bien *notify()* para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método *join()* de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

2.2.4. Grupos de hilos

Los grupos de hilos nos permitirán crear una serie de hilos y manejarlos todos a la vez como un único objeto. Si al crear un hilo no se especifica ningún grupo de hilos, el hilo creado pertenecerá al grupo de hilos por defecto.

Podemos crearnos nuestro propio grupo de hilos instanciando un objeto de la clase **ThreadGroup**. Para crear hilos dentro de este grupo deberemos pasar este grupo al constructor de los hilos que creemos.

```
ThreadGroup grupo = new ThreadGroup("Grupo de hilos");
Thread t = new Thread(grupo,new EjemploHilo());
```

2.3. Entrada/salida

Los programas muy a menudo necesitan enviar datos a un determinado destino, o bien leerlos de una determinada fuente externa, como por ejemplo puede ser un fichero para almacenar datos de forma permanente, o bien enviar datos a través de la red, a memoria, o a otros programas. Esta entrada/salida de datos en Java la realizaremos por medio de *flujos (streams)* de datos, a través de los cuales un programa podrá recibir o enviar datos en serie.

2.3.1 Flujos de datos de entrada/salida

Existen varios objetos que hacen de flujos de datos, y que se distinguen por la finalidad del flujo de datos y por el tipo de datos que viajen a través de ellos. Según el tipo de datos que transporten podemos distinguir:

- Flujos de caracteres
- Flujos de bytes

Dentro de cada uno de estos grupos tenemos varios pares de objetos, de los cuales uno nos servirá para leer del flujo y el otro para escribir en él. Cada par de objetos será utilizado para comunicarse con distintos elementos (memoria, ficheros, red u otros programas). Estas clases, según sean de entrada o salida y según sean de caracteres o de bytes llevarán distintos sufijos, según se muestra en la siguiente tabla:

	Flujo de entrada / lector	**Flujo de salida / escritor**
Caracteres	XXXXReader	XXXXWriter
Bytes	XXXXInputStream	XXXXOutputStream

Donde XXXX se referirá a la fuente o sumidero de los datos. Puede tomar valores como los que se muestran a continuación:

File	Acceso a ficheros
Piped	Comunicación entre programas mediante tuberías (pipes)
String	Acceso a una cadena en memoria (solo caracteres)
CharArray	Acceso a un array de caracteres en memoria (solo caracteres)
ByteArray	Acceso a un array de bytes en memoria (solo bytes)

Además podemos distinguir los flujos de datos según su propósito, pudiendo ser:

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún procesamiento con ellos. Tenemos por ejemplo flujos que realizan un filtrado de los datos que viajan a través de ellos (con prefijo Filter),

conversores datos (con prefijo Data), bufferes de datos (con prefijo Buffered), preparados para la impresión de elementos (con prefijo Print), etc.

Un tipo de filtros de procesamiento a destacar son aquellos que nos permiten convertir un flujo de bytes a flujo de caracteres. Estos objetos son **InputStreamReader** y **OutputStreamWriter**. Como podemos ver en su sufijo, son flujos de caracteres, pero se construyen a partir de flujos de bytes, permitiendo de esta manera acceder a nuestro flujo de bytes como si fuese un flujo de caracteres.

Para cada uno de los tipos básicos de flujo que hemos visto existe una superclase, de la que heredaran todos sus subtipos, y que contienen una serie de métodos que serán comunes a todos ellos. Entre estos métodos encontramos los métodos básicos para leer o escribir caracteres o bytes en el flujo a bajo nivel. En la siguiente tabla se muestran los métodos más importantes de cada objeto:

InputStream	**read**(), **reset**(), **available**(), **close**()
OutputStream	**write**(int b), **flush**(), **close**()
Reader	**read**(), **reset**(), **close**()
Writer	**write**(int c), **flush**(), **close**()

Aparte de estos métodos podemos encontrar variantes de los métodos de lectura y escritura, otros métodos, y además cada tipo específico de flujo contendrá sus propios métodos. Todas estas clases se encuentran en el paquete **java.io**. Para más detalles sobre ellas se puede consultar la especificación de la API de Java.

2.3.2. Entrada, salida y salida de error estándar

Al igual que en C, en Java también existen los conceptos de entrada, salida, y salida de error estándar. La entrada estándar normalmente se refiere a lo que el usuario escribe en la consola, aunque el sistema operativo puede hacer que se tome de otra fuente. De la misma forma la salida y la salida de error estándar lo que hacen normalmente es mostrar los mensajes y los errores del programa respectivamente en la consola, aunque el sistema operativo también podrá redirigirlas a otro destino.

En Java esta entrada, salida y salida de error estándar se tratan de la misma forma que cualquier otro flujo de datos, estando estos tres elementos encapsulados en tres objetos de flujo de datos que se encuentran como propiedades estáticas de la clase **System**:

	Tipo	**Objeto**
Entrada estándar	InputStream	System.**in**
Salida estándar	PrintStream	System.**out**

Salida de error estándar	PrintStream	System.**err**

Para la entrada estándar vemos que se utiliza un objeto **InputStream** básico, sin embargo para la salida se utilizan objetos **PrintWriter** que facilitan la impresión de texto ofreciendo a parte del método común de bajo nivel *write(int b)* para escribir bytes, dos métodos más: *print(s)* y *println(s)*. Estas funciones nos permitirán escribir cualquier cadena, tipo básico, o bien cualquier objeto que defina el método *toString()* que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

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

En el caso de la impresión de errores por la salida de error de estándar, deberemos utilizar:

```
System.err.println("Error: Se ha producido un error");
```

Además la clase **System** nos permite sustituir estos flujos por defecto por otros flujos, cambiando de esta forma la entrada, salida y salida de error estándar.

2.3.3. Acceso a ficheros

Podremos acceder a ficheros bien por caracteres, o bien de forma binaria (por bytes). Las clases que utilizaremos en cada caso son:

	Lectura	**Escritura**
Caracteres	FileReader	FileWriter
Binarios	FileInputStream	FileOutputStream

Para crear un lector o escritor de ficheros deberemos proporcionar al constructor el fichero del que queremos leer o en el que queramos escribir. Podremos proporcionar esta información bien como una cadena de texto con el nombre del fichero, o bien construyendo un objeto **File** representando al fichero al que queremos acceder. Este objeto nos permitirá obtener información adicional sobre el fichero, a parte de permitirnos realizar operaciones sobre el sistema de ficheros.

A continuación vemos un ejemplo simple de la copia de un fichero carácter a carácter:

```
public void copia_fichero() {
    int c;
    try {
        FileReader in = new FileReader("fuente.txt");
        FileWriter out = new FileWriter("destino.txt");
```

```
        while( (c = in.read()) != -1)
        {
            out.write(c);
        }
        in.close();
        out.close();
    } catch(FileNotFoundException e1) {
        System.err.println("Error: No se encuentra el fichero");
    } catch(IOException e2) {
        System.err.println("Error leyendo/escribiendo fichero");
    }
}
```

En el ejemplo podemos ver que para el acceso a un fichero es necesario capturar dos excepciones, para el caso de que no exista el fichero al que queramos acceder y por si se produce un error en la E/S.

Para la escritura podemos utilizar el método anterior, aunque muchas veces nos resultará mucho más cómodo utilizar un objeto **PrintWriter** con el que podamos escribir directamente líneas de texto:

```
public void escribe_fichero() {
    FileWriter out = null;
    PrintWriter p_out = null;
    try {
        out = new FileWriter("result.txt");
        p_out = new PrintWriter(out);
        p_out.println("Este texto será escrito en el fichero salida");
    } catch(IOException e) {
        System.err.println("Error al escribir en el fichero");
    } finally {
        p_out.close();
    }
}
```

Observad también el uso del bloque *finally*, para cerrar el fichero tanto si se produce un error al escribir en él como si no.

Un caso particular: ficheros de propiedades

La clase **java.util.Properties** permite manejar de forma muy sencilla lo que se conoce como *ficheros de propiedades*. Dichos ficheros permiten almacenar una serie de pares *nombre=valor*, de forma que tendría una apariencia como esta:

```
#Comentarios
elemento1=valor1
elemento2=valor2
...
elementoN=valorN
```

Para leer un fichero de este tipo, basta con crear un objeto *Properties*, y llamar a su método *load()*, pasándole como parámetro el fichero que queremos leer, en forma de flujo de entrada (*InputStream*):

```
Properties p = new Properties();
```

```
p.load(new FileInputStream("datos.txt");
```

Una vez leído, podemos acceder a todos los elementos del fichero desde el objeto *Properties* cargado. Tenemos los métodos *getProperty* y *setProperty* para acceder a y modificar valores:

```
String valorElem1 = p.getProperty("elemento1");
p.setProperty("elemento2", "otrovalor");
```

También podemos obtener todos los nombres de elementos que hay, y recorrerlos, mediante el método *propertyNames()*, que nos devuelve una *Enumeration* para ir recorriendo:

```
Enumeration en = p.propertyNames();
while (en.hasMoreElements())
{
        String nombre = (String)(en.nextElement());
        String valor = p.getProperty(nombre);
}
```

Una vez hayamos leído o modificado lo que quisiéramos, podemos volver a guardar el fichero de propiedades, con el método *store* de *Properties*, al que se le pasa un flujo de salida (*OutputStream*) y una cabecera para el fichero:

```
p.store(new FileOutputStream("datos.txt"), "Fichero de propiedades");
```

2.3.4. Lectura de tokens

Hemos visto como leer un fichero carácter a carácter, pero en el caso de ficheros con una gramática medianamente compleja, esta lectura a bajo nivel hará muy difícil el análisis de este fichero de entrada. Necesitaremos leer del fichero elementos de la gramática utilizada, los llamados **tokens**, como pueden ser palabras, número y otros símbolos.

La clase **StreamTokenizer** se encarga de partir la entrada en **tokens** y nos permitirá realizar la lectura del fichero directamente como una secuencia de **tokens**. Esta clase tiene una serie de constantes identificando los tipos de **tokens** que puede leer:

StreamTokenizer.**TT_WORD**	Palabra
StreamTokenizer.**TT_NUMBER**	Número real o entero
StreamTokenizer.**TT_EOL**	Fin de línea
StreamTokenizer.**TT_EOF**	Fin de fichero
Carácter de comillas establecido	Cadena de texto encerrada entre comillas
Símbolos	Vendrán representados por el código del carácter ASCII del símbolo

Dado que un **StreamTokenizer** se utiliza para analizar un fichero de texto, siempre habrá que crearlo a partir de un objeto **Reader** (o derivados).

```
StreamTokenizer st = new StreamTokenizer(reader);
```

El método **nextToken()** leerá el siguiente token que encuentre en el fichero y nos devolverá el tipo de **token** del que se trata. Según este tipo podremos consultar las propiedades **sval** o **nval** para ver qué cadena o número respectivamente se ha leído del fichero. Tanto cuando se lea un **token** de tipo **TT_WORD** como de tipo cadena de texto entre comillas el valor de este **token** estará almacenado en **sval**. En caso de la lectura sea un número, su valor se almacenará en **nval** que es de tipo **double**. Como los demás símbolos ya devuelven el código del símbolo como tipo de **token** no será necesario acceder a su valor por separado. Podremos consultar el tipo del último **token** leído en la propiedad **ttype**.

Un bucle de procesamiento básico será el siguiente:

```
while(st.nextToken() != StreamTokenizer.TT_EOF) {
    switch(st.ttype) {
        case StreamTokenizer.TT_WORD:
            System.out.println("Leida cadena: " + st.sval);
            break;
        case StreamTokenizer.TT_NUMBER:
            System.out.println("Leido numero: " + st.nval);
            break;
    }
}
```

Podemos distinguir tres tipos de caracteres:

Ordinarios (**ordinaryChars**)	Caracteres que forman parte de los *tokens*.
De palabra (**wordChars**)	Una secuencia formada enteramente por este tipo de caracteres se considerará una palabra.
De espacio en blanco (**whitespaceChars**)	Estos caracteres no son interpretados como *tokens*, simplemente se utilizan para separar *tokens*. Normalmente estos caracteres son el espacio, tabulador, y salto de línea.

Para establecer qué caracteres pertenecerán a cada uno de estos tipos utilizaremos los métodos *ordinaryChars*, *wordChars* y *whitespaceChars* del objeto **StreamTokenizer** respectivamente. A cada uno de estos métodos le pasamos un rango de caracteres (según su código ASCII), que serán establecidos al tipo correspondiente al método que hayamos llamado. Por ejemplo, si queremos que una palabra sea una secuencia de cualquier carácter imprimible (con códigos ASCII desde 32 a 127) haremos lo siguiente:

```
st.wordChars(32,127);
```

Los caracteres pueden ser especificados tanto por su código ASCII numérico como especificando ese carácter entre comillas simples. Si ahora queremos hacer que las palabras sean separadas por el caracter ':' (dos puntos) hacemos la siguiente llamada:

```
st.whitespaceChars(':', ':');
```

De esta forma, si hemos hecho las llamadas anteriores el *tokenizer* leerá palabras formadas por cualquier carácter imprimible separadas por los dos puntos ':'. Al querer cambiar un único carácter, como siempre deberemos especificar un rango, deberemos especificar un rango formado por ese único carácter como inicial y final del rango. Si además quisieramos utilizar el guión '-' para separar palabras, no siendo caracteres consecutivos guión y dos puntos en la tabla ASCII, tendremos que hacer una tercera llamada:

```
st.whitespaceChars('-', '-');
```

Así tendremos tanto el guión como los dos puntos como separadores, y el resto de caracteres imprimibles serán caracteres de palabra. Podemos ver que el **StreamTokenizer** internamente implementa una tabla, en la que asocia a cada carácter uno de los tres tipos mencionados. Al llamar a cada uno de los tres métodos cambiará el tipo de todo el rango especificado al tipo correspondiente al método. Por ello es importante el orden en el que invoquemos este método. Si en el ejemplo en el que hemos hecho estas tres llamadas las hubiésemos hecho en orden inverso, al establecer todo el rango de caracteres imprimibles como *wordChars* hubiésemos sobrescrito el resultado de las otras dos llamadas y por lo tanto el guión y los dos puntos no se considerarían separadores.

Podremos personalizar el *tokenizer* indicando para cada carácter a que tipo pertenece. Además de con los tipos anteriores, podemos especificar el carácter que se utilice para encerrar las cadenas de texto (**quoteChar**), mediante el método *quoteChar*, y el carácter para los comentarios (**commentChar**), mediante *commentChar*. Esto nos permitirá definir comentarios de una línea que comiencen por un determinado carácter, como por ejemplo los comentarios estilo Pascal comenzados por el carácter almohadilla ('#'). Además tendremos otros métodos para activar comentarios tipo C como los comentarios *barra-barra* (//) y *barra-estrella* (/* */).

2.3.5. Acceso a ficheros o recursos dentro de un JAR

Hemos visto como leer y escribir ficheros, pero cuando ejecutamos una aplicación contenida en un fichero JAR, puede que necesitemos leer recursos contenidos dentro de este JAR.

Para acceder a estos recursos deberemos abrir un flujo de entrada que se encargue de leer su contenido. Para ello utilizaremos el método `getResourceAsStream` de la clase `Class`:

```
InputStream in = getClass().getResourceAsStream("/datos.txt");
```

De esta forma podremos utilizar el flujo de entrada obtenido para leer el contenido del fichero que hayamos indicado. Este fichero deberá estar contenido en el JAR de la aplicación.

Especificamos el carácter '`/`' delante del nombre del recurso para referenciarlo de forma relativa al directorio raíz del JAR. Si no lo especificásemos de esta forma se buscaría de forma relativa al directorio correspondiente al paquete de la clase actual.

2.3.6. Codificación de datos

Si queremos guardar datos en un fichero binario deberemos codificar estos datos en forma de *array* de *bytes*. Los flujos de procesamiento `DataInputStream` y `DataOutputStream` nos permitirán codificar y descodificar respectivamente los tipos de datos simples en forma de *array* de *bytes* para ser enviados a través de un flujo de datos.

Por ejemplo, podemos codificar datos en un *array* en memoria (`ByteArrayOutputStream`) de la siguiente forma:

```
String nombre = "Jose";
String edad = 25;

ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

dos.writeUTF(nombre);
dos.writeInt(edad);

dos.close();
baos.close();

byte [] datos = baos.toByteArray();
```

Podremos descodificar este *array* de *bytes* realizando el procedimiento inverso, con un flujo que lea un *array* de *bytes* de memoria (`ByteArrayInputStream`):

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);

String nombre = dis.readUTF();
int edad = dis.readInt();
```

Si en lugar de almacenar estos datos codificados en una *array* en memoria queremos guardarlos codificados en un fichero, haremos lo mismo simplemente sustituyendo el flujo canal de datos `ByteArrayOutputStream` por un `FileOutputStream`. De esta forma podremos utilizar cualquier canal de datos para enviar estos datos codificados a través de él.

2.3.7. Serialización de objetos

Si queremos enviar un objeto complejo a través de un flujo de datos, deberemos convertirlo en una serie de bytes. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos.

Para leer o escribir objetos podemos utilizar los objetos **ObjectInputStream** y **ObjectOutputStream** que incorporan los métodos **readObject()** y

writeObject(Object obj) respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de ser *serializables*.

Serán *serializables* aquellos objetos que implementan la interfaz **Serializable**. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

3. Interfaz Gráfica

3.1. AWT

3.1.1. Introducción a AWT

AWT (*Abstract Windows Toolkit*) es la parte de Java que se emplea para construir **interfaces gráficas** de usuario. Este paquete ha estado presente desde la primera versión (la 1.0), aunque con la 1.1 sufrió un cambio notable. En la versión 1.2 se incorporó también a Java una librería adicional, **Swing**, que enriquece a AWT en la construcción de aplicaciones gráficas.

Controles de AWT

Java proporciona una serie de **controles** que podremos colocar en las aplicaciones visuales que implementemos. Dichos controles son subclases de la clase **Component**, y forman parte del paquete **java.awt**. Las más comunes son:

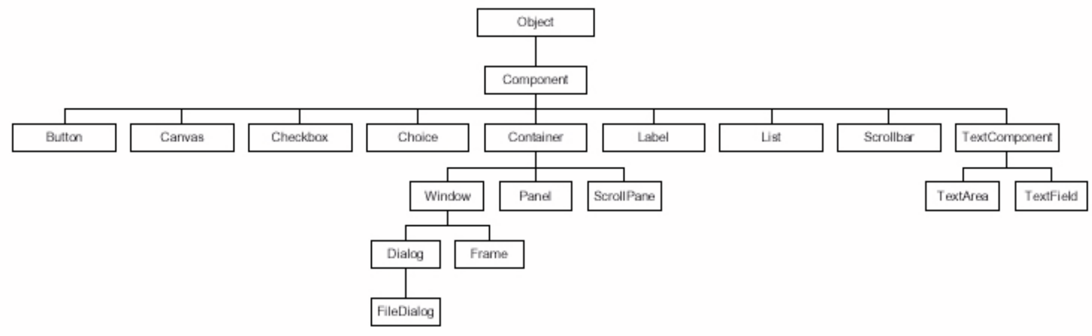


Figura 1. Estructura de clases de AWT

Los controles sólo se verán si los añadimos sobre un contenedor (un elemento de tipo *Container*, o cualquiera de sus subtipos). Para ello utilizamos el método **add(...)** del contenedor para añadir el control. Por ejemplo, si queremos añadir un botón a un *Panel*:

```
Button boton = new Button("Pulsame");
Panel panel = new Panel();
...
panel.add(boton);
```

Component	La clase padre *Component* no se puede utilizar directamente. Es una clase abstracta, que proporciona algunos métodos útiles para sus subclases.
Botones	Para emplear la clase **Button**, en el constructor simplemente indicamos el texto que queremos

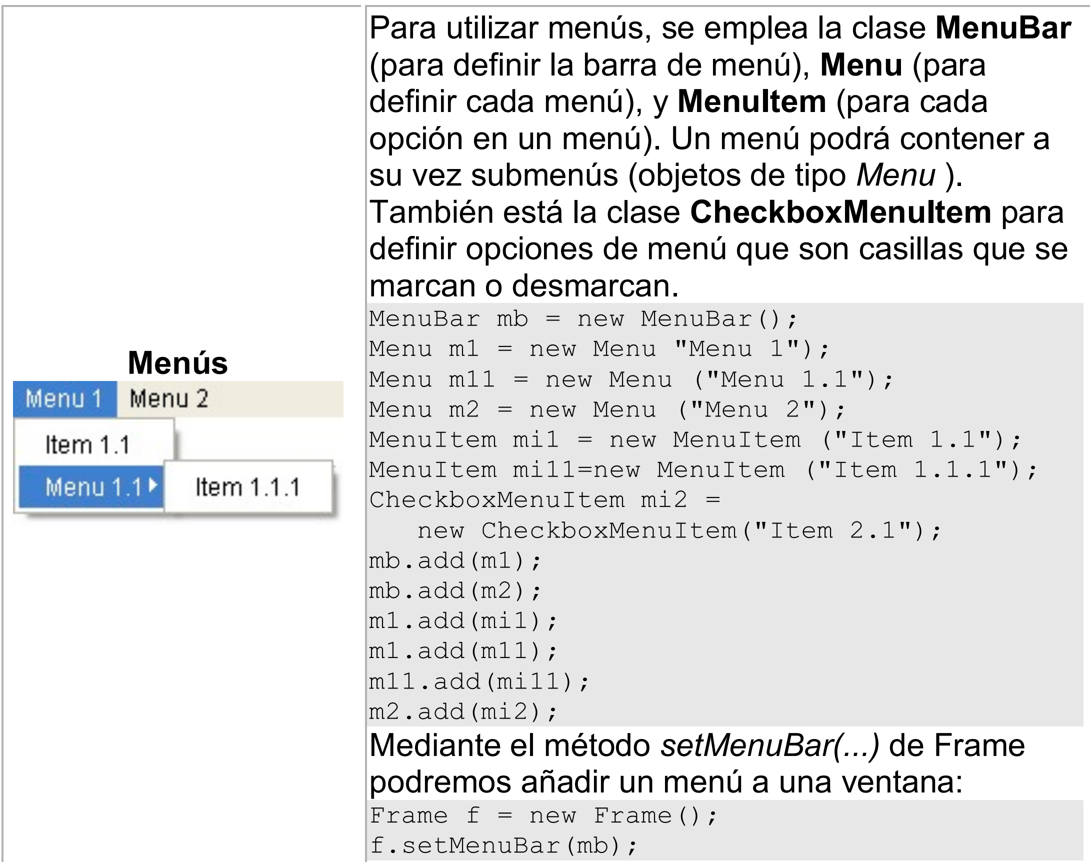
Pulsame	que tenga : `Button boton = new Button("Pulsame");`
Etiquetas Etiqueta	Para utilizar **Label**, el uso es muy similar al botón: se crea el objeto con el texto que queremos darle: `Label etiq = new Label("Etiqueta");`

Areas de dibujo	La clase **Canvas** se emplea para heredar de ella y crear componentes personalizados.
	Accediendo al objeto *Graphics* de los elementos podremos darle la apariencia que queramos: dibujar líneas, pegar imágenes, etc:
	`Panel p = new Panel();`
	`p.getGraphics().drawLine(0, 0, 100, 100);`
	`p.getGraphics().drawImage(...);`

Casillas de verificación
Mostrar subdirectorios
	Checkbox se emplea para marcar o desmarcar opciones. Podremos tener controles aislados, o grupos de *Checkboxes* en un objeto **CheckboxGroup**, de forma que sólo una de las casillas del grupo pueda marcarse cada vez.
	`Checkbox cb = new Checkbox`
	`("Mostrar subdirectorios",`
	`false);`
	`System.out.println ("Esta marcada: " +`
	`cb.getState());`

Listas
Opcion 1
Opcion 1
Opcion 2
	Para utilizar una **lista desplegable** (objeto **Choice**), se crea el objeto y se añaden, con el método **addItem(...)** , los elementos que queramos a la lista:
	`Choice ch = new Choice();`
	`ch.addItem("Opcion 1");`
	`ch.addItem("Opcion 2");`
	`...`
	`int i = ch.getSelectedIndex();`
	Para utilizar **listas fijas** (objeto **List**), en el constructor indicamos cuántos elementos son visibles. También podemos indicar si se permite seleccionar varios elementos a la vez. Dispone de muchos de los métodos que tiene *Choice* para añadir y consultar elementos.
	`List lst = new List(3, true);`
	`lst.addItem("Opcion 1");`
	`lst.addItem("Opcion 2");`

Cuadros de texto
Hola
Texto 2
	Al trabajar con **TextField** o **TextArea**, se indica opcionalmente en el constructor el número de columnas (y filas en el caso de *TextArea*) que se quieren en el cuadro de texto.
	`TextField tf = new TextField(30);`
	`TextArea ta = new TextArea(5, 40);`
	`...`
	`tf.setText("Hola");`
	`ta.appendText("Texto 2");`
	`String texto = ta.getText();`



Menús

Para utilizar menús, se emplea la clase **MenuBar** (para definir la barra de menú), **Menu** (para definir cada menú), y **MenuItem** (para cada opción en un menú). Un menú podrá contener a su vez submenús (objetos de tipo *Menu*). También está la clase **CheckboxMenuItem** para definir opciones de menú que son casillas que se marcan o desmarcan.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu "Menu 1");
Menu m11 = new Menu ("Menu 1.1");
Menu m2 = new Menu ("Menu 2");
MenuItem mi1 = new MenuItem ("Item 1.1");
MenuItem mi11=new MenuItem ("Item 1.1.1");
CheckboxMenuItem mi2 =
   new CheckboxMenuItem("Item 2.1");
mb.add(m1);
mb.add(m2);
m1.add(mi1);
m1.add(m11);
m11.add(mi11);
m2.add(mi2);
```

Mediante el método *setMenuBar(...)* de Frame podremos añadir un menú a una ventana:

```
Frame f = new Frame();
f.setMenuBar(mb);
```

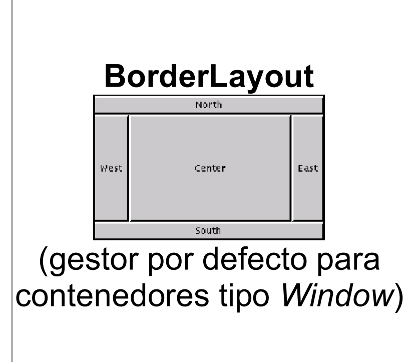
3.1.2. Gestores de disposición

Para colocar los controles Java en los contenedores se hace uso de un determinado **gestor de disposición**. Dicho gestor indica cómo se colocarán los controles en el contenedor, siguiendo una determinada distribución. Para establecer qué gestor queremos, se emplea el método *setLayout(...)* del contenedor. Por ejemplo:

```
Panel panel = new Panel();
panel.setLayout(new BorderLayout());
```

Veremos ahora los gestores más importantes:

BorderLayout



(gestor por defecto para contenedores tipo *Window*)

Divide el área del contenedor en 5 zonas: Norte (*NORTH*), Sur (*SOUTH*), Este (*EAST*), Oeste (*WEST*) y Centro (*CENTER*), de forma que al colocar los componentes deberemos indicar en el método *add(...)* en qué zona colocarlo:

```
panel.setLayout(new BorderLayout());
Button btn = new Button("Pulsame");
panel.add(btn, BorderLayout.SOUTH);
```

Al colocar un componente en una zona, se colocará sobre el que existiera anteriormente

	en dicha zona (lo tapa).

FlowLayout
(gestor por defecto para contenedores de tipo *Panel*)
	Con este gestor, se colocan los componentes en fila, uno detrás de otro, con el tamaño preferido (*preferredSize*) que se les haya dado. Si no caben en una fila, se utilizan varias.
	`panel.setLayout(new FlowLayout());`
	`panel.add(new Button("Pulsame"));`

GridLayout	Este gestor sitúa los componentes en forma de tabla, dividiendo el espacio del contenedor en celdas del mismo tamaño, de forma que el componente ocupa todo el tamaño de la celda. Se indica en el constructor el número de filas y de columnas. Luego, al colocarlo, va por orden (rellenando filas de izquierda a derecha).
	`panel.setLayout(new GridLayout(2,2));`
	`panel.add(new Button("Pulsame"));`
	`panel.add(new Label("Etiqueta"));`

Sin gestor	Si especificamos un gestor ***null***, podremos colocar a mano los componentes en el contenedor, con métodos como **setBounds(...)** , o **setLocation(...)**:
	`panel.setLayout(null);`
	`Button btn = new Button ("Pulsame");`
	`btn.setBounds(0, 0, 100, 30);`
	`panel.add(btn);`

Ejemplo: Vemos el aspecto de algunos componentes de AWT, y el uso de gestores de disposición en este ejemplo:

Código

El código nos muestra cómo se crea una clase que es una ventana principal (hereda de *Frame*), y define un gestor que es un *GridLayout*, con 4 filas y 2 columnas. En ellas vamos colocando etiquetas (*Label*), botones (*Button*), casillas de verificación (*Checkbox*), listas desplegables (*Choice*) y cuadros de texto (*TextField*). Además, se crea un menú con diferentes opciones.

3.1.3. Modelo de Eventos en Java

Hasta ahora hemos visto qué tipos de elementos podemos colocar en una aplicación visual con AWT, y cómo colocarlos sobre los distintos contenedores que nos ofrece la librería. Pero sólo con esto nuestra aplicación no hace nada: no sabemos cómo emitir una determinada respuesta al pulsar un botón, o

realizar una acción al seleccionar una opción del menú. Para definir todo esto se utilizan los llamados **eventos**.

Entendemos por **evento** una acción o cambio en una aplicación que permite que dicha aplicación produzca una respuesta. El **modelo de eventos** de AWT se descompone en dos grupos de elementos: las fuentes y los oyentes de eventos. Las **fuentes** son los elementos que generan los eventos (un botón, un cuadro de texto, etc), mientras que los **oyentes** son elementos que están a la espera de que se produzca(n) determinado(s) tipo(s) de evento(s) para emitir determinada(s) respuesta(s).

Para poder gestionar eventos, necesitamos definir el **manejador de eventos** correspondiente, un elemento que actúe de oyente sobre las fuentes de eventos que necesitemos considerar. Cada tipo de evento tiene asignada una **interfaz**, de modo que para poder gestionar dicho evento, el manejador deberá implementar la interfaz asociada. Los oyentes más comunes son:

ActionListener	Para eventos de acción (pulsar un *Button* , por ejemplo)
ItemListener	Cuando un elemento (*Checkbox*, *Choice* , etc), cambia su estado
KeyListener	Indican una acción sobre el teclado: pulsar una tecla, soltarla, etc.
MouseListener	Indican una acción con el ratón que no implique movimiento del mismo: hacer click, presionar un botón, soltarlo, entrar / salir...
MouseMotionListener	Indican una acción con el ratón relacionada con su movimiento: moverlo por una zona determinada, o arrastrar el ratón.
WindowListener	Indican el estado de una ventana

Cada uno de estos tipos de evento puede ser producido por diferentes fuentes. Por ejemplo, los *ActionListeners* pueden producirse al pulsar un botón, elegir una opción de un menú, o pulsar Intro. Los *MouseListener* se producen al pulsar botones del ratón, etc.

Toda la gestión de eventos se lleva a cabo desde el paquete **java.awt.event**.

Modos de definir un oyente

Supongamos que queremos realizar una acción determinada al pulsar un botón. En este caso, tenemos que asociar un *ActionListener* a un objeto *Button*, e indicar dentro de dicho *ActionListener* qué queremos hacer al pulsar el botón. Veremos que hay varias formas de hacerlo:

1. Que la propia clase que usa el control implemente el oyente

```
class MiClase implements ActionListener
{
    public MiClase()
    {
           ...
        Button btn = new Button("Boton");
        btn.addActionListener(this);
           ...
    }

    public void actionPerformed(ActionEvent e)
    {
        // Aqui va el codigo de la accion
    }
}
```

2. Definir otra clase aparte que implemente el oyente

```
class MiClase
{
     public MiClase()
     {
        ...
        Button btn = new Button("Boton");
        btn.addActionListener(new MiOyente());
        ...
     }
}

class MiOyente implements ActionListener
{
     public void actionPerformed(ActionEvent e)
     {
         // Aqui va el codigo de la accion
     }
}
```

3. Definir una instancia interna del oyente

```
class MiClase
{
     public MiClase()
     {
         ...
         Button btn = new Button("Boton");
         btn.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
                // Aqui va el codigo de la accion
            }
         });
         ...
    }
}
```

Uso de los "adapters"

Algunos de los oyentes disponibles (como por ejemplo *MouseListener,* consultad su API) tienen varios métodos que hay que implementar si queremos definir el oyente. Este trabajo puede ser bastante pesado e innecesario si sólo queremos usar algunos métodos. Por ejemplo, si sólo queremos hacer algo al

hacer click con el ratón, deberemos redefinir el método *mouseClicked*, pero deberíamos escribir también los métodos *mousePressed, mouseReleased*, etc, y dejarlos vacíos.

Una solución a esto es el uso de los *adapters*. Asociado a cada oyente con más de un método hay una clase ...*Adapter* (para *MouseListener* está *MouseAdapter* , para *WindowListener* está *WindowAdapter*, etc). Estas clases implementan las interfaces con las que se asocian, de forma que se tienen los métodos implementados por defecto, y sólo tendremos que sobreescribir los que queramos modificar.

Veamos la diferencia con el caso de *MouseListener*, suponiendo que queremos asociar un evento de ratón a un *Panel* para que haga algo al hacer click sobre él.

1. Mediante Listener:

```
class MiClase
{
     public MiClase()
     {
        ...
        Panel panel = new Panel();
        panel.addMouseListener(new MouseListener()
        {
           public void mouseClicked(MouseEvent e)
           {
               // Aqui va el codigo de la accion
           }

           public void mouseEntered(MouseEvent e)
           {
               // ... No se necesita
           }

           public void mouseExited(MouseEvent e)
           {
               // ... No se necesita
           }

           public void mousePressed(MouseEvent e)
           {
               // ... No se necesita
           }

           public void mouseReleased(MouseEvent e)
           {
               // ... No se necesita
           }
        });
        ...
    }
}
```

Vemos que hay que definir todos los métodos, aunque muchos queden vacíos porque no se necesitan.

2. Mediante Adapter:

```
class MiClase
{
     public MiClase()
     {
          ...
          Panel panel = new Panel();
          panel.addMouseListener(new MouseAdapter()
          {
             public void mouseClicked(MouseEvent e)
             {
                  // Aqui va el codigo de la accion
             }
          });
          ...
     }
}
```

Vemos que aquí sólo se añaden los métodos necesarios, el resto ya están implementados en *MouseAdapter* (o en el *adapter* que corresponda), y no hace falta ponerlos.

Ejemplo: Vemos el uso de oyentes en este ejemplo: Código

La aplicación muestra distintos tipos de eventos que podemos definir sobre una aplicación:

- Tenemos una etiqueta llamada *lblCont*. Tiene definido un evento de tipo *MouseListener* para que, cuando el ratón esté dentro de la etiqueta, muestre un texto, cuando esté fuera, muestre otro.
- Por otra parte, tenemos un botón (variable *btn*) con un evento de tipo *ActionListener* para que, al pulsar sobre él, se incremente en 1 un contador que hay en un cuadro de texto.
- También tenemos una lista desplegable (variable *ch*) que tiene un evento de tipo *ItemListener* para que, al cambiar el elemento seleccionado, se actualiza el valor del contador del cuadro de texto a dicho elemento seleccionado.
- Finalmente, la ventana principal tiene un evento de tipo *WindowListener* para que, al pulsar el botón de cerrar la ventana, se finalice la aplicación (son las últimas líneas de código del constructor).

3.1.4. Pasos generales para construir una aplicación gráfica con AWT

Con todo lo visto hasta ahora, ya deberíamos ser capaces de construir aplicaciones más o menos completas con AWT. Para ello, los pasos a seguir son:

1. Definir la clase principal, que será la ventana principal de la aplicación

Cualquier aplicación AWT debe tener una ventana principal que sea de tipo **Frame**. Así pues lo primero que debemos hacer es definir qué clase hará de *Frame*:

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

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

public class MiAplicacion extends Frame
{
     public MiAplicacion()
     {
         setSize(500, 400);
         setLayout(new GridLayout(1, 1));
         ...
     }
}
```

podemos definir un constructor, y dentro hacer algunas inicializaciones como el tamaño de la ventana, el gestor de disposición, etc.

2. Colocar los controles en la ventana

Una vez definida la clase, e inicializada la ventana, podemos colocar los componentes en ella:

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

public class MiAplicacion extends Frame
{
     public MiAplicacion()
     {
         setSize(500, 400);
         setLayout(new GridLayout(1, 1));

         Button btn = new Button("Hola");
         this.add(btn);

         JPanel p = new JPanel();
         JLabel l = new JLabel("Etiqueta");
         JLabel l2 = new JLabel ("Otra etiqueta");
         p.add(l);
         p.add(l2);
         this.add(p);
     }
}
```

En nuestro caso añadimos un botón, y un panel con 2 etiquetas.

3. Definir los eventos que sean necesarios

Escribimos el código de los eventos para los controles sobre los que vayamos a actuar:

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

public class MiAplicacion extends Frame
{
     public MiAplicacion()
     {
         setSize(500, 400);
```

```
        setLayout(new GridLayout(1, 1));

        Button btn = new Button("Hola");
        this.add(btn);

        JPanel p = new JPanel();
        JLabel l = new JLabel("Etiqueta");
        JLabel l2 = new JLabel ("Otra etiqueta");
        p.add(l);
        p.add(l2);
        this.add(p);

        btn.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println ("Boton pulsado");
            }
        });

        addWindowListener(new WindowAdapter()
        {
            public void windowClosing (WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}
```

4. Mostrar la ventana

Desde el método *main* de nuestra clase principal podemos hacer que se muestre la ventana:

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

public class MiAplicacion extends Frame
{
    public MiAplicacion()
    {
        ...
    }

    public static void main(String[] args)
    {
        MiAplicacion ma = new MiAplicacion();
        ma.show();
    }
}
```

5. Definir otras subventanas o diálogos

Aparte de la clase principal, podemos definir otros *Frames* en otras clases, e interrelacionarlos. También podemos definir diálogos (*Dialogs*) que dependan de una ventana principal (*Frame*) y que se muestren en un momento dado.

3.2. Swing

3.2.1. Introducción a Swing

Anteriormente se ha visto una descripción de los controles *AWT* para construir aplicaciones visuales. En cuanto a estructura, no hay mucha diferencia entre los controles proporcionados por *AWT* y los proporcionados por *Swing*: éstos se llaman, en general, igual que aquéllos, salvo que tienen una "J" delante; así, por ejemplo, la clase *Button* de *AWT* pasa a llamarse *JButton* en *Swing* , y en general la estructura del paquete de Swing (**javax.swing**) es la misma que la que tiene *java.awt*.

Pero yendo más allá de la estructura, existen importantes diferencias entre los componentes *Swing* y los componentes *AWT*:

- Los componentes *Swing* están escritos sin emplear código nativo, con lo que ofrecen más versatilidad multiplataforma (podemos dar a nuestra aplicación un aspecto que no dependa de la plataforma en que la estemos ejecutando).

- Los componentes *Swing* ofrecen más capacidades que los correspondientes *AWT*: los botones pueden mostrar imágenes, hay más facilidades para modificar la apariencia de los componentes, etc.

- Al mezclar componentes *Swing* y componentes *AWT* en una aplicación, se debe tener cuidado de emplear contenedores *AWT* con elementos *Swing*, puesto que los contenedores pueden solapar a los elementos (se colocan encima y no dejan ver el componente).

3.2.2. Características específicas de Swing

Resumen de controles

Los controles en Swing tienen en general el mismo nombre que los de AWT, con una "J" delante. Así, el botón en Swing es *JButton* , la etiqueta es *JLabel* , etc. Hay algunas diferencias, como por ejemplo *JComboBox* (el equivalente a *Choice* de AWT), y controles nuevos. Vemos aquí un listado de algunos controles:

JComponent	La clase padre para los componentes Swing es *JComponent* , paralela al *Component* de AWT.
Botones	Se tienen botones normales (**JButton**), de verificación (**JCheckBox**), de radio (**JRadioButton**), etc, similares a los *Button*, *Checkbox* de AWT, pero con más posibilidades (se pueden añadir imágenes, etc).

Etiquetas	Las etiquetas son **JLabel**, paralelas a las *Label* de AWT pero con más características propias (iconos, etc).
Cuadros de texto	Las clases **JTextField** y **JTextArea** representan los cuadros de texto en Swing, de forma parecida a los *TextField* y *TextArea* de AWT.
Listas	Las clases **JComboBox** y **JList** se emplean para lo mismo que *Choice* y *List* en AWT.
Diálogos y ventanas	Las clases **JDialog** (y sus derivadas) y **JFrame** se emplean para definir diálogos y ventanas. Se tienen algunos cuadros de diálogo específicos, para elegir ficheros (*JFileChooser*), para elegir colores (*JColorChooser*), etc.
Menús	Con **JMenu**, **JMenuBar**, **JMenuItem**, se construyen los menús que se construian en AWT con *Menu*, *MenuBar* y *MenuItem*.

Gestores de disposición y modelo de eventos

Los gestores de disposición de Swing son los mismos que los vistos en AWT. Sólo debemos tener en cuenta que hay ciertos métodos de **JFrame** a los que no podemos acceder directamente (en *Frame* sí podemos), y se debe acceder a ellos a través de un método llamado *getContentPane*. Ejemplos de estos métodos son **add** y **setLayout**, que pasan a usarse de la siguiente forma:

```
public class MiFrame extends JFrame
{
    public MiFrame()
    {
        Button b = new Button("Hola");

        this.add(b);                                      // ERROR
        this.getContentPane().add(b);                     // OK
        this.setLayout(new BorderLayout());               // ERROR
        this.getContentPane().setLayout(new BorderLayout()); // OK
    }
...
```

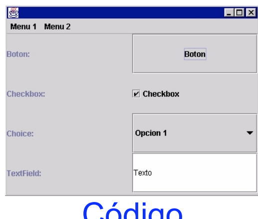
El modelo de eventos también es el mismo que el visto en AWT.

Otras características

Swing ofrece otras posibilidades, que se comentan brevemente:

- Uso de **acciones**, objetos **Action** que coordinan tareas realizadas por distintos elementos.
- Uso de **bordes**, elementos que bordean los controles y ofrecen un mejor aspecto visual a la aplicación.
- Uso de **iconos**: algunos componentes permiten que se les indique un icono a mostrar, mediante la clase **ImageIcon**.
- Uso de la **apariencia** (*look and feel*): podemos indicar qué aspecto queremos que tenga la aplicación: específico de Windows, de Motif, etc.
- Uso de **hilos** para gestionar eventos: algunos eventos pueden bloquear componentes durante mucho tiempo, y es mejor separar el tratamiento del evento en un hilo para liberar el componente.
- Uso de **temporizadores**: con la clase **Timer** podemos definir acciones que queremos ejecutar en un momento determinado o con una periodicidad determinada.

Ejemplo: Vemos el aspecto de algunos componentes de Swing, paralelo al visto en el tema de AWT:



Código

Observad cómo se pasa una aplicación de AWT a Swing. Hay que cambiar los componentes de AWT por los equivalentes de Swing (*Frame* por *JFrame*, *Button* por *JButton*, etc), y luego hay algunos métodos a los que no podemos llamar directamente, como son los métodos **add** y **setLayout** de *JFrame*, a los que se debe llamar a través del método **getContentPane** (IMPORTANTE: esto se aplica única y exclusivamente a ciertos métodos de *JFrame*, como los indicados).

Ejemplo: Vemos un ejemplo de uso de iconos y temporizadores (como icono se emplea esta imagen): Código

Para utilizar los iconos se utiliza un objeto de tipo **ImageIcon** y se dice cuál es el fichero de la imagen. Para el temporizador, se utiliza un objeto de tipo **Timer**. Vemos que se define un *ActionListener*, que se ejecuta cada X milisegundos (1000, en este caso), ejecutando así un trabajo periódico (mediante el método *setRepeats* del *Timer* indicamos que el listener se ejecute periódicamente, o no).

3.3. Applets

Los ejemplos vistos hasta ahora son **aplicaciones**, puesto que son instancias de la clase **Frame** o **JFrame**, y por tanto son ventanas que pueden ejecutarse independientemente.

Un **applet** es una aplicación normalmente corta (aunque no hay límite de tamaño), cuya principal funcionalidad es ser accesible a un servidor Internet (una aplicación que pueda visualizarse desde un navegador).

La forma de definir un applet es muy similar a la definición de una aplicación, salvo por algunas diferencias:

- No se hereda de **Frame**, sino de **Applet** (clase *java.applet.Applet*)
- No hay constructor, en su lugar hay un método **init()** que veremos a continuación
- No hay método **main()**, puesto que el applet no puede autoejecutarse. Lo que se ejecuta es la página HTML para ver el applet en el navegador.

Un ejemplo básico de applet sería:

```
public class MiApplet extends Applet
{
    public void init()
    {
        setLayout(new BorderLayout());
```

```
        Button b = new Button("Hola");
        add(b, BorderLayout.NORTH);
        ...
    }
}
```

El **appletviewer** es un navegador mínimo distribuido con Java, que espera como argumento un fichero HTML, que contendrá una marca indicando el código que cargará el *appletviewer* . Podemos así cargar un applet que esté en una página HTML con:

```
appletviewer <fichero HTML>
```

En la página HTML debemos incluir etiquetas que permitan cargar el applet, como son las etiquetas APPLET u OBJECT:

```
<HTML>
  <BODY>
  ...
  <APPLET CODE = MiApplet.class WIDTH = 300 HEIGHT = 100>
  </APPLET>
  ...
  </BODY>
</HTML>
```

Donde se indican el fichero *.class* compilado del applet, la anchura y altura. Este código se coloca en un fichero HTML y puede verse desde cualquier navegador que soporte Java, o con el programa *appletviewer* .

La clase **Applet** tiene unos métodos predefinidos para controlar los applets:

- **init ( )**: este método se llama cada vez que el appletviewer carga por primera vez la clase. En él deben inicializarse las características del applet que se quieran (tamaño, imágenes, controles, valores de variables, etc).
- **start ( )**: llamada para arrancar el applet cada vez que es visitado.
- **stop ( )**: llamada para detener la ejecución del applet. Se llama cuando el applet desaparece de la pantalla.
- **destroy ( )**: se llama cuando ya no se va a usar más el applet, y hay que liberar los recursos dispuestos por el mismo.

3.3.1. Applets Swing

La única diferencia entre los applets construidos en *AWT* y los construidos con *Swing* es que éstos heredan de la clase *JApplet* en lugar de la clase *Applet*. Pero se tiene el inconveniente de que actualmente sólo la utilidad *appletviewer* está preparada para ejecutar applets de *Swing* con Java 1.2 o posteriores. Para el resto de navegadores deberemos contar con el Java Plug-in 1.1.1, que contiene la versión 1.0.3 de *Swing* . El resto de la estructura de los applets es la misma que para AWT.

Ejemplo: Vemos el ejemplo anterior convertido en applet. Puede verse aquí el código y aquí la página HTML con el applet.

3.4. Gráficos y animación

Hasta ahora hemos visto la creación de aplicaciones con una interfaz gráfica a partir de una serie de componentes definidos en la API de AWT y de Swing (ventanas, botones, campos de texto, etc).

En este punto veremos como dibujar nuestros propios gráficos directamente en pantalla. Para ello Java nos proporciona acceso al contexto gráfico del área donde vayamos a dibujar, permitiéndonos a través de éste modificar los pixels de este área, dibujar una serie de figuras geométricas, así como volcar imágenes en ella.

3.4.1. Gráficos en AWT

Para dibujar gráficos en un área de la pantalla, AWT nos ofrece un objeto con el contexto gráfico de dicha área, perteneciente a la clase **Graphics**. Este objeto nos ofrece una serie de métodos que nos permiten dibujar distintos elementos en pantalla. Más adelante veremos con detalle los métodos más importantes.

Este objeto **Graphics** nos lo deberá proporcionar AWT en el momento en que vayamos a dibujar, ya que no podremos obtenerlo por nuestra cuenta de ninguna otra forma.

Para dibujar en pantalla cualquier componente AWT, estos componentes proporcionan dos métodos: **paint(Graphics g)** y **update(Graphics g)**.

Estos métodos serán invocados por AWT cuando necesite dibujar su contenido en pantalla. Por ejemplo, cuando la ventana se muestre por primera vez, AWT invocará al método **paint(Graphics g)** de todos los componentes AWT que contenga la aplicación, de forma que estos se dibujen en pantalla. Cuando una aplicación minimizada se maximice, o una aplicación que estaba total o parcialmente tapada por otra pase a primer plano de nuevo, también podrá ser necesario invocar dicho método para volver a dibujar los componentes de la ventana.

Cada componente AWT definirá su propio método **paint(Graphics g)** para dibujar su contenido. Por ejemplo, un *Button* dibujará en pantalla la forma del botón y el texto. Si creamos una subclase de estos componentes y sobrescribimos su método **paint(Graphics g)** con un método propio, dentro de él podremos utilizar el objeto de contexto gráfico **Graphics** para dibujar en el área de dicho componente.

La mayoría de los componentes tienen ya definido su propio comportamiento y apariencia, por lo que no deberemos sobrescribir este método ya que el componente dejaría de funcionar correctamente. Sin embargo, hay un componente diseñado para que el usuario pueda utilizarlo como área de dibujo, este es el caso del **Canvas**. Este componente simplemente comprende un área vacía de la pantalla, en la que nosotros podremos dibujar nuestros propios gráficos sobrescribiendo su método **paint(Graphics g)**. Sobrescribiremos el método de la siguiente forma:

```
public class MiCanvas extends Canvas {
        public void paint(Graphics g) {
                // Dibujamos en el área del canvas
                // usando el objeto g proporcionado
        }
}
```

Con esto en la clase MiCanvas hemos creado un componente propio en el que nosotros controlamos lo que se dibuja en pantalla. Podremos añadir este componente a nuestra aplicación de la misma forma que añadimos cualquier otro componente AWT:

```
MiCanvas mc = new MiCanvas();
panel.add(mc);
```

Hemos de recordar que no podemos controlar cuando se invoca el método **paint(Graphics g)**, esté método será invocado por AWT en el momento en el que el SO necesite que la ventana sea redibujada. En él simplemente definimos como se dibujará el contenido de nuestro componente en pantalla, y AWT ya se encargará de invocarlo cuando sea necesario.

3.4.2 Contexto gráfico: Graphics

El objeto **Graphics** nos permitirá acceder al contexto gráfico de un determinado componente y a través de él dibujar en su área en pantalla. Vamos a ver ahora como dibujar utilizando dicho objeto.

3.4.2.1 Atributos

El contexto gráfico tendrá asociado el color del lápiz que usamos en cada momento para dibujar, así como la fuente de texto que se utilizará en el caso de que dibujemos una cadena de texto. Para consultar o modificar el color o la fuente asociadas al contexto gráfico se proporcionan los siguientes métodos:

- **Color getColor() / setColor(Color c)**: Obtiene/Establece el color del lápiz en el contexto gráfico. Deberemos utilizar un objeto de la clase **Color** para especificar el color que queremos establecer. Los objetos **Color** pueden ser construidos a partir de las componentes RGB del color deseado, o bien utilizar como color alguno de los colores predefinidos como constantes en la misma clase **Color** (*Color.red, Color.black*, etc).
- **Font getFont() / setFont (Font f)**: Obtiene/Establece la fuente que se utilizará para dibujar texto. Utilizaremos la clase **Font** para especificar la fuente de texto que vamos a utilizar.
- **setPaintMode() / setXORMode(Color c)**: Establece el modo de dibujo. El modo **Paint** dibuja directamente en el área con el color actual, sobrescribiendo con este color el valor anterior de los pixels sobre los que se dibuja. El modo **XOR** realiza una operación XOR entre el color del nuevo pixel y el que habia antes en esa posición, dibujando el resultado de dicha operación. Más adelante veremos la utilidad de este modo.

Una vez establecidos estos atributos en el contexto gráfico, podremos dibujar en él una serie de elementos utilizando una serie de métodos de **Graphics**. Estos métodos comienzan por **drawXXXX** para dibujar el contorno de una determinada forma, o **fillXXXX** para dibujar dicha forma con relleno.

El sistema de coordenadas del área en pantalla tendrá la coordenada (0,0) en su esquina superior izquierda, y las coordenadas serán positivas hacia la derecha (coordenada x) y hacia abajo (coordenada y), tal como se muestra a continuación:

Figura 1. Coordenadas del área de dibujo

3.4.2.2 Figuras

El contexto gráfico nos ofrece una serie de métodos para dibujar en él las principales primitivas geométricas básicas:

- **draw- / fillRect(int x, int y, int width, int height)**: Dibuja un rectángulo dadas sus coordenadas de inicio y el ancho y el alto.
- **draw- / fillOval(int x, int y, int width, int height)**: Dibuja una elipse dadas las coordenadas de inicio y el ancho y el alto del rectángulo que la contiene.
- **drawLine(int x1, int y1, int x2, int y2)**: Dibuja una línea desde el punto *(x1,y1)* hasta *(x2,y2)*.
- **draw- / fillPolygon(int [ ] xPoint, int [ ] yPoint, int nPoint)**: Dibuja un polígono cerrado cuyos *nPoint* vértices tienen las coordenadas *(x,y)* especificadas en las listas *xPoint* e *yPoint* proporcionadas.

Por ejemplo, el siguiente canvas aparecerá con un dibujo de un círculo rojo y un rectángulo (cuadrado) verde:

```
public class MiCanvas extends Canvas {
        public void paint(Graphics g) {
                g.setColor(Color.red);
                g.fillOval(10,10,50,50);
                g.setColor(Color.green);
                g.fillRect(60,60,50,50);
        }
}
```

3.4.2.3 Texto

Aparte de dibujar figuras geométricas también podremos dibujar cadenas de texto. Para ello se proporciona el método **drawString(String s, int x, int y)** que dibuja la cadena *s* en las coordenadas *(x,y)*. Este punto corresponderá al inicio de la cadena, en la línea de base del texto como se muestra a continuación:

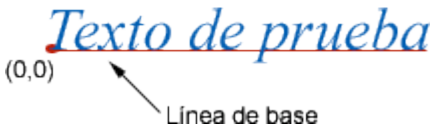


Figura 2. Línea de base del texto

Con esto dibujaremos un texto en pantalla, pero podemos querer hacer por ejemplo que el texto sea sensible a pulsaciones del ratón sobre él. Lo único que nos proporciona AWT es la posición del ratón dentro del área, por lo que para saber si está sobre el texto tendremos que saber que región ocupa el texto. Aquí es donde encontramos el problema, según la fuente, la cadena escrita, y el contexto donde la escribamos, el texto puede tener distintas dimensiones, y nosotros sólo conocemos las coordenadas de comienzo. La solución a esto la proporciona el objeto **FontMetrics**, que podemos obtener llamando al método **getFontMetrics(Font f)** del contexto gráfico o del componente AWT (ambos tipos de objetos contienen este método). Este objeto nos dará información sobre las métricas de dicha fuente en este contexto. De este objeto podemos sacar la siguiente información:

- **int stringWidth(String s)**: Nos devuelve el ancho que tendrá la cadena *s* en pixels.
- **int getAscent()** / **int getMaxAscent()**: Nos devuelve la altura típica o la máxima altura respectivamente que tendrán los caracteres (valores positivos).
- **int getDescent()** / **int getMaxDescent()**: Nos devuelve lo que descienden los carácteres desde la línea de base, tanto el descenso típico como el máximo respectivamente (valores positivos).

Con estas medidas podremos conocer exactamente los límites de una cadena de texto, tal como se muestra a continuación:

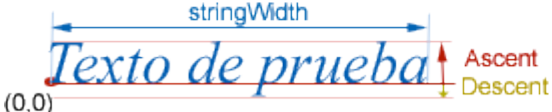


Figura 3. Métricas del texto

3.4.2.4 Imágenes

Por último, un método importante es el que nos permite volcar una imagen al área de dibujo. Las imágenes en Java se encapsulan mediante la clase **Image**. Podemos o bien crearnos una imagen vacia para dibujar nosotros en ella, o cargar imágenes desde ficheros. Para cargar una imagen de un fichero en caso de un **Applet**, simplemente deberemos llamar al método **getImage(URL**

url) de la clase **Applet** que nos devolverá el objeto **Image** con la imagen cargada, que podrá estar en formato GIF, JPG o PNG. En el caso de una aplicación deberemos seguir los siguientes pasos:

Obtener la implementación del toolkit de AWT a partir de cualquier componente AWT de nuestra aplicación. Si tenemos una ventana con un canvas, tanto la ventana como el canvas nos servirían dado que ambos son componentes AWT. Sobre el componente llamaremos al método **getToolkit()** para obtener el objeto **Toolkit**:

```
Canvas mc = new MiCanvas();
Toolkit toolkit = mc.getToolkit();
```

También podemos obtener el toolkit por defecto si no podemos acceder a ningún componente AWT:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
```

Utilizar el método **createImage(String fichero)** del Toolkit para cargar la imagen del fichero de nombre *fichero*:

```
Image img = toolkit.createImage("foto.jpg");
```

Una vez obtenida la imagen podremos dibujarla en el contexto gráfico con **drawImage(Image img, int x, int y, ImageObserver obs)**. Con esto dibujaremos la imagen *img* en las coordenadas *(x,y)* del área. Además necesitamos proporcionar el objeto **ImageObserver** que se utilizará para avisar a la aplicación cuando la imagen esté cargada del todo, y por lo tanto puede mostrarla por pantalla. Cualquier componente AWT capaz de mostrar imagenes implementará la interfaz **ImageObserver**, por lo que podremos utilizarlo en la llamada a dicho método. Por ejemplo, si dibujamos la imagen en un **Canvas**, utilizaremos el mismo **Canvas** como **ImageObserver** ya que es el componente en el que vamos a observar la imagen:

```
public class MiCanvas extends Canvas {
        public void paint(Graphics g) {
                Toolkit toolkit = getToolkit();
                Image img = toolkit.createImage("foto.jpg");

                g.drawImage(img, 0, 0, this);
        }
}
```

Si no necesitamos utilizar un **ImageObserver** podemos especificar *null* en este parámetro. Esto será útil cuando trabajemos con imágenes que sabemos que ya están cargadas, y cuando no tengamos la referencia al componente donde se va a dibujar la imagen.

3.4.2.5 Otros métodos

Otros métodos útiles del contexto gráfico son:

- **clearRect(int x, int y, int width, int height)**: Vacia el rectángulo especificado del área. El color del rectángulo será el del color de fondo del componente AWT (el contexto gráfico no define color de fondo, este color es propio del componente AWT en el que se dibuja).
- **clipRect(int x, int y, int width, int height)**: Define un rectángulo de recorte. Cuando definimos un área de recorte en el contexto gráfico, sólo se dibujarán en pantalla los pixels que caigan dentro de este área. El espacio que ocupa el componente en el que dibujamos es un área de recorte impuesta por el sistema. Nunca se dibujarán los pixels que escribamos fuera de este espacio. Este método establece un recorte en el área de recorte anterior, si ya existia un rectángulo de recorte, el nuevo rectángulo de recorte será la intersección de ambos. Si queremos eliminar el área de recorte anterior deberemos usar el método **setClip(null)**.
- **copyArea(int x, int y, int width, int height, int dx, int dy)**: Copia el área dentro del rectángulo especificado en las coordenadas de destino *(dx, dy)*.

3.4.3 Animaciones

Hasta ahora hemos visto como dibujar gráficos en pantalla, pero lo único que hacemos es definir un método que se encargue de dibujar el contenido del componente, y ese método será invocado cuando el sistema necesite dibujar la ventana.

Sin embargo puede interesarnos cambiar dinámicamente los gráficos de nuestro componente. Para ello deberemos indicar el momento en el que queremos que se redibujen los gráficos, ya que el sistema por si solo sólo llamará a **paint(Graphics g)** cuando sea necesario volver a dibujar la ventana porque su contenido se ha perdido, pero no lo llamará cuando hayamos realizado cambios.

3.4.3.1 Redibujado del área

Para forzar que se redibuje el área del componente, deberemos llamar al método **repaint()** del componente (del canvas por ejemplo). Con eso estamos solicitando al sistema que se repinte el componente, pero no lo repinta en el mismo momento en el que se llama. El sistema introducirá esta solicitud en la cola de ventanas de debe repintar, y cuando tenga tiempo repintará su contenido.

```
MiCanvas mc = new MiCanvas();
...
mc.repaint();
```

En este caso para repintar el componente no llamará a su método **paint(Graphics g)**, sino al método **update(Graphics g)**. Este es el método que se encarga de actualizar los gráficos. Su implementación por defecto consiste en borrar los gráficos actuales del área del componente, y llamar a **paint(Graphics g)** para pintarlo de nuevo.

Imaginemos que estamos moviendo un rectángulo por pantalla. El rectángulo irá cambiando de posición, y en cada momento lo dibujaremos en la posición en la que se encuentre. Pero si no borramos el contenido de la pantalla en el instante anterior, el rectángulo aparecerá en todos los lugares donde ha estado en instantes anteriores produciendo este efecto indeseable de dejar rastro. Por ello el método **update(Graphics g)** vacía todo el área del componente antes de invocar a **paint(Graphics g)** para que dibuje los gráficos en el instante actual.

Ejemplo de efecto flicker

Sin embargo, al estar vaciando y volviendo a dibujar en el componente, veremos en éste un efecto parpadeo (*flicker*). Por ello, si queremos que nuestra aplicación no muestre este aspecto de aplicación *amateur*, deberemos sobrescribir el método **update(Graphics g)** para que su única función sea llamar a **paint(Graphics g)** sin vaciar previamente el área:

```
public class MiCanvas {
        public void update(Graphics g) {
                paint(g);
        }

        public void paint(Graphics g) {
                // Aqui dibujamos el contenido del componente
        }
}
```

Pero ahora nos encontramos con otros problemas. Al no borrar la pantalla los objetos pueden dejar rastro. Además puede que queramos dibujar varios componentes en pantalla, y si los dibujamos uno detrás de otro puede producirse el efecto poco deseable de ver como se va construyendo la imagen. Para evitar que esto ocurra y conseguir unas animaciones limpias, utilizaremos la técnica del *doble buffer*.

Ejemplo de efecto rastro

3.4.3.2 Técnica del doble buffer

La técnica del *doble buffer* consiste en dibujar todos los elementos que queremos mostrar en una imagen en memoria, y una vez se ha dibujado todo, volcarlo a pantalla como una unidad. De esta forma, mientras se va dibujando la imagen, como no se hace directamente en pantalla no veremos efectos de parpadeo al borrar el contenido anterior, ni veremos como se va creando la imagen, en pantalla se mostrará la imagen cuando esté completa.

Ejemplo de doble buffer

Para utilizar esta técnica lo primero que deberemos hacer es crearnos el denominado *back buffer* que será el buffer en memoria donde dibujamos la imagen. Para implementarlo en Java utilizaremos una imagen (objeto **Image**) que tendremos en memoria, y sobre la que dibujaremos el contenido que queramos mostrar. Deberemos crear una imagen del mismo tamaño del

componente en el que vamos a dibujar. Para crear una imagen en blanco podemos usar el método **createImage(int width, int height)** que se encuentra en cualquier componente AWT (**Component**), como por ejemplo nuestro canvas, y crea una imagen vacia con las dimensiones especificadas. Tendremos que crearla con las dimensiones del componente:

```
Image backbuffer = createImage(getWidth(), getHeight());
```

Al igual que cada componente tiene un contexto gráfico asociado en el que podemos dibujar, una imagen en memoria también tendrá su contexto gráfico. Mientras el contexto gráfico de los componentes hace referencia a la pantalla, el de una imagen hará referencia a un espacio de memoria en el que se almacena la imagen, pero la forma de dibujar en ambos se realiza de la misma forma a través de la misma interfaz (objeto **Graphics**). Para obtener el contexto gráfico de una imagen utilizaremos el método **getGraphics()** de la misma:

```
Graphics offScreen = backbuffer.getGraphics();
```

Una vez obtenido este contexto gráfico, dibujaremos todo lo que queremos mostrar en él, en lugar de hacerlo en pantalla. Una vez hemos dibujado todo el contenido en este contexto gráfico, deberemos volcar la imagen a pantalla para que ésta se haga visible:

```
g.drawImage(backbuffer, 0, 0, this);
```

La imagen conviene crearla una única vez, ya que la animación puede redibujar frecuentemente, y si cada vez que lo hacemos creamos un nuevo objeto imagen estaremos malgastando memoria inutilmente. Es buena práctica de programación en Java instanciar nuevos objetos las mínimas veces posibles, intentando reutilizar los que ya tenemos.

Podemos ver como quedaría nuestra clase ahora:

```
public MiCanvas extends Canvas {

        // Backbuffer
        Image backbuffer = null;

        // Ancho y alto del backbuffer
        int width, height;

        // Coordenadas del rectangulo dibujado
        int x, y;

        public void update(Graphics g) {
                paint(g);
        }

        public void paint(Graphics g) {
                // Solo creamos la imagen la primera vez
                // o si el componente ha cambiado de tamaño
                if( backbuffer == null ||
                        width != getWidth() ||
```

```
                        height != getHeight() )
                {
                        width = getWidth();
                        height = getHeight();
                        backbuffer = createImage(width, height);
                }

                Graphics offScreen = backbuffer.getGraphics();

                // Vaciamos el área de dibujo

                offScreen.clearRect(0,0,getWidth(), getHeight());

                // Dibujamos el contenido en offScreen
                offScreen.setColor(Color.red);
                offScreen.fillRect(x,y,50,50);

                // Volcamos el back buffer a pantalla
                g.drawImage(backbuffer,0,0,this);
        }
}
```

En este ejemplo se dibuja un rectángulo rojo en la posición *(x,y)* de la pantalla que podrá ser variable, tal como veremos a continuación añadiendo a este ejemplo métodos para realizar la animación.

3.4.3.3 Código para la animación

Si queremos hacer una animación tendremos que ir cambiando ciertas propiedades de los objetos de la imagen (por ejemplo su posición) y solicitar que se redibuje tras cada cambio. El bucle para la animación podría ser el siguiente:

```
public class MiCanvas extends Canvas {
        ...
        public void anima() {
                // El rectangulo comienza en (10,10)
                x = 10;
                y = 10;

                while(x < 100) {
                        x++;
                        repaint();

                        try {
                                Thread.currentThread().sleep(100);
                        } catch(InterruptedException e) {}
                }
        }
}
```

Con este código de ejemplo veremos una animación en la que el rectángulo que dibujamos partirá de la posición (10,10) y cada 100ms se moverá un pixel hacia la derecha, hasta llegar a la coordenada (100,10).

Para ello sólo tendremos que invocar el método *anima()* de nuestro componente. Pero es importante no invocar este método directamente desde cualquier respuesta a un evento, como puede ser la pulsación de un boton:

```
public void actionPerformed(ActionEvent e) {
        // ¡¡¡ No hay que hacer esto !!!
        mc.anima();
}
```

¿Por qué no debemos hacer esto? Parece dificil ver que esto pueda causar algún problema, pero debemos pensar que Java tiene un hilo en el que se tratan los eventos que se van produciendo, y este mismo hilo será el que se encargue de repintar la pantalla cuando se lo solicitemos. Sin embargo, si desde este hilo llamamos al método que realiza la animación y éste no devuelve el control hasta que la animación no ha terminado, al no continuar el hilo no podrá repintar el contenido de la pantalla, por mucho que se lo pidamos dentro de nuestra función de animación (la ventana de la aplicación quedaría como "colgada"). Esto producirá el efecto de no ver la animación mientras se está realizando, sólo se actualizará la pantalla una vez haya terminado la animación, por lo tanto se producirá un salto desde el estado inicial hasta el estado final.

Por lo tanto, si queremos hacer una animación lo mejor será crear un hilo independiente que se encargue de realizar dicha animación, y de esta manera el hilo de procesamiento de eventos pueda continuar realizando sus tareas mientras se ejecuta la animación:

```
public class MiCanvas extends Canvas implements Runnable {
        ...
        public void run() {
                anima();
        }
}
```

Ahora si que podremos invocar la animación desde el código de los eventos creando un hilo independiente que se encargue de ella:

```
public void actionPerformed(ActionEvent e) {
        // Así si que funciona
        Thread t = new Thread(mc);
        t.start();
}
```

3.4.3.4 Modo XOR

Si no queremos tener que sobrescribir la pantalla entera cada vez que se actualiza, podemos utilizar otra técnica que se basa en el modo XOR de dibujo para evitar que los objetos dejen rastro al ser modificados.

Lo que haremos en este caso es activar el modo XOR mientras estemos modificando un objeto. Cada vez que realicemos un cambio en el objeto (movimiento, cambio de tamaño o forma), lo que haremos será dibujar de nuevo el objeto en su posición anterior, de forma que al estar activado el modo

XOR se borrará el objeto, y a continuación lo dibujamos en su nueva posición. Este modo es muy utilizado en programas de dibujo, mientras estamos dibujando figuras en pantalla.

3.4.4 API de Java 2D

Java 2D es una nueva API introducida a partir de JDK 1.2, que extiende AWT para proporcionar un extenso conjunto de funcionalidades para trabajar con gráficos 2D, texto e imágenes. Aporta una serie de formas primitivas básicas con las que podremos trabajar.

Además de mostrar gráficos por pantalla, es capaz de sacarlos a través de la impresora, utilizando un modelo uniforme de render para ambos casos.

El mecanismo de render es el mismo que vimos en AWT para versiones anteriores de JDK, cuando el sistema necesita redibujar un componente, se invoca a su método **paint** o **update**. Sin embargo, lo que se proporciona es un objeto **Graphics2D**, que extiende a **Graphics** proporcionando acceso a las nuevas funcionalidades de Java 2D.

Además, todos aquellos componentes de Swing derivados de **JComponent** implementan internamente el doble buffer, por lo que no tendremos que ocuparnos de hacerlo nosotros. Simplemente deberemos rellenar el código del método **paint**, borrando el contenido del contexto gráfico que se nos proporciona, si fuese necesario, y dibujando los gráficos. Es importante hacer notar que no todos los componentes de Swing derivan de **JComponent**, como es el caso de **JFrame** por ejemplo, por lo que en este caso no implementará internamente el doble buffer. Para saber si un componente deriva de éste, simplemente tendremos que ir a la documentación de la API de Java y en la página de dicho componente veremos toda la jerarquía de sus ascendientes.

Lo que deberemos hacer (siempre que trabajemos con la versión 1.2 de JDK o posteriores) será hacer una conversión cast del objeto **Graphics** proporcionado a un objeto **Graphics2D**. Esto es así porque a los métodos **paint** y **update** se les estará proporcionando en realidad un objeto **Graphics2D**, aunque la referencia a él sea de tipo **Graphics** por cuestión de mantener la compatibilidad con versiones anteriores. Si queremos utilizar las funcionalidades mejoradas que ofrece la nueva API de Java 2D, deberemos obtener el objeto **Graphics2D** de la siguiente forma:

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

Este objeto nos permitirá realizar un mayor número de operaciones en el contexto gráfico pudiendo así obtener de forma sencilla unos gráficos mejorados.

En este nuevo contexto gráfico tendremos los siguiente atributos:

- *Paint*: Relleno de las figuras o texto dibujadas. En este caso el relleno no tiene por qué ser un color sólido. También podrá ser un determinado patrón o un gradiente.

Figura 4. Relleno de las figuras

- *Font*: Fuente utilizada para dibujar texto.
- *Stroke*: Contorno de las figuras o texto dibujadas (lápiz). Podremos utilizar para el contorno lineas continuas o discontinuas, y distintos grosores de línea.

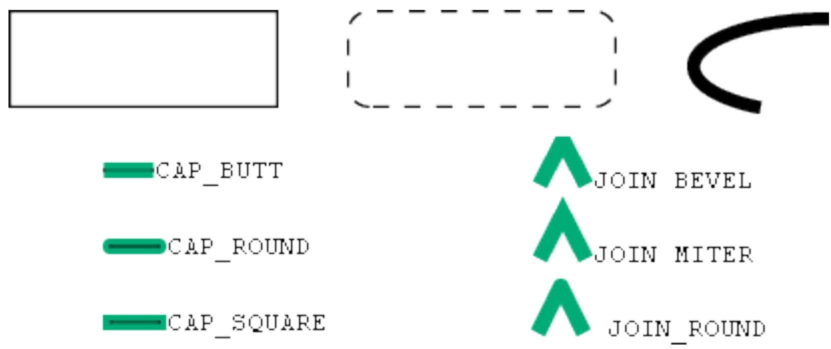


Figura 5. Tipos de lápìces

- *Transform*: Podremos realizar transformaciones con las figuras, texto o imágenes que dibujemos. Podremos utilizar aquí cualquier transformación afín, por lo que podremos realizar traslaciones, rotaciones, escalados, desencajados, en resumen, cualquier transformación que pueda ser definida por una matriz de transformación 3x3.
- *Composite*: Se refiere a la forma de combinar las distintas figuras que dibujemos. Podremos hacer que al dibujar una figura sobre otra se produzcan diferentes efectos, como quedarse encima o debajo, o bien producirse una intersección, unión o resta de figuras.

Figura 6. Composición de figuras

- *Clip*: Realiza un recortado del contenido que dibujemos. Nos permite definir unos límites del área en la que queremos que se dibuje. Todo lo que quede fuera de este área no se dibujará en pantalla. El área de recortado no se limita sólo a rectángulos en este caso.
- *Calidad del render (RenderingHints)*: Aquí podremos cambiar determinadas opciones de render. Por ejemplo, podremos usar antialiasing para reducir el efecto escalera de los gráficos.

Figura 7. Efecto aliasing

Se proporcionan una seríe de métodos **setXXXX** y **getXXXX** para obtener y modificar los atributos anteriores.

Para dibujar figuras podemos utilizar el siguiente método:

```
g2.draw(Shape figura);
```

La información de las figuras estará encapsulada en clases derivadas de **Shape** (podemos encontrar rectángulos, líneas, curvas, etc). Deberemos instanciar la figura adecuada y dibujarla en el contexto gráfico mediante el método anterior.

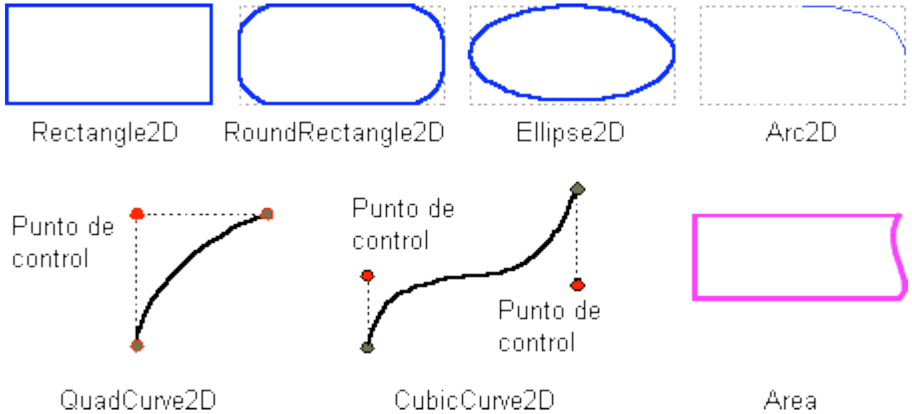


Figura 8. Tipos de figuras

3.4.5 Modo a pantalla completa

A partir de JDK 1.4.0 podremos utilizar un modo de gráficos a pantalla completa. Con este modo tendremos acceso exclusivo y directo al dispositivo de la pantalla.

En AWT hemos visto anteriormente que era el sistema el que se encargaba de mandar eventos para repintar el contenido de nuestra ventana. Esto tiene que hacerse así ya que el dispositivo gráfico (la pantalla) se comparte con el

sistema operativo y el resto de aplicaciones en ejecución, por lo que es el mismo sistema el que nos tiene que indicar cuando podemos redibujar.

Sin embargo, al tener ahora un acceso exclusivo a la pantalla, podremos dibujar en ella en cualquier momento. Esto es lo que se conoce como render activo, frente al render pasivo que hemos visto anteriormente.

Para utilizar este modo exclusivo a pantalla completa lo primero que deberemos hacer es obtener el dispositivo gráfico (**GraphicsDevice**) que vamos a utilizar. Lo obtendremos a partir del entorno gráfico (**GraphicsEnvironment**) de nuestra máquina:

```
GraphicsEnvironment ge =
        GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice gd = ge.getDefaultScreenDevice();
```

Podemos tener varios dispositivos gráficos en nuestra máquina, que podrán ser obtenidos también a través del entorno gráfico. Con el método visto anteriormente obtendremos el dispositivo primario que es el que querremos utilizar normalmente, ya que lo normal será disponer de un solo dispositivo gráfico.

Una vez tenemos el dispositivo gráfico podemos comprobar si soporta el modo a pantalla completa con el siguiente método (dentro de *GraphicsDevice*):

```
boolean b =  gd.isFullScreenSupported()
```

Una vez hemos comprobado que soporta dicho modo podremos pasar a pantalla completa indicando que la ventana (**Window**) de nuestra aplicación vamos a mostrar en la pantalla. Para ello utilizamos el siguiente método (dentro de *GraphicsDevice*):

```
gd.setFullScreenWindow(Window w)
```

Del objeto de dispositivo gráfico podremos obtener la lista de modos soportados por dicho dispositivo (resolución y profundidad de color), y seleccionar cualquiera de estos modos siempre que la operación esté permitida en nuestro sistema. Obtenemos el modo gráfico actual con:

```
DisplayMode dm = gd.getDisplayMode();
```

Para obtener todos los modos soportados haremos lo siguiente:

```
DisplayMode [] dms = gd.getDisplayModes();
```

Los datos sobre los modos gráficos se encapsulan en la clase **DisplayMode**. Una vez elegido el modo que queremos utilizar podemos seleccionarlo con:

```
gd.setDisplayMode(dm);
```

Antes hemos hablado del concepto de render activo. Para utilizar este tipo de render tendremos que obtener el objeto de contexto gráfico asociado al componente donde vamos a dibujar (la ventana en este caso):

```
Graphics g = getGraphics();
```

Una vez hemos terminado de dibujar en la pantalla, debemos llamar a **dispose** para liberar los recursos que estuviese utilizando.

```
g.dispose();
```

Podemos usar tanto el modo activo como el modo pasivo cuando trabajemos a imagen completa. Sin embargo el modo activo nos dará un mayor control y podremos utilizar técnicas avanzada como el intercambio (*flipping*) de páginas de memoria de video.

Antes hemos visto como implementar el doble buffer. Pero ahora que tenemos acceso directo al dispositivo gráfico, podremos usar una técnica más rápida para implementarlo. Existen dos técnicas:

BLT: Leido como *blit* (*blitting*), que significa *BLock Transfer*. Es una técnica similar a la que hemos descrito anteriormente. El backbuffer es un área en memoria, y cuando queremos volcar este contenido a pantalla copiaremos el contenido de dicho área a la memoria del dispositivo gráfico.

FLIP: Se refiere al intercambio (*flipping*) de páginas de memoria de video. Normalmente la tarjeta gráfica tendrá varias páginas de memoria (doble buffer o triple buffer normalmente). En un momento dado el dispositivo gráfico estará mostrando el contenido de una de estas páginas. Lo que haremos será utilizar la página que no se esté mostrando como backbuffer. Una vez hayamos terminado de dibujar, haremos que esa página pase a ser el buffer de pantalla mostrándose así su contenido sin tener que hacer ninguna transferencia de datos, y la otra página pasará a ser el backbuffer donde dibujaremos a continuación. Esta técnica, puesto que la página contiene todo el contenido de la pantalla, no podremos utilizarla si estamos compartiendo la pantalla con otras aplicaciones, ya que afectariamos a todas ellas. Es por esto que esta técnica solo podemos utilizarla al trabajar a pantalla completa.

Con el modo a pantalla completa podremos utilizar cualquiera de las dos técnicas anteriores. Para ello tenemos las clases **BufferStrategies** y **BufferCapabilities** que implementan dichas técnicas.

EJEMPLO

La siguiente clase es un ejemplo sencillo de cómo establecer la pantalla completa y dibujar animaciones en ella.

El constructor de la clase principal se encarga de obtener el dispositivo gráfico (*GraphicsDevice*), elegir el modo gráfico (*DisplayMode*), y luego pasar a modo

pantalla completa con el modo gráfico seleccionado. Después hay un hilo (campo *t*) que es el que se encarga de hacer las animaciones.

Para las animaciones, definimos un método *update* que sólo llame a *paint*, y un método *paint* que vaya dibujando cada frame de la animación, pero con un doble buffer. El método *dibuja* se encarga de dibujar en el *backbuffer*, y luego en *paint* se vuelca ese contenido directamente en pantalla. La animación en sí (método *dibuja*) consiste en mover un círculo rojo desde una X inicial de 10 hasta una final de 200.

Veréis también una clase interna llamada *DlgModos* que se encarga de elegir el modo gráfico (*DisplayMode*) de entre una lista de posibles modos. Simplemente muestra un cuadro de diálogo para que el usuario elija el modo que quiera. Luego, el modo elegido es asignado al dispositivo gráfico (*GraphicsDevice*).

Este otro ejemplo es un juego completo hecho utilizando estas técnicas. Fue desarrollado por Miguel Angel Lozano, profesor de este departamento, y es una variante del juego Pang que antiguamente se solía ver en las máquinas recreativas. Podéis probarlo ejecutando la clase *Panj*.

3.4.6 Sonido y música. Java Sound

Hemos visto como incluir gráficos y animaciones en nuestras aplicaciones y applets. Con Java además podremos añadir sonido y música de forma sencilla.

En las primeras versiones de JDK los applets permitían cargar clips de audio, con lo que podiamos reproducir música y sonido en la web. Los tipos de ficheros y formatos reconocidos entonces eran bastante limitados (MIDI, WAV, AU).

A partir de Java 2, se incorpora la API de Java Sound, que nos permite además de tratar con una mayor número de formatos, incorporar sonido tanto a applets como aplicaciones y trabajar con la reproducción del sonido a un nivel más bajo, lo cual nos dará mayor flexibilidad para incluir todo tipo de efectos de sonido en nuestras aplicaciones.

Nos referiremos a las músicas y a los efectos de sonido como clips de audio. Estos clips de audio estarán encapsulados en la clase **AudioClip**. Para cargar un clip de audio, los applets siempre han incorporado el siguiente método:

```
AudioClip clip = getAudioClip(url);
```

Para ello deberemos estar dentro un applet, ya que dicho método pertenece a la clase **Applet** y para usarlo debe haber un objeto **Applet** instanciado.

Si estamos dentro de una aplicación, no tendremos acceso a este método. Por ello, a partir de JDK 1.2 se incorpora a la clase **Applet** el siguiente método estático:

```
AudioClip clip = Applet.newAudioClip(url);
```

Al ser estático no hará falta haber instanciado un applet para poder usarlo, por lo que lo podremos utilizar desde cualquier aplicación Java. El problema que tenemos en este caso es que debemos proporcionar una URL, cuando lo más seguro es que queramos cargar un fichero del disco local. Podemos obtener una url para esto de la siguiente forma:

```
URL url = new URL("file:" + System.getProperty("user.dir")
                         + "/sonido.wav";
```

Utilizamos la propiedad del sistema *user.dir* para obtener el directorio actual, y con él podemos construir un URL para el acceso a ficheros locales.

Una vez obtenido el clip de audio, podremos reproducirlo.

```
clip.play();
```

Reproduce el clip una sola vez.

```
clip.loop();
```

Reproduce el clip ciclicamente. Util en el caso de que queramos tener una música de fondo que no pare de sonar.

```
clip.stop();
```

Detiene la reproducción del clip de audio. Sobretodo útil para músicas, y cuando hayamos establecido que se reproduzca ciclicamente.

En la API Java Sound incorporada en las últimas versiones de Java, tenemos bastante más clases que nos permitirán controlar la secuenciación, el mezclado, etc. Estas clases se incluyen en el paquete **javax.sound** y subpaquetes.