FORMACIÓN Y TECNOLOGÍAS JAVA
UNIVERSIDAD DE ALICANTE

Curso de Programación en Lenguaje Java

Sesiones de Ejercicios

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

Sesión 1 5
Sesión 2 13
Sesión 3 21
Sesión 4 27
Sesión 5 35
Sesión 6 39
Sesión 7 45
Sesión 8 51
Sesión 9 59
Sesión 10 65
Sesión 11 73
Sesión 12 81
Sesión 13 89
Sesión 14 95
Sesión 15 101

Sesión 1

Compilación. Paquetes. Hola Mundo.

1. Instalación del JDK 1.4

En este primer ejercicio vamos a realizar la instalación del JDK, necesario para compilar y ejecutar programas Java.

1. Instala el SDK (Software Developer Kit) Java 1.4.2 (se encuentra en la página de recursos). Es el conjunto de herramientas que Sun proporciona para desarrollar en Java.
2. Comprueba que la instalación ha funcionado correctamente, ejecutando el compilador de Java:

```
> C:\j2sdk1.4.2_02\bin\javac
```

3. Actualiza la variable de entorno `PATH`, añadiendo el directorio `C:\j2sdk1.2.4_02\bin` para que no tengas que teclear todo el camino cada vez que quieras ejecutar algún programa de Java. Para obtener información más detallada sobre cómo actualizar la variable de entorno PATH, puedes consultar el apéndice B de los apuntes del curso.
4. Comprueba que todo funciona bien, cambiándote a cualquier directorio (por ejemplo `c:\tmp` y escribiendo:

```
C:\tmp> javac
```

2. Mis primeros programas Java

En este ejercicio vamos a realizar, compilar y ejecutar dos programas Java. En uno de ellos habrá algún error y deberás corregirlo.

1. Antes de empezar a programar, lee el apartado **1.1** (*Compilación y ejecución de programas Java*) de los apuntes.
2. Crea el directorio `c:\java`, este va a ser el directorio de trabajo donde vas a guardar los programas ejemplo de esta primera sesión. Escribe el siguiente programa Java en un fichero de texto con un editor de texto cualquiera, como el *Notepad*. Graba el programa con el nombre `Ejemplo1.java` (es importante la "E" mayúscula).

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

Compila la clase y verás como se genera el fichero `.class` que contiene los bytecodes. Este fichero es el que después debe ejecutar la máquina virtual java (JVM):

```
C:\java> javac Ejemplo1.java
C:\java> dir

. <DIR> 16/06/04 1:59a .
.. <DIR> 16/06/04 1:59a ..
EJEMPL~1 CLA 433 16/06/04 2:07a Ejemplo1.class
EJEMPL~1 JAV 130 16/06/04 1:59a Ejemplo1.java
```

Ejecuta el programa con la JVM:

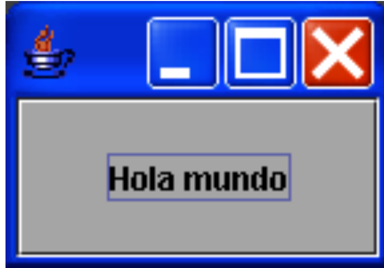
```
C:\java>java Ejemplo1
Hola mundo
```

Para ejecutar un programa Java, debes invocar con la JVM una clase que contenga un método estático llamado main; este es el método que se ejecutará.

3. Escribe ahora el siguiente programa, y sálvalo con el nombre de `Ejemplo2.java`.

```
import java.awt.*;
import java.awt.event.ActionEvent;
import javax.swing.*;

public class MiPrimerGUI {
    public static void main(String[] args) {
      // Defino la accion a ejecutar
      Action action = new AbstractAction("Hola mundo") {
           public void actionPerformed(ActionEvent evt) {
                System.out.println("Hola mundo");
             }
        };

        JButton button = new JButton(action);
        JFrame frame = new JFrame();

        // Añado el boton al marco
        frame.getContentPane().add(button,
BorderLayout.CENTER);

        // Dimensiones del marco
        int frameWidth = 100;
        int frameHeight = 100;
        frame.setSize(frameWidth, frameHeight);

        // Muestro el marco
        frame.setVisible(true);

        // Le digo al frame que salga de la aplicacion
        // cuando se cierre

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Compila el programa y dará un mensaje de error. ¿Cuál es el error? ¿Cómo corregirlo? (contestalo en el fichero **respuestas.txt**). Corrige el error y ejecuta la clase. Deberá aparecer la siguiente ventana, y cada vez que pinches en el botón debería aparecer "Hola mundo" en la salida estándar.



No intentes entender el código. Esto es sólo un aperitivo de lo que vendrá en el último módulo del curso.

3. Trabajando con paquetes

Vamos ahora a realizar unos cortos ejemplos para introducir el uso de paquetes en las clases Java.

1. Vamos ahora a actualizar la variable CLASSPATH. Seguimos en el directorio `C:\java`, en el que se encuentra la clase compilada `Ejemplo1.class`. Si actualizamos el CLASSPATH (recuerda que es la variable del entorno que le indica al compilador y al intérprete Java dónde buscar las clases) de la siguiente forma

   ```
   set CLASSPATH=C:\
   ```

   y vuelves a probar a ejecutar el programa ejemplo

   ```
   C:\java> java Ejemplo1
   <MENSAJE DE ERROR>
   ```

   veras que aparece un mensaje de error. Para que funcione correctamente debes volver a actualizar el CLASSPATH

   ```
   C:\java> set CLASSPATH=.
   C:\java> java Ejemplo1
   Hola mundo
   ```

   ¿Qué ha causado el error? ¿Por qué se ha arreglado? ¿Qué pasará si actualizas el CLASSPATH de la siguiente forma: "`set CLASSPATH = .;C:\java`" ? (contesta en el fichero **respuestas.txt**).

2. Vamos a declarar que la clase `Ejemplo1` se defina en el paquete `modulo1.sesion1`. Para eso debemos cambiar el código fuente de la siguiente forma.

   ```
   package modulo1.sesion1;

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

Compila el programa, crea la estructura de directorios `modulo1\sesion1` y coloca allí la nueva clase compilada. Puedes hacerlo con:

```
C:\java> javac Ejemplo1.java
C:\java> mkdir modulo1
C:\java> cd modulo1
C:\java\modulo1> mkdir sesion1
C:\java\modulo1> cd ..
C:\java> copy Ejemplo1.class modulo1\sesion1
C:\java> del Ejemplo1.class
```

Una forma más directa de hacer lo mismo es usando la directiva `-d` del compilador de Java. Esta directiva permite indicarle al compilador un directorio en el que vamos a dejar los ficheros `.class` generados. Si los ficheros `.class` están en un package, el compilador se encargará de crear los directorios necesarios:

```
C:\java> javac Ejemplo1.java -d .
```

Por último, para ejecutar el programa recién compilado hay que llamar a la clase indicando todo su camino (`modulo1.sesion1.Ejemplo1`). La variable CLASSPATH debe contener el directorio padre y podemos llamar al programa desde cualquier directorio.

```
C:\java>set CLASSPATH=.;C:\java
C:\java>cd ..
C:\> java modulo1.sesion1.Ejemplo1
Hola mundo
```

3. Por último, vamos a crear un fichero JAR en el que se incluya la clase `Ejemplo1` y el paquete en el que reside:

```
C:\java> jar cvf ejemplo.jar modulo1
manifest agregado
agregando: modulo1/(entrada = 0) (salida= 0)(almacenado
0%)
agregando: modulo1/sesion1/(entrada = 0) (salida=
0)(almacenado 0%)
agregando: modulo1/sesion1/Ejemplo1.class(entrada = 442)
(salida= 305)(desinflado 30%)
```

Actualizamos el CLASSPATH con el camino del fichero JAR y ya podemos llamar al intérprete

```
C:\java> set CLASSPATH=C:\java\ejemplo.jar
C:\java> java modulo1.sesion1.Ejemplo1
Hola mundo
```

4. ¿Qué valor deberías poner en el CLASSPATH para poder usar la clase `misclases.utils.Robot` que tiene siguiente PATH en el sistema operativo: `C:\java\lib\misclases\utils\Robot.class`? ¿Y si la clase se encuentra en el mismo paquete dentro del fichero JAR `misclases.jar`,

que tiene el PATH: `C:\java\lib\misclases.jar`? (contesta en el fichero **respuestas.txt**)

4. El juego de los números

Vamos ahora a un ejercicio en el que compilarás un programa algo más completo.

1. Vamos a ver un ejemplo algo más completo, en el que se utilizan objetos y clases definidas por el usuario. Escribe y compila la siguiente clase, es una clase con un único método (`startGame`). Lo siguiente es una imagen, por lo que tendrás que teclearlo todo. Corrige los errores (seguro que tendrás más de uno al copiar!) hasta que los únicos errores pendientes sean los derivados de que la clase `Player` no está definida (6 errores).

```
public class GuessGame {
    Player p1;
    Player p2;
    Player p3;

    public void startGame() {
        p1 = new Player();
        p2 = new Player();
        p3 = new Player();

        int guessP1 = 0;
        int guessP2 = 0;
        int guessP3 = 0;

        boolean p1IsRight = false;
        boolean p2IsRight = false;
        boolean p3IsRight = false;

        System.out.println("Estoy pensando un numero entre 0 y 9 ...");
        int targetNumber = (int) (Math.random() * 10);
        System.out.println("El numero a adivinar es: " + targetNumber);

        while (true) {

            p1.guess();
            p2.guess();
            p3.guess();

            guessP1 = p1.number;
            System.out.println("El jugador 1 dice: " + guessP1);
            guessP2 = p2.number;
            System.out.println("El jugador 2 dice: " + guessP2);
            guessP3 = p3.number;
            System.out.println("El jugador 3 dice: " + guessP3);

            if (guessP1 == targetNumber)
                p1IsRight = true;
            if (guessP2 == targetNumber)
                p2IsRight = true;
            if (guessP3 == targetNumber)
                p3IsRight = true;

            if (p1IsRight || p2IsRight || p3IsRight) {
                System.out.println("Tenemos un ganador!");
                System.out.println("El jugador 1 ha hacertado?: " + p1IsRight);
                System.out.println("El jugador 2 ha hacertado?: " + p2IsRight);
                System.out.println("El jugador 3 ha hacertado?: " + p3IsRight);
                break;
            } else {
                System.out.println("Los jugadores lo intentan otra vez");
            }
        }
    }

}
```

2. Escribe y compila la clase `Player` (en el directorio actual) para que el programa funcione correctamente.

   Como verás, ninguna clase tiene el método `main`, por lo que no se puede lanzar el programa; escribe la clase `GameLauncher`, con el método main que lance el programa. El resultado debe ser algo como:

```
C:\plj\modulo1>java GameLauncher
Estoy pensando un numero entre 0 y 9 ...
El numero a adivinar es: 2
Yo digo: 0
Yo digo: 6
Yo digo: 2
El jugador 1 dice: 0
El jugador 2 dice: 6
El jugador 3 dice: 2
Tenemos un ganador!
El jugador 1 ha hacertado?: false
El jugador 2 ha hacertado?: false
El jugador 3 ha hacertado?: true
```

5. Un último programa

Por último, vas a escribir un programa completo a partir de un ejemplo.

1. Veamos un último programa, que escribe en la salida estándar los argumentos que se le pasan, separados por dos puntos ":".

```
public class Echo {
    public static void main(String[] args) {
        int i=0;
        while (i < args.length){
            System.out.print(args[i]);
            System.out.print(":");
            i++;
        }
        System.out.println();
    }
}
```

2. Escribe un programa `Reverse` que escriba en la salida estándar los argumentos que se le pasa al intérprete Java, pero invertidos y separados por dos puntos ":".

```
C:\java>java Reverse Hola que tal
tal:que:Hola
```

3. Escribe un último programa `Reverse2` que invierta también los caracteres de cadaa palabra. Para ello, puedes acceder al carácter i-ésimo de una palabra usando el método `charAt` de la clase `String`. Por ejemplo `str.charAt(0)` devuelve el carácter 0 (el primero) del objeto `String` que está en la variable `str`. También necesitas saber la longitud de una palabra. Puedes obtener la longitud de un `String str`, llamando al método `length` así: `str.length()`.

```
C:\java>java Reverse2 Hola que tal
lat:euq:aloH:
```

PARA ENTREGAR

Debes crear un ZIP llamado **sesion1.zip** con:

- El fichero de texto **respuestas.txt** con las cuestiones contestadas
- Los ficheros **Ejemplo1.java**, **Ejemplo2.java**, **GuessGame.java**, **Player.java**, **GameLauncher.java, Reverse.java** y **Reverse2.java**.

¡No lo entregues todavía! Debes entregar todas las sesiones del primer módulo juntas.

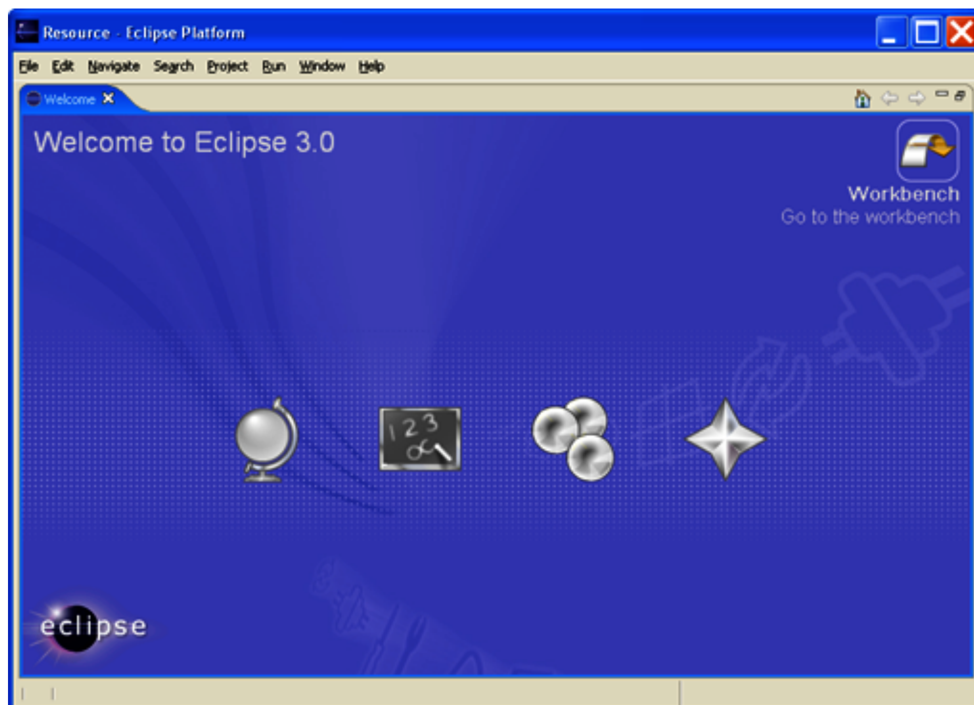
Sesión 2

Eclipse. Paquetes. Bean Shell.

1. Instalación de Eclipse

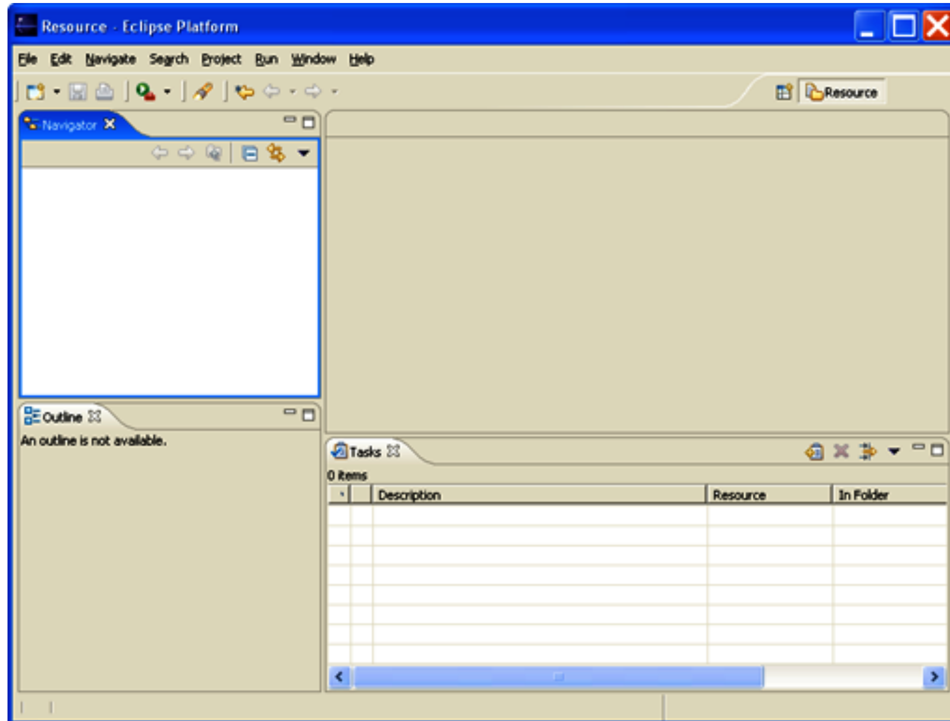
En este primer ejercicio vamos a instalar el entorno de desarrollo **Eclipse 3.0**

1. Descomprime el fichero Eclipse (ver la página de recursos) correspondiente a tu sistema operativo en algún directorio del sistema. El fichero es un archivo ZIP que contiene todos sus ficheros bajo el directorio `./eclipse`. Por ejemplo, en Windows descomprímelo en `C:\` y en Linux lo puedes descomprimir en `/usr/local/` (si tienes permiso de super usuario; si no, lo puedes descomprimir en `/home/<user>/`). En Mac OS X debes descomprimir el fichero y arrastrar la aplicación Eclipse a la carpeta de aplicaciones.
2. Arranca Eclipse haciendo doble click sobre la aplicación. Es un programa escrito en Java y debes tener instalado el JDK o el JRE (ya lo has hecho en la sesión 1) para que funcione. Cuando Eclipse arranca por primera vez pide el directorio de trabajo. Puedes aceptar el que te sugiere (`C:\eclipse\workspace`), o indicarle alguno propio.
3. Cuando aceptes el directorio de trabajo, aparecerá la siguiente ventana de presentación.



Desde esta ventana de presentación puedes ver tutoriales, ejemplos o entrar directactemente en la zona de trabajo (esquina superior derecha). Haz esto último para comenzar a trabajar en Eclipse. Podrás volver a la

ventana de presentación en cualquier momento con la opción *Help -> Welcome*). Quédate por ahora en la zona de trabajo:



4. Lee el apartado 1.4 de los apuntes (*Eclipse: un entorno gráfico para desarrollo Java*), en especial los apartados 1.4.2 (*Configuración visual: perspectivas, vistas y editores*) y 1.4.5 (*Proyectos Java*). Las pantallas de ese apartado están tomadas de una versión anterior de Eclipse y son algo diferentes al aspecto de la versión 3.0. Puede ser que en esta primera lectura haya cosas que no entiendas; no te preocupe demasiado, se refieren a aspectos avanzados que podrás usar más adelante.

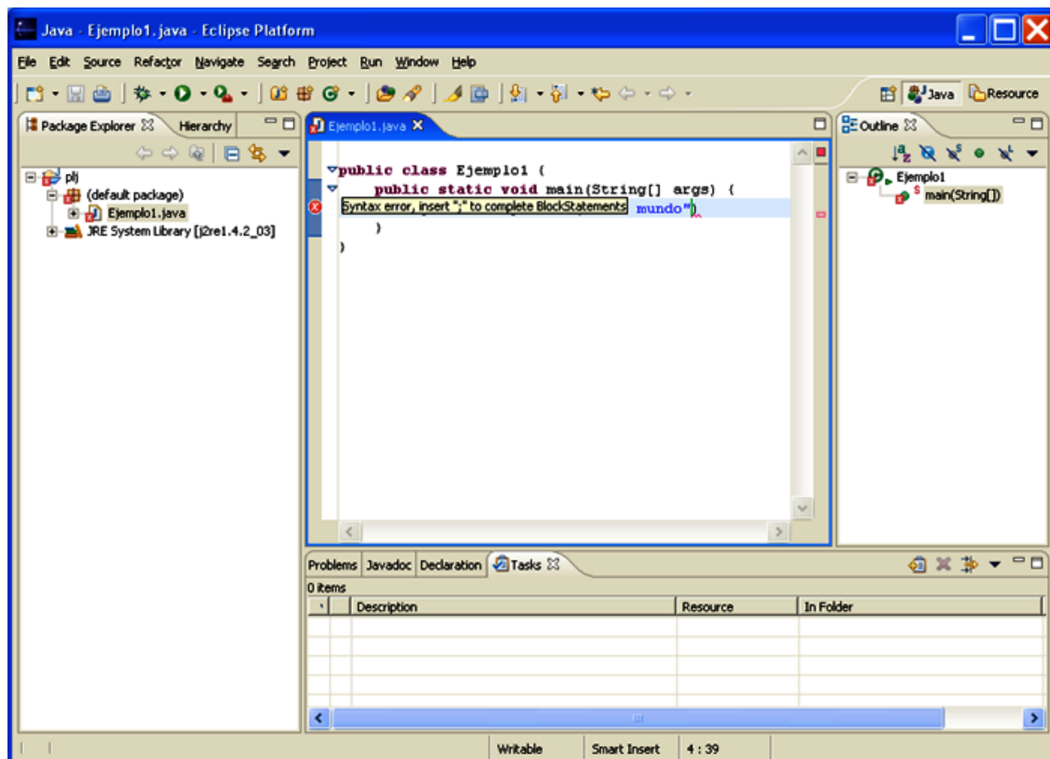
2. Las primeras clases Java en Eclipse

Vamos a crear un proyecto en Eclipse en el que programaremos todas las clases de la sesión de ayer, para que compruebes la diferencia entre usar este entorno y programar con un editor básico y la línea de comando.

1. Crea un proyecto Java nuevo con la opción *File -> New -> Project*. Llámalo `plj` y acepta todas las opciones por defecto. Verás que Eclipse te pide pasar a la perspectiva Java. Acepta. El proyecto se corresponde con un directorio que se ha creado con el mismo nombre en el directorio de trabajo de Eclipse. **Puedes usar este proyecto durante todo el curso**.
   En el proyecto pueden residir todo un conjunto de clases y paquetes; esto es, en un proyecto puede haber más de un programa ejecutable

(recuerda que los programas Java son clases que tienen el método `main`).

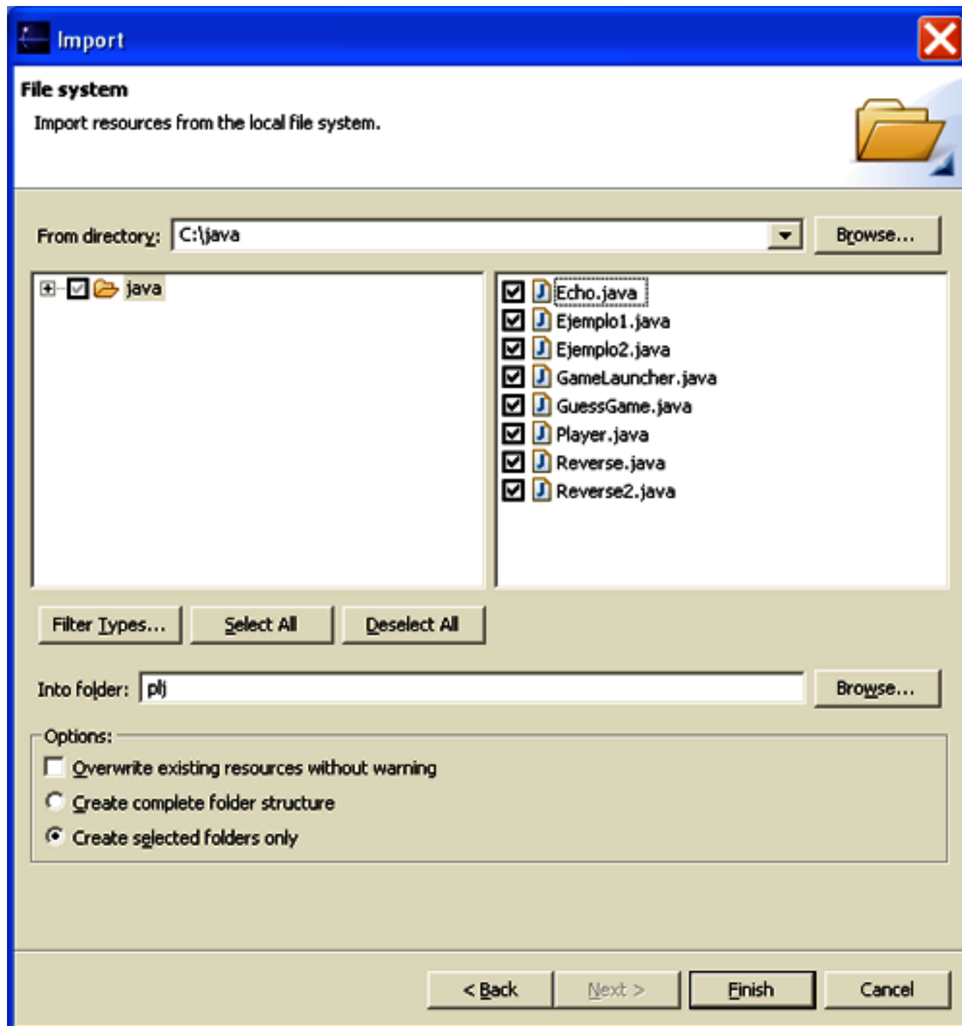
2. Vamos a crear la clase `Ejemplo1` con el *"Hola mundo"*. Selecciona el proyecto y, con el botón derecho, escoge la opción *New -> Class*. Dale el nombre `Ejemplo1` y activa la opción para que cree un esqueleto del método "`public static void main (String[] args)`". Pulsa en *Finish* y aparecerá el editor de Eclipse con el fichero de clase. Al crear la clase, Eclipse introduce comentarios con tareas por hacer que puedes ver en la vista *Tasks*. Puedes borrar estos comentarios y se borran automáticamente las tareas. Escribe en este fichero el programa Ejemplo1 de la sesión de ayer. Durante la escritura podrás comprobar que el editor chequea la sintaxis cuando grabas el fichero, indicándote si hay un error, qué tipo de error es, e incluso sugiriéndote la posible solución:



Cuando salves el programa correctamente escrito verás que desaparecen todas las marcas de errores.

Si examinas en el sistema operativo el directorio `plj` creado en el espacio de trabajo de Eclipse, verás que se encuentra el fichero `Ejemplo1.java` y el fichero compilado `Ejemplo1.class`. Eclipse sólo ha añadido los ficheros `.project` y `.classpath` que definen algunas constantes del proyecto. Esta es una de las muchas ventajas de Eclipse frente a otros entornos: su limpieza. Lo que hay en la ventana de proyecto es lo que hay en el directorio del sistema operativo. Puedes examinar los ficheros del sistema operativo cambiando a la perspectiva *Resource*.

3. Vamos a ejecutar la clase `Ejemplo1`. Selecciona la clase y, con el botón derecho, escoge la opción *Run -> Java Application* (también puedes seleccionar en el menú la opción *Run -> Run As -> Java Application*). Aparecerá la vista *Console* con la salida de la ejecución del programa.
4. Escribe el resto de clases de la sesión 1, excepto la que está en el package `misclases.utils`. Para ello puedes importar las clases en el paquete desde el sistema de ficheros. Escoge la opción *File -> Import -> File system*, selecciona el directorio `c:\java` y marca todas las clases que quieres importar (todas en este caso):



Todas las clases se copiarán al proyecto actual y se compilarán automáticamente.

5. ¿Cómo lanzar un programa con argumentos en la línea de comandos? Para esto es necesario crear una **configuración de ejecución**, un elemento muy útil de Eclipse. Selecciona en el menú la opción *Run -> Run ...* . Aparecerá la ventana de gestión de configuraciones de ejecución, en la que podrás crear y guardar con un nombre una configuración de ejecución. Dale a esta configuración el nombre `conf1`,

selecciona como *Main class* la clase `Reverse2` y dale los valores que quieras a los argumentos del programa. Puedes guardar la configuración con la opción *Apply* y ejecutarla con *Run*. La configuración queda guardada y puedes lanzarla cuando quieras, por ejemplo después de realizar modificaciones en el programa principal.

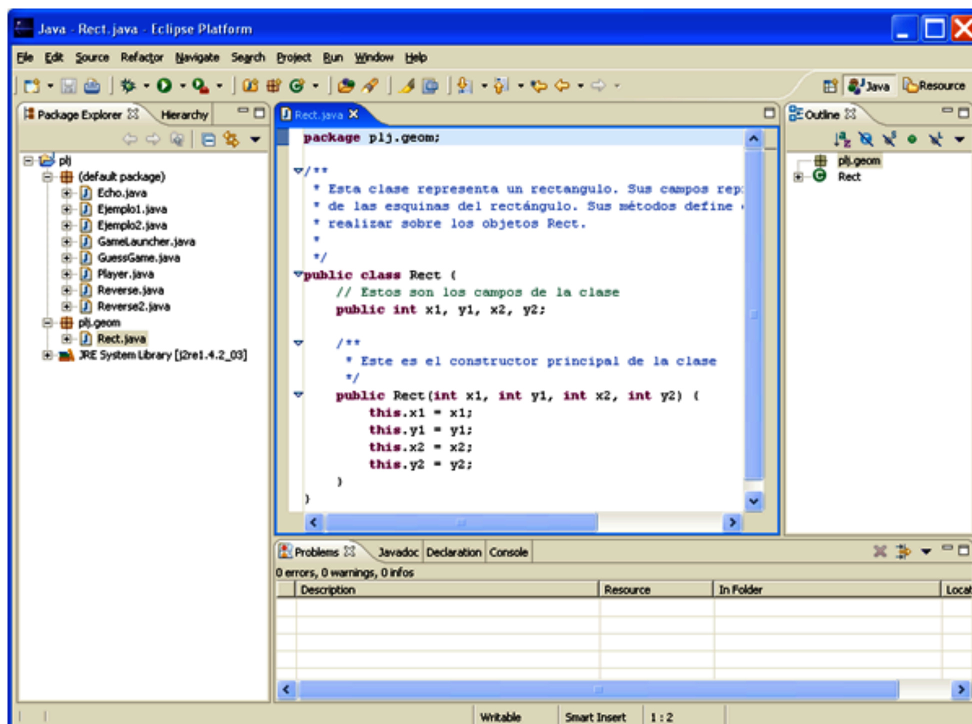
3. La clase plj.geom.Rect

En este ejercicio vamos a compilar en Eclipse la clase `Rect`, que define un rectángulo, dentro del *package* `plj.geom`.

1. Comenzamos creando el *package* `plj.geom`. Para ello, escogemos la opción del menú *File-> New-> Package*. Y creamos el paquete con el nombre `plj.geom`. Inmediatamente aparecerá en la vista *Package Explorer*. Eclipse también habrá creado en el sistema de ficheros los directorios `plj/geom` dentro del directorio de proyecto `plj`.
2. Desempaqueta el fichero de plantillas de esta sesión de ejercicios e importa en el package recién creado las clases `Rect.java` y `Position.java`. Vamos a centrarnos en la clase `Rect`, la clase `Position` la usuaremos en otra sesión. Se trata de una clase con la que se definen objetos de tipo rectángulo. Mira el código fuente de la clase. Verás que, por estar definida en el package `plj.geom`, se declara la línea

```
package plj.geom;
```

al comienzo del fichero. En Eclipse tendremos una configuración similar a esta:



3. Escribe una clase de prueba que use la clase `Rect` y que se llame **`PruebaRect.java`**, en el que se realicen varias operaciones con dos

rectángulos recién creados y se compruebe la clase `Rect`. Escríbela fuera del package `plj.geom`. Debe tener el siguiente esqueleto:

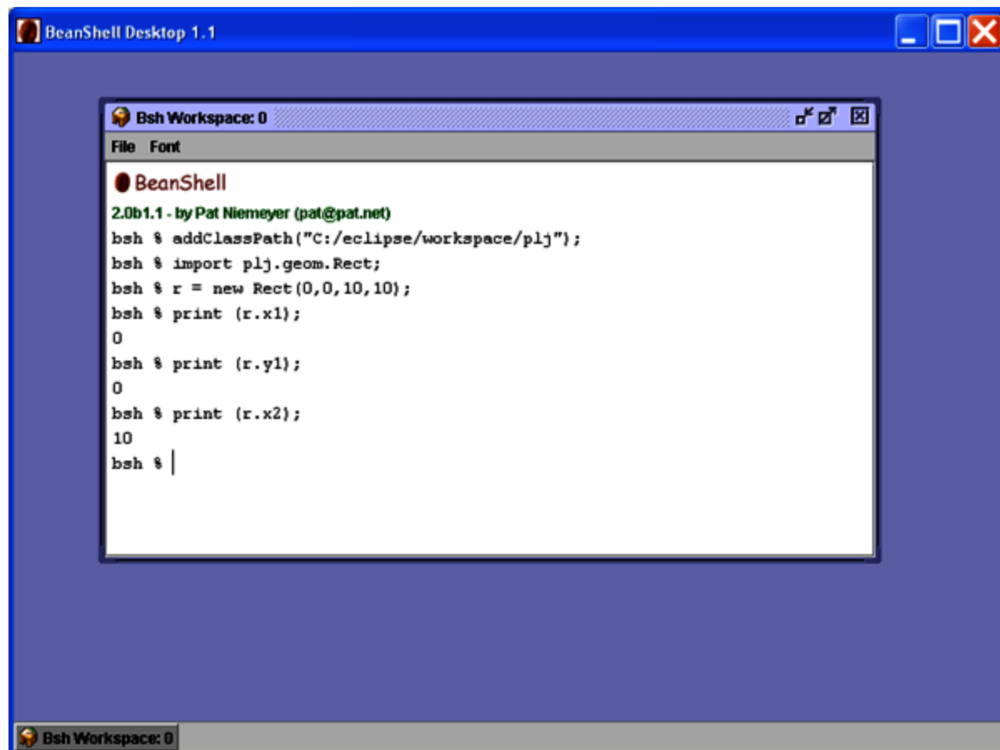
```
import plj.geom.Rect;

public class PruebaRect {

      public static void main(String[] args) {
              Rect r1, r2, r3;
        // operaciones con los rectángulos
      }
}
```

4. Por último, crea el paquete **`modulo1.sesion2`** y mueve todas las clases a ese nuevo paquete. Lo puedes hacer arrastrando las clases al nuevo paquete y Eclipse se encarga de declarar el paquete en el código fuente de las clases. Prueba alguna clase para comprobar que no ha habido ningún error.

4. El intérprete BeanShell

Vamos a terminar probando una herramiente muy útil: un intérprete interactivo de Java. El intérprete se llama BeanShell y puede ejecutar código Java de forma interactiva.

1. Descomprime BeanShell en cualquier directorio del sistema de ficheros. Se trata de un programa Java que necesita que tengas JDK o JRE instalado para que funcione. Lánzalo haciendo un doble click sobre el fichero `bsh-2.0b1.jar`. Aparecerá un entorno como el que se muestra en la siguiente figura. El Workspace es un intérprete interactivo en el que puedes ejecutar instrucciones Java. Vamos a probar en ese intérprete la clase `Rect`.



2. Para añadir al CLASSPATH de BeanShell el directorio donde se encuentra el paquete `plj` se usa la instrucción `addClassPath("C:/eclipse/workspace/plj")`. Para imprimir un valor a hay que usar la instrucción `print (a)`. Un ejemplo de ejecución es el siguiente:

```
bsh % addClassPath("G:/eclipse/workspace/plj");
bsh % import plj.geom.Rect;
bsh % Rect r = new Rect(0,0,10,10);
bsh % print (r.toString());
[0.0,0.0; 10.0,10.0]
bsh % r.move(2.0,4.0);
bsh % print (r);   // llama implícitamente a
r.toString()
[2.0,4.0; 12.0,14.0]
bsh %
```

Ejecuta en el intérprete las mismas instrucciones que escribiste en el programa `PruebaRect.java`. Copia todas las instrucciones y las respuestas de BeanShell en el fichero **respuestas.txt**.

El intérprete BeanShell es muy completo y tiene funcionalidades que escapan totalmente de este curso. Puedes consultar su funcionamiento en la documentación que se entrega en los recursos. Una instrucción muy útil es

```
    bsh % reloadClasses();
```

que permite volver a cargar las clases que están en el classpath de BeanShell. Esta instrucción es útil cuando hemos modificado una clase y

queremos comprobar el funcionamiento en BeanShell. Notar que las variables usadas previamente no pueden ser usadas con la nueva clase.

PARA ENTREGAR

Exporta todo el proyecto con Eclipse a un fichero ZIP llamado **eclipse.zip** usando la opción del menú *File -> Export -> ZIP File*.

Escribe en el fichero **respuestas.txt** qué impresión te han causado Eclipse y BeanShell. Crea, por último, un fichero ZIP llamado **sesion2.zip** con el ficheros **respuestas.txt** y el fichero **eclipse.zip**.

Sesión 3

API de Java. Métodos y objetos. Modificador static. Herencia.

1. El API de Java

Vamos a comenzar con un ejercicio para aprender a consultar el API de Java (y algunas cosas más).

1. Repasa la sección *API de Java* del apartado 1.1.1 de los apuntes.
2. Busca en el API de Java el paquete `java.util.zip`. Consultando la página HTML que describe el paquete, contesta en el fichero **respuestas.txt** las siguientes preguntas:
   - ¿Para qué es el paquete?
   - ¿Qué interfaces, clases y excepciones se declaran en el paquete?
3. Busca en el API la clase `Stack`. Contesta en el fichero **respuestas.txt** las siguientes preguntas:
   - ¿En qué paquete se encuentra la clase `Stack`? ¿Qué instrucción `import` tendrías que definir para usar la clase `Stack`?
   - ¿Qué constructores tiene la clase?
   - ¿Qué métodos modifican el estado de un objeto `Stack`?
   - ¿De qué clase son los objetos que puedes añadir y obtener de un stack?
   - El siguiente código no es correcto, ¿por qué? (consulta el apartado 1.7.3 de los apuntes) ¿Cuál sería el código correcto?

     ```
     int i = 10; Stack pila = new Stack(); pila.add(i);
     ```

     Nota: en la versión 1.5 de Java y en BeanShell el código de arriba es correcto ya que usan una técnica denominada autoboxing.

4. Ejecuta en BeanShell un conjunto instrucciones que trabajen con un `Stack`. Recuerda que para visualizar el valor de un objeto `o` en BeanShell debes usar el comando `print(o)`. Copia la sesión de interacción con BeanShell en el fichero **respuestas.txt**.

2. Métodos y objetos

Vamos a hacer un ejercicio más elaborado en el que se definan clases y objetos. Puedes usar el entorno de desarrollo que desees. Si tu ordenador no tiene la potencia (memoria sobre todo) suficiente para que Eclipse funcione con fluidez usa algún editor de texto para editar las clases.

Todas las clases las vamos a definir en el paquete `modulo1.sesion3`. Se encuentran en la plantilla de ejercicios de la sesión 3.

1. Supongamos las siguientes clases `Contador` y `ContadorTest`

Fichero "Contador.java":

```
package modulo1.sesion3;
public class Contador {
     static int acumulador = 0;
     int valor;

     static public int acumulador() {
         return acumulador;
     }

     public Contador(int valor) {
         this.valor = valor;
         acumulador += valor;
         // no es valido this.acumulador
         // es valido Contador.acumulador
     }

     public void inc() {
         valor++;
         acumulador++;
     }

     public int getValor(){
         return valor;
     }
}


package modulo1.sesion3;  public class ContadorTest {
public static void main(String[] args) {
Contador c1, c2;
System.out.println(Contador.acumulador());
c1 = new Contador(3);         c2 = new Contador(10);
c1.inc();          c1.inc();          c2.inc();
System.out.println(c1.getValor());
System.out.println(c2.getValor());
System.out.println(Contador.acumulador);             } }
```

Fichero "ContadorTest.java":

```
package modulo1.sesion3;
public class ContadorTest {
      public static void main(String[] args) {
         Contador c1, c2;
         System.out.println(Contador.acumulador());
         c1 = new Contador(3);
         c2 = new Contador(10);
         c1.inc();
         c1.inc();
         c2.inc();
         System.out.println(c1.getValor());
         System.out.println(c2.getValor());
         System.out.println(Contador.acumulador);
       }
}
```

Compila las clases y pruébalas. Responde a las siguientes preguntas en el fichero **respuestas.txt**:

- ¿Se pueden realizar las siguientes modificaciones en el código de la clase `Contador`? ¿Por qué?
  1. Cambiar "acumulador += valor" en el constructor Contador por "this.acumulador += valor".
  2. Cambiar "acumulador += valor" en el constructor Contador por "Contador.acumulador += valor".
  3. Cambiar "valor++" por "this.valor++" en el método inc().
- ¿Qué valores imprime el programa ContadorTest?
- Si cambiamos en la clase Contador la línea "static int acumulador = 0" por "private static int acumulador = 0", ¿aparece algún error? ¿por qué?
- ¿Qué sucede si no inicializamos el valor del campo acumulador?

2. Vamos a complicar un poco más el código de Contador, añadiendo una constante (`VALOR_INICIAL`) a la clase y otro nuevo constructor. El código es el que sigue (en negrita lo que se ha añadido). El modificador final indica que el valor asignado a `VALOR_INICIAL` no puede modificarse.

```
package modulo1.sesion3;

public class Contador {
     static int acumulador;
     final static int VALOR_INICIAL=10;
     int valor;

    static public int acumulador() {
         return acumulador;
    }

    public Contador(int valor) {
       this.valor = valor;
       acumulador += valor;
    }

    public Contador(){
        this(Contador.VALOR_INICIAL);
    }

    public void inc() {
        this.valor++;
        acumulador++;
    }

    public int getValor(){
        return this.valor;
     }
}
```

Fíjate en la llamada "`this(Contador.VALOR_INICIAL)`". ¿Qué hace? Escribe un programa ejemplo `ContadorTest2` que compruebe el funcionamiento de la clase modificada. Por último, una pregunta algo complicada: ¿Qué sucede si cambiamos la línea

"`this(Contador.VALOR_INICIAL)`" por "`new Contador(Contador.VALOR_INICIAL)`"?

3. Por último, realiza las siguientes modificaciones en la clase `Contador`:
   - Añade una variable de clase `nContadores` que contenga el número de contadores creados
   - Añade una variable de clase `valores` que contenga un array con los valores de los contadores creados.
   - Añade un metodo `getValores` que devuelva un array con los valores de los contadores creados.

3. Un ejemplo de herencia

1. En el fichero de plantillas de esta sesión encontrarás los ficheros `Figura.java`, `Circulo.java`, `Rectangulo.java` y `FiguraTest.java`, que contienen el siguiente código:

```
package modulo1.sesion3;

public abstract class Figura {
     double area;
     public abstract double getArea();
}

package modulo1.sesion3;

class Circulo extends Figura {
      double radio;

     public Circulo(double radio) {
         this.radio = radio;
     }

     public double getArea() {
         area = 3.14 * (radio * radio);
         return (area);
     }
}

package modulo1.sesion3;
class Rectangulo extends Figura {
     double lado;
     double altura;

     public Rectangulo(double lado, double altura) {
         this.lado = lado;
         this.altura = altura;
     }

     public double getArea() {
         area = lado * altura;
         return (area);
     }
}
```

```
package modulo1.sesion3;
import java.util.Stack;

public class FiguraTest {
     public static void main(String args[]) {
        Figura f;
        Rectangulo rect = new Rectangulo(10, 15);
        System.out.println("El area del rectangulo es: "
                      + rect.getArea());
        Circulo circ = new Circulo(3);
        System.out.println("El area del circulo es: "
                      + circ.getArea());

        Stack pila = new Stack();
        pila.push(rect);
        pila.push(circ);
        while (!pila.isEmpty()) {
          f = (Figura) pila.pop();
          System.out.println("El area de la figura es: "
                  + f.getArea());
         }
     }
}
```

2. Compila las clases y ejecuta el ejemplo. Contesta las siguientes preguntas en el fichero **respuestas.txt**:
   - ¿Podría crear un objeto de tipo `Figura`? ¿Por qué?
   - ¿Por qué puedo añadir a la pila objetos de tipo `Rectangulo` y `Circulo`, siendo tipos diferentes?
   - Supongamos un método como el siguiente

     ```
     public void draw(Figura fig);
     ```

     _Podría pasar un objeto de la clase `Circulo` como parámetro del método? ¿Y un objeto de la clase `Rectangulo`? ¿Y un objeto de la clase `plj.geom.Rect`?

   - ¿Cómo se deberían modificar las clases para que el método `getArea()` no fuera abstracto, sino que tuviera la implementación definida en la clase `Figura` (y funcione correctamente, claro).

4. Crea una jerarquía de clases

Define e implementa un ejemplo de jerarquía de clases. Recuerda algunas reglas básicas sobre herencia de clases:

- Una subclase A hereda de una superclase B si pasa el **test ES-UN**. Este test consiste, sencillamente, en comprobar que A **ES-UN** B. Por ejemplo, un Sonar **ES-UN** Sensor. O también, un Perro **ES-UN** Animal. O también, un Empleado **ES-UNA** Persona. Pero un Animal no **ES-UN** Naturaleza (por lo que Animal no puede ser subclase de Naturaleza).

- Construye una subclase sólo cuando necesites hacer una versión **más específica** de una clase y necesites sobrecargar o añadir nuevas conductas.
- Usa una clase abstracta cuando quieras definir una **plantilla** para para un grupo de subclases, y tengas algún código de implementación que todas las clases puedan usar. Haz la clase abstracta cuando quieras garantizar que nadie va a hacer objetos de esa clase.

PARA ENTREGAR

Debes crear un ZIP llamado **sesion3.zip** con:

- El fichero de texto **respuestas.txt** con las cuestiones contestadas.
- Los ficheros con las clases que has implementado y modificado.

Sesión 4

Interfaces. Arrays. Colecciones.

1. El API de Java (cont.)

Vamos a comenzar de nuevo con un ejercicio sobre el API de Java.

1. Busca en el API de Java el paquete `java.util`. Consultando las páginas HTML que describen el paquete, contesta en el fichero **respuestas.txt** las siguientes preguntas:
   - ¿Qué interfaces se declaran en el paquete?
   - ¿Qué excepciones?
2. Busca en el paquete la interfaz `Iterator`.
   - ¿Qué métodos se declaran en la interfaz `Iterator`?
   - ¿Qué métodos de las clases de `java.util` devuelven un `Iterator`? (¡No se te ocurra buscarlos uno a uno!, hay una forma mucho más fácil).

2. Un ejemplo de interfaces

1. Descomprime el fichero de plantillas del ejercicio. Encontrarás las clases `Watcher`, `Sensor`, `Sonar` y `Bumper`. La clase `Sensor` es una clase abstracta de la que heredan `Sonar` y `Bumper`. La clase `Sensor` implementa la interfaz `Watcher`. Suponemos que todas estas clases se están definiendo para implementar un programa Java que simula el movimiento de un robot móvil en un mapa predefinido. El robot tendrá objetos de tipo `Sonar` y `Bumper` (y otros) que se usarán para tomar lecturas del entorno simulado en el que se mueve.

```
package modulo1.sesion4;

import plj.geom.*;

/**
 * La interfaz Watcher especifica una posición y una
 * dirección en la que se mira. Ambos datos están
referidos
 * a un sistema de referencia (mapa) global.
 */
public interface Watcher {
    public Position getPos();
    public double angle();
}

package modulo1.sesion4;

public abstract class Sensor implements Watcher {
    int state;
    private long time;

    public abstract void doReadings();
```

```
    public int getState() {
        return state;
    }

    public void fire() {
        time = System.currentTimeMillis();
        doReadings();
    }

    public long lastTimeFired() {
        return time;
    }
}

package modulo1.sesion4;

import plj.geom.Position;

public class Sonar extends Sensor {
    double[] readings;
    int resolution;    // numero de lecturas
    int rangeMax;      // maximo alcance
    double angleMax;   // arco del sonar

    public  Sonar(int  resolution,  int  rangeMax,  double
angleMax){
        this.resolution = resolution;
        this.rangeMax = rangeMax;
        this.angleMax = angleMax;
    }

    public double[] getReadings() {
        return readings;
    }

    /**
     * doReadings() actualiza el array readings
     * con todas las lecturas de distancias
     * de objetos frente al sonar en la direccion
     *  angle()  y  en  un  arco  y  una  resolucion  definida
por
     * el tipo de sonar
     */
    public void doReadings() {
    }

    public Position getPos() {
        return null;
    }

    public double angle() {
        return 0;
    }
}

package modulo1.sesion4;

import plj.geom.Position;
```

```
public class Bumper extends Sensor {
    boolean bumped;

    /**
     * doReadings() actualiza bumped.
     * true -> si ha habido un contacto
     * false -> si el sensor no ha chocado con nada
     */
    public void doReadings() {
    }

    public boolean bumped() {
        return bumped;
    }

    public Position getPos() {
        return null;
    }

    public double angle() {
        return 0;
    }
}
```

2. Compíla las clases. Contesta a las siguientes preguntas en el fichero **respuestas.txt**:
   - ¿Se pueden crear objetos de tipo `Sensor`?
   - ¿Qué sucede si no definimos el método `angle()` en la clase `Bumper`?¿Por qué?
   - ¿Cómo habría que modificar las clases si añadimos el método `move()` en la interfaz `Watcher`?
   - ¿Podríamos definir la clase `Sensor` como concreta (no abstracta)? ¿Habría que añadirle algún método? ¿Habría que quitarle algún método? ¿Podría `doReadings()` seguir siendo un método abstracto?
3. Escribe un programa de ejemplo que use esas clases.

2. Implementando interfaces

1. Define e implementa un ejemplo de interfaz y de clase o clases que lo implementen. Recuerda algunas reglas básicas sobre clases e interfaces:
   - Una subclase A hereda de una superclase B si pasa el **test ES-UN**. Este test consiste, sencillamente, en comprobar que A **ES-UN** B. Por ejemplo, un Sonar **ES-UN** Sensor. O también, un Perro **ES-UN** Animal. O también, un Empleado **ES-UNA** Persona. Pero un Animal no **ES-UN** Naturaleza (por lo que Animal no puede ser subclase de Naturaleza).
   - Construye una subclase sólo cuando necesites hacer una versión **más específica** de una clase y necesites sobrecargar o añadir nuevas conductas.
   - Usa una clase abstracta cuando quieras definir una **plantilla** para para un grupo de subclases, y tengas algún código de implementación que todas las clases puedan usar. Haz la clase

abstracta cuando quieras garantizar que nadie va a hacer objetos de esa clase.

- Usa un interface cuando quieras definir un **papel** que puedan jugar otras clases, independientemente de dónde se encuentran esas clases en el árbol de herencia. Recuerda que la definición de interfaces es la forma en la que Java resuelve la imposibilidad de usar herencia múltiple. **Una subclase no puede tener más de una superclase, sin embargo sí que puede implementar más de una interface.**

3. Arrays

Vamos a hacer un pequeño ejemplo en el que manejemos arrays. También vamos a usar la entrada estándar para introducir valores a un programa Java.

1. Prueba el siguiente programa. Es un programa que define un array estático y después lo ordena. Usa el método `sort` para ordenar arrays estáticos que se define en la clase `java.util.Arrays`.

```
package modulo1.sesion4;

import java.util.Arrays;

public class ArrayDemo1 {
    public static void main(String args[]) {
        int vec[] = { 37, 47, 23, -5, 19, 56 };
        Arrays.sort(vec);
        for (int i = 0; i < vec.length; i++) {
            System.out.println(vec[i]);
        }
    }
}
```

   Cambia el tipo de array a un array de `Strings`, inicializa el array con valores de cadenas arbitrarias y prueba si funciona el programa. ¿Cómo piensas que puede funcionar correctamente, sin añadir ningún parámetro al método `sort()` que le diga el tipo de elementos que hay en el array? (Contesta en el fichero **respuestas.txt**).

2. Prueba el siguiente programa, que realiza una búsqueda binaria en un array

```
package modulo1.sesion4;
import java.util.Arrays;

public class ArrayDemo2 {
    public static void main(String args[]) {
        int vec[] = {-5, 19, 23, 37, 47, 56};
        int slot = Arrays.binarySearch(vec, 35);
        slot = -(slot + 1);
        System.out.println("Punto de inserion = " +
slot);
    }
}
```

El programa busca en el array el número 35 y no lo encuentra. Por eso el método `binarySearch` devuelve un número negativo. El número -(`slot` + 1) es el lugar donde se encontraría el número en el caso de estar en el array.

3. Prueba el siguiente programa, que muestra cómo pedir datos al usuario usando la entrada estándar. El programa espera números doubles y usa el método estático `parseDouble()` de la clase `Double` para convertir la cadena que escribe el usuario en un `double`.

```
package modulo1.sesion4;

import java.io.*;

public class StandardInput {
    public static void main(String[] args) throws
IOException {
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));

        for (;;) {
            System.out.print("StandardInput> ");
            String line = in.readLine();
            // El programa termina cuando llegamos al
fin del
            // fichero o el usuario introduce quit
            if ((line == null) || line.equals("quit"))
                break;
            // Parsea la linea y calcula la raiz
cuadrada
            try {
                double x = Double.parseDouble(line);
                System.out.println("raiz cuadrada de " +
x + " = "
                        + Math.sqrt(x));
            }
            // Si algo falla muestra un mensaje de error
            catch (Exception e) {
                System.out.println("Entrada erronea");
            }
        }
    }
}
```

4. Termina juntando todos los programas de este punto: escribe un programa `ArrayDemo3.java` que pida al usuario números enteros hasta que escriba la cadena "fin", introduzca los números en un array y después pida al usuario un número a buscar en el array. El programa debe terminar dando un mensaje que diga en qué posición se encuentra el número o el mensaje "número no encontrado" si el número no está en el array.

4. ArrayLists

Vamos a ver un pequeño ejemplo que use el framework Collections. En el apartado 1.7 de los apuntes hay una explicación detallada de las clases e interfaces de este framework. El uso de interfaces es muy importante en las colecciones.

1. Compila y prueba el siguiente programa:

```
package modulo1.sesion4;

import java.util.*;

public class CollectionDemo1 {

    public static void main(String args[]) {
        // Creo una coleccion de objetos
        List list = new ArrayList();

        // Anyado elementos a la colección
        list.add("This is a String");
        list.add(new Short((short) 12));
        list.add(new Integer(35));

        // Obtengo un iterador de la lista y lo recorro
        for (Iterator i = list.iterator(); i.hasNext();)
{
            System.out.print(i.next());
        }
    }
}
```

   Contesta las siguientes preguntas en el fichero **respuestas.txt**.

   - ¿Qué es `List`, una clase abstracta o un interfaz? ¿Sería correcto hacer la siguiente llamada?

     ```
     List miLista = new List();
     ```

   - ¿Funcionaría bien el programa si se sustituye new `ArrayList()` por new `LinkedList()` ?¿Qué cambiaría?
   - ¿Qué clases concretas pueden tener los objetos que se pasen cómo parámetro a un método definido de la siguiente forma?

     ```
     public void metodoQueUsaList(List unaLista)
     ```

2. Si necesitas que la colección de elementos esté ordenada, debes usar la clase `TreeSet`, que implementa la interfaz `SortedSet`. Prueba el siguiente ejemplo:

```
package modulo1.sesion4;

import java.util.*;

public class CollectionDemo2 {
```

```
    public static void main(String args[]) {
        // Creo una coleccion de objetos
        List list = new ArrayList();

        // Anyado elementos a la colección
        list.add(new Integer(50));
        list.add(new Integer(-1));
        list.add(new Integer(35));

        // Creo un TreeSet a partir de la lista

        TreeSet set = new TreeSet(list);

        // Anyado otros elementos al conjunto

        set.add(new Integer(99));
        set.add(new Integer(-5));

        // Obtengo un iterador del conjunto y lo recorro
        for (Iterator i = set.iterator(); i.hasNext();)
{
            System.out.print(i.next()+" ");
        }
    }
}
```

3. Por último, de forma similar al ejercicio anterior, escribe un programa `CollectionDemo3` que pida al usuario números enteros hasta que escriba la cadena "fin", introduzca los números en un conjunto y después pida al usuario un número a buscar en el conjunto. El programa debe terminar dando un mensaje que diga si el número introducido está en el conjunto o no.
Nota 1: usa el API de la interfaz SortedSet para encontrar los métodos con los que comprobar si el número está en el conjunto.
Nota 2: nota que en las colecciones deben introducirse objetos y no datos primitivos. Lee el apartado 1.7.3 de los apuntes si quieres más información sobre los wrappers de los tipos básicos.

PARA ENTREGAR

Debes crear un ZIP llamado **sesion4.zip** con:

- El fichero de texto **respuestas.txt** con las cuestiones contestadas
- Los ficheros con las clases que has implementado y modificado.

Sesión 5

Modificadores de acceso. Miniproyecto.

1. Modificadores de acceso

Vamos a comenzar la última sesión del módulo con un pequeño ejercicio para comprobar los modificadores de acceso de Java.

Antes de nada, hagamos un pequeño resumen. En Java existen cuatro posibles niveles de acceso: `private`, por defecto (si no declaramos nada), `protected`, `public`. Estos cuatro niveles tienen la siguiente política de acceso:

- **`private`**: no se permite el acceso al elemento, ni siquiera para los objetos de alguna subclase. **Sólo se puede acceder al elemento desde la misma clase**. Si, por ejemplo, declaramos un método A de una superclase Super como private, ese método A no es heredado por las subclases de Super.
- **defecto**: es el nivel de acceso que tiene un elemento si no declaramos nada. **Se puede acceder al elmento desde cualquier clase del mismo paquete**. Si, por ejemplo, el campo A de la clase Clase1 que está en el paquete modulo1.sesion5 no tiene modificador de acceso (tiene un acceso por defecto), cualquier clase de este paquete va a poder acceder a su valor.
- **`protected`**: puede ser que una subclase no esté en el mismo paquete que la superclase. Si un elemento tiene el modificador protected, **se puede acceder a él desde el mismo paquete y desde cualquier subclase, aunque la subclase no esté en el mismo paquete**.
- **`public`**: un elemento public **es accesible desde cualquier otra clase** sin ninguna restricción.

1. Vamos a hacer un pequeño ejercicio para probar estos niveles de acceso. Supongamos la siguiente clase en el paquete `modulo1.sesion5`

```
package modulo1.sesion5;
public class Acceso {
    public int valorPublico;
    int valorDefecto;
    protected int valorProtected;
    private int valorPrivate;
}
```

   y ahora supongamos las dos siguientes clases que van a comprobar el acceso a los campos de Acceso:

```
package modulo1.sesion5;
public class TestAcceso{
    public void testeador() {
        int i;
```

```
        Acceso acceso = new Acceso();
        i = acceso.valorPrivado;
        i = acceso.valorDefecto;
        i = acceso.valorProtected;
        i = acceso.valorPublico;
    }
}
package modulo1.sesion5;
public class TestAccesoSubclase extends Acceso{
    public void testeador() {
        int i;

        i = this.valorPrivado;
        i = this.valorDefecto;
        i = this.valorProtected;
        i = this.valorPublico;
    }
}
```

La primera clase es una clase normal que está en el mismo paquete y la segunda es una subclase de Acceso. Contesta a las siguientes preguntas en el fichero **respuestas.txt**:

- ¿En qué campos de la clase `TestAcceso` hay un error?
- ¿En qué campos de la clase `TestAccesoSubclase` hay un error?

2. Copia ahora ambas clases de prueba en el paquete `modulo1.sesion4`, modificando la instrucción package y añadiendo el import de la clase `modulo1.sesion5.Acceso`:

```
package modulo1.sesion4;
import modulo1.sesion5.Acceso;
```

¿Qué ha cambiado ahora? ¿Qué componentes son accesibles?

2. Miniproyecto

En el fichero de plantillas de esta sesión se encuentran la siguientes clases que implementan un sencillo juego de un laberinto: `Posicion`, `Laberinto Jugador`, `EstadoJuego`, `Vista`, `Controlador`, `JuegoLaberinto` y `JuegoLauncher`.

Vamos primero a explicar brevemente el funcionamiento del juego y después comentaremos el ejercicio a realizar.

Funcionamiento del juego

El juego define un laberinto (siempre el mismo) en el que se mueve el jugador. La posición inicial del jugador es aleatoria. El jugador se mueve introduciendo un comando de texto. El laberinto y el jugador aparecen representado en modo texto y se muestra por la salida estándar cada vez que el jugador ha introducido un comando:

```
# #############
#    *  #     #
# ### # # ### #
```

```
#   # #   ### #
### ########  #
#   #   ####  #
# ### ###### ##
# #    ##### ##
#   ##     # ##
############ ##

INTRODUCE MOVIMIENTO (A,B,D,I,S)>
```

Diseño e Implementación

El juego se ha diseñado siguiendo la estrategia (o el patrón, como se suele decir) Modelo-Vista-Controlador. La idea de esta estrategia es independizar el funcionamiento del programa de su presentación al usuario. De esta forma, es más sencillo modificar la interfaz de usuario (la forma de obtener los datos del usuario y de presentar los resultados) sin afectar al funcionamiento del programa.

En nuestro caso, tenemos la clase Vista que se encarga de mostrar el laberinto y de obtener la instrucción del usuario. Tenemos también la clase Controlador que se encarga de realizar un paso de ejecución del juego. Y, por último, tenemos las clases Jugador, Laberinto y EstadoJuego que mantienen los distintos elementos del juego.

Si, por ejemplo, quisiéramos adaptar el juego a un entorno gráfico, sólo tendríamos que modificar la clase `Vista` y esto no afectaría al funcionamiento interno del juego.

Vamos ya a plantear los ejercicios:

1. Compila todas las clases en el paquete `modulo1.sesion5` y prueba que funciona la aplicación.
2. Modifica la aplicación para incluir las siguientes características:
    - Incorporar un jugador enemigo (lo llamaremos robot) que se mueve aleatoriamente por el laberinto.
    - Si chocan en la misma celda el robot y el jugador, terminar el juego con un mensaje de error. Por ejemplo:"Fin de partida. Has chocado".
    - Si el robot sale del laberinto, terminar el juego con un mensaje. Por ejemplo: "Fin de partida. Ha ganado el robot".

PARA ENTREGAR

Debes crear un ZIP llamado **sesion5.zip** con:

- El fichero de texto **respuestas.txt** con las cuestiones contestadas
- Los ficheros con las clases que has implementado y modificado.

Sesión 6

1. Vamos a construir un ejercicio completo de uso de excepciones. Para ello seguiremos los pasos que se indican a continuación:

1. Antes de comenzar, lee la introducción del punto **2.1** (*Excepciones*), y los apartados **2.1.1** (*Tipos de excepciones*) y **2.1.2** (*Captura de excepciones*) del tema 2 de teoría.
2. Echa un vistazo a la clase *Ej1.java* que se proporciona en la plantilla. Verás que calcula la división de dos números (en el método *divide*), y se ejecuta dicha división en el método *main*, pasándole los dos números como argumentos en el parámetro *args* de dicho método.
3. Compila y prueba el funcionamiento de la clase, ejecutando una división sencilla, como por ejemplo 20 / 4, de la siguiente forma:

```
javac Ej1.java
java Ej1 20 4
```

4. Observa que la clase no realiza ciertos controles: no se comprueba si se le han pasado parámetros al *main* al ejecutar, ni si dichos parámetros son números correctos, ni si se puede realizar la división (no se podría dividir por un número que fuese cero, por ejemplo). Vamos a ir controlando dichos errores mediante lanzamiento y captura de excepciones, en los siguientes pasos.
5. Dentro del método *main* vamos a comprobar que se hayan pasado 2 parámetros al programa, y que dichos dos parámetros sean dos números.
   a. Para lo primero (comprobar que se han pasado 2 parámetros), capturaremos una excepción de tipo **ArrayIndexOutOfBoundsException**, al acceder al parámetro *args*:

```
public static void main(String[] args)
{
    String param1=null, param2=null;
    int dividendo=0, divisor=0;

    try
    {
            param1 = args[0];
            param2 = args[1];
    } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println ("Faltan parametros");
            System.exit(-1);
    }

    ...

    System.out.println ("Result.: ...);
}
```

Prueba después la excepción capturada, ejecutando el programa con algo como:

```
java Ej1 20
```

Debería capturar la excepción y mostrar por pantalla el mensaje "*Faltan parametros*".

b. Para lo segundo (comprobar que los dos parámetros son números), capturaremos una excepción de tipo **NumberFormatException** si falla el método de conversión a entero de los parámetros de tipo *String* (es decir, capturaremos la excepción al llamar a *Integer.parseInt(...)*):

```
public static void main(String[] args)
{
    String param1=null, param2=null;
    int dividendo=0, divisor=0;

    ...

    try
    {
            dividendo = Integer.parseInt(param1);
            divisor = Integer.parseInt(param2);
    } catch (NumberFormatException e2) {
            System.out.println ("...");
            System.exit(-1);
    }

    System.out.println ("Resultado: ...);
}
```

Prueba después la excepción capturada, ejecutando el programa con algo como:

```
java Ej1 20 hola
```

Debería capturar la excepción y mostrar por pantalla el mensaje "*Formato incorrecto del parametro*".

c. Observa que se puede poner todo junto, ahorrándonos las variables *param1* y *param2*, y capturando las dos excepciones en un solo bloque:

```
public static void main(String[] args)
{
    int dividendo=0, divisor=0;

    try
    {
            dividendo = Integer.parseInt(args[0]);
            divisor = Integer.parseInt(args[1]);
    } catch (ArrayIndexOutOfBoundsException e) {
            ...
```

```
    } catch (NumberFormatException e2) {
            ...
    }

    System.out.println ("Resultado: ...);
}
```

Modifica el método *main* para dejarlo todo en un solo bloque *try*, y vuelve a probar los distintos casos de fallo indicados antes, para ver que el programa sigue comportándose igual.

6. Lee ahora el punto **2.1.3** (*Lanzamiento de excepciones*) del tema 2 de teoría.
7. Una vez tenemos comprobado que se pasan 2 parámetros, y que éstos son numéricos, sólo nos falta comprobar que se puede realizar una división correcta, es decir, que no se va a dividir por cero. Eso lo vamos a comprobar dentro del método *divide.*
   a. Al principio del método, comprobamos si el segundo parámetro del mismo (el divisor) es cero, si lo es, lanzaremos una excepción indicando que el parámetro no es correcto. Dentro de los subtipos de excepciones de **RuntimeException**, tenemos una llamada **IllegalArgumentException** que nos puede ayudar. Probamos a poner estas líneas al principio del método *divide*:

```
public static int divide(int dividendo, int divisor)
{
    if (divisor == 0)
            throw new IllegalArgumentException ("...");

    ...
}
```

   b. Compila y ejecuta la clase. Prueba con algo como:

```
java Ej1 20 0
```

   ¿Qué mensaje aparece? ¿Qué significa?

   c. Notar que lanzamos la excepción en el método *divide* pero no la capturamos en *main*. Por eso nos muestra un mensaje de error más largo, con la traza de la excepción. Si quisiéramos controlar qué saca el error, deberíamos capturar la excepción en el *main* y mostrar el texto que quisiéramos.

   Capturemos la excepción en el *main* y mostremos la traza del error:

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

    try
    {
```

```
        System.out.println ("Resultado:...);
    } catch (IllegalArgumentException e3) {
            e3.printStackTrace()
    }
}
```

¿Qué mensaje aparece? ¿Qué significa?

8. Lee ahora el punto **2.1.4** (*Creación de nuevas excepciones*) del tema 2 de teoría.
9. Vamos a añadir una última excepción al método *divide* para comprobar que se realiza una división de números naturales (es decir, enteros mayores que 0). Para ello vamos a crear una excepción propia, llamada **NumeroNaturalException**, como la siguiente:

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

10. ¿Para qué sirve la llamada a *super* en el constructor en este caso?
11. Ahora añadiremos el lanzamiento de esta excepción en el método *divide*, comprobando, tras ver si el divisor es cero, que tanto dividendo como divisor son positivos:

```
public static int divide(int dividendo, int divisor)
{
  if (divisor == 0)
          throw new IllegalArgumentException ("...");
  if (dividendo < 0 || divisor < 0)
          throw new NumeroNaturalException("...");

  ...
}
```

   a. Prueba a compilar la clase. ¿Qué error te da? ¿A qué puede deberse?
   b. Para subsanarlo, hay que indicar en el método *divide* que dicho método puede lanzar excepciones de tipo **NumeroNaturalException**. Eso se hace mediante una cláusula *throws* en la declaración del método:

```
public static nt divide(int dividendo, int divisor)
throws NumeroNaturalException
{
    if(divisor == 0)
      throw new IllegalArgumentException ("...");
    if (dividendo < 0 || divisor < 0)
      throw new NumeroNaturalException("...");

      ...
}
```

NOTA: esta cláusula sólo hay que introducirla para excepciones que sean de tipo *checked*, es decir, que se pueda predecir que pueden pasar al ejecutar un programa. Dichas excepciones son los subtipos fuera de **RuntimeException**, y cualquier excepción que podamos crear nosotros.

c. Prueba a compilar la clase. ¿Qué error te da ahora? ¿Por qué?
d. Lo que nos queda por hacer para terminar de corregir la clase es capturar la excepción que se lanza en el método *divide,* cuando utilizamos dicho método en el *main*:

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

   try
   {
      System.out.println ("Resultado: " ...);
   } catch (IllegalArgumentException e3) {
      e3.printStackTrace()
   } catch (NumeroNaturalException e4) {
      System.out.println ("Error: " + e4.getMessage());
      System.exit(-1);
   }
}
```

¿Para qué sirve la llamada a "getMessage()" (qué texto obtenemos con esa llamada)?
Nota que cuando hemos añadido antes la excepción **IllegalArgumentException** no ha habido que poner una cláusula "throws" en la declaración de "divide", ni capturar la excepción en el *main* para poder compilar, y sin embargo sí lo hemos hecho ahora para la **NumeroNaturalException**. ¿A qué se debe esta diferencia?

12. Finalmente, compila y ejecuta el programa de las siguientes formas, observando que da la salida adecuada:

```
java Ej1 20 4            // 5
java Ej1 20              // "Faltan parametros"
java Ej1 20 hola         // "Formato incorrecto del parametro"
java Ej1 20 0            //  Excepción  de  tipo  IllegalArgument
java Ej1 20 -1           // "Error: La division no es natural"
```

13. **(OPTATIVO)** Crea una nueva excepción **DivideParException**, que controle si se va a dividir un número impar entre 2. En ese caso, deberá lanzar una excepción indicando que la división no es exacta. Añade el código necesario al programa principal para controlar esta excepción nueva.

PARA ENTREGAR

- Fichero **Ej1.java** con todas las modificaciones indicadas, de forma que las pruebas realizadas en el paso 13 den los resultados correctos.

- Fichero **NumeroNaturalException.java** con la nueva excepción creada.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 7

1. Implementaremos ahora un ejercicio para practicar diferentes aspectos sobre hilos y multiprogramación:

1. Antes de comenzar, lee la introducción del punto **2.2** (*Hilos*), y los apartados **2.2.1** (*Creación de hilos*) y **2.2.2** (*Estado y propiedades de los hilos*) del tema 2 de teoría.
2. Echa un vistazo a la clase *Ej2.java* que se proporciona en la plantilla de la sesión. Verás que tiene una subclase llamada *MiHilo* que es un hilo. Tiene un campo *nombre* que sirve para dar nombre al hilo, y un campo *contador*. También hay un método *run* que itera el contador del 1 al 1000; en cada iteración hace un *System.gc()*, es decir, llama al recolector de basura de Java. Lo hacemos para consumir algo de tiempo en cada iteración.

   Desde la clase principal (*Ej2*), se crea un hilo de este tipo, y se ejecuta (en el constructor). También en el constructor hacemos un bucle **do...while**, donde el programa principal va examinando qué valor tiene el contador del hilo en cada momento, y luego duerme un tiempo hasta la próxima comprobación, mientras el hilo siga vivo.

3. Compila y prueba el funcionamiento de la clase de la siguiente forma, comprobando que el hilo se lanza y ejecuta sus 1000 iteraciones.

   ```
   javac Ej2.java
   java Ej2
   ```

   ¿Cuántos hilos o flujos de ejecución hay en total? ¿Qué hace cada uno? (AYUDA: en todo programa al menos hay UN flujo de ejecución, que no es otro que el programa principal. Aparte, podemos tener los flujos secundarios (hilos) que queramos).

4. Vamos a añadir y lanzar dos hilos más de tipo *MiHilo* en la clase *Ej2.* Para ello podemos copiar y pegar el código que crea y lanza el primer hilo, para hacer otros dos nuevos:

   ```
   public Ej2()
   {
     MiHilo t = new MiHilo("Hilo 1");
     MiHilo t2 = new MiHilo("Hilo 2");
     MiHilo t3 = new MiHilo("Hilo 3");

     t.start();
     t2.start();
     t3.start();

     do
     {
            System.out.print("Hilo 1 = " + t.contador +
   ```

```
                          ", Hilo 2 = " + t2.contador +
                          ", Hilo 3 = " + t3.contador + "\r");
          try {
                  Thread.currentThread().sleep(100);
          } catch(InterruptedException e) { }

  } while (t.isAlive() || t2.isAlive() || t3.isAlive());
}
```

El bucle **do...while** lo modificamos también, y ahora nos muestra cuánto vale el contador de cada hilo cada vez. Dentro del bucle, hacemos que el hilo actual (es decir, el programa principal), se duerma cada 100 ms, para dejar sitio a los hilos en el procesador.

Ejecuta después el programa varias veces (3 o 4), y comprueba el orden en el que terminan los hilos. ¿Existe mucha diferencia de tiempo entre la finalización de éstos? ¿Por qué?

5. Vamos a modificar la prioridad de los hilos, para hacer que terminen en orden inverso al que se lanzan. Para ello daremos al hilo 3 la máxima prioridad, al hilo 2 una prioridad normal, y al hilo 1 una prioridad mínima:

```
public Ej2()
{
  MiHilo t = new MiHilo("Hilo 1");
  MiHilo t2 = new MiHilo("Hilo 2");
  MiHilo t3 = new MiHilo("Hilo 3");

  t.setPriority(Thread.MIN_PRIORITY);
  t2.setPriority(Thread.NORM_PRIORITY);
  t3.setPriority(Thread.MAX_PRIORITY);

  t.start();
  t2.start();
  t3.start();

  ...
}
```

Observa si los hilos terminan en el orden establecido. ¿Existe esta vez más diferencia de tiempos en el orden de finalización?

6. Observa los campos locales de la clase *MiHilo*: la variable *contador* y el campo *nombre*. A juzgar por lo que has visto en la ejecución de los hilos: ¿Podría modificar el hilo t2 el valor de estos campos para el hilo t3, es decir, comparten los diferentes hilos estos campos, o cada uno tiene los suyos propios? ¿Qué pasaría si los tres hilos intentasen modificar el campo *valor* de la clase principal *Ej2*, también tendrían una copia cada uno o accederían los tres a la misma variable?
7. **(OPTATIVO)** Crea una clase **Ej2b** similar a la anterior, pero en la que no sea el programa principal quien active los 3 hilos, sino que cada uno de los 3, el empezar a funcionar, cree e inicie el hilo siguiente (hasta un total de 3 hilos). De esta forma, se ejecutarán en paralelo, pero no será

el programa principal quien los lance. Este sólo lanzará un hilo, que será el responsable de seguir la cadena.

2. En este ejercicio vamos a practicar con la sincronización entre múltiples hilos, resolviendo el problema de los productores y los consumidores. Vamos a definir 3 clases: el hilo **Productor**, el hilo **Consumidor**, y el objeto **Recipiente** donde el productor deposita el valor producido, y de donde el consumidor extrae los datos.

1. Antes de comenzar, lee el apartado **2.2.3** (*Sincronización de hilos*) del tema 2 de teoría.
2. Echa un vistazo a la clase *Ej3.java* que se proporciona en la plantilla de la sesión. Verás que tiene 3 subclases: una llamada *Productor*, que serán los hilos que se encarguen de producir valores, otra llamada *Consumidor*, que serán los hilos que se encargarán de consumir los valores producidos por los primeros, y una tercera llamada *Recipiente*, donde los *Productores* depositarán los valores producidos, y los *Consumidores* retirarán dichos valores.

   Los valores producidos no son más que números enteros, que los productores generarán y los consumidores leerán.

   Si miramos el código de los hilos *Productores*, vemos (método *run*) que los valores que producen van de 0 a 9, y entre cada producción duermen una cantidad aleatoria, entre 1 y 2 segundos.

   En cuanto a los *Consumidores*, vemos que entre cada consumición también duermen una cantidad aleatoria entre 1 y 2 segundos, y luego consumen el valor.

   Para producir y para consumir se utilizan los métodos *produce* y *consume*, respectivamente, de la clase *Recipiente*.

3. Compila y prueba varias veces (3 o 4) el funcionamiento de la clase.

   ```
   javac Ej3.java
   java Ej3
   ```

   ¿Funciona bien el programa? ¿Qué anomalía(s) detectas?

4. Como habrás podido comprobar, el programa no funciona correctamente: hay iteraciones en las que el hilo *Consumidor* consume ANTES de que el *Productor* produzca el nuevo valor, o el *Productor* produce dos valores seguidos sin que el *Consumidor* los consuma.

   Es necesario sincronizarlos de forma que el *Consumidor* se espere a que el *Productor* produzca, y luego el *Productor* espere a que el *Consumidor* consuma, antes de generar otro nuevo valor. Es decir, el funcionamiento correcto debería ser que el consumidor consuma

exactamente los mismos valores que el productor ha producido, sin saltarse ninguno ni repetirlos.

5. Vamos a añadir el código necesario en los métodos *produce* y *consume* para sincronizar el acceso a ellos. El comportamiento debería ser el siguiente:
   a. Si queremos producir y todavía hay datos disponibles en el recipiente, esperaremos hasta que se saquen, si no produciremos y avisamos a posibles consumidores que estén a la espera.

```
public void produce(int valor)
{
    /* Si hay datos disponibles esperar */

    if(disponible) {
            try {
                    wait();
            } catch(InterruptedException e) { }
    }

    this.valor = valor;

    /* Ya hay datos disponibles */

    disponible = true;

    /* Notificar de la llegada de datos a consumidores */

    notifyAll();
}
```

¿Qué hace el método *wait* en este código?

¿Para qué se utiliza el flag *disponible*?

¿Qué efecto tiene la llamada a *notifyAll*?¿Qué pasaría si no estuviese?

   b. Si queremos consumir y no hay datos disponibles en el recipiente, esperaremos hasta que se produzcan, si no consumimos el valor disponible y avisamos a posibles productores que estén a la espera.

```
public int consume()
{
    /* Si no hay datos disponibles */

    if(!disponible) {
            try {
                    wait();
            } catch(InterruptedException e) { }
    }

    /* Ya no hay datos disponibles */
```

```
    disponible = false;

    /* Notificar de que el recipiente esta libre */
    notifyAll();

    return valor;
}
```

6. Compilad y ejecutad el programa. ¿Qué excepción o error da? ¿A qué puede deberse?
7. El error del paso anterior lo da porque no podemos llamar a los métodos *wait* o *notify/notifyAll* si no tenemos la variable cerrojo. Dicha variable se consigue dentro de bloques de código *synchronized*, de forma que el primer hilo que entra es quien tiene el cerrojo, y hasta que no salga o se ponga a esperar, no lo liberará.

   Para corregir el error, debemos hacer que tanto *produce* como *consume* sean dos métodos *synchronized*, de forma que nos aseguremos que sólo un hilo a la vez entrará, y podrá ejecutar dichos métodos.

```
public synchronized int consume()
{
  ...
}

public synchronized void produce (int valor)
{
  ...
}
```

8. Compilar y comprobar que el programa funciona correctamente. Si lo ejecutáis varias veces, podría darse el caso de que aún salgan mensajes de consumición ANTES de que se muestren los de producción respectivos, por ejemplo:

```
Produce 0
Consume 0
Consume 1
Produce 1
Produce 2
Consume 2
...
```

   ... pero aún así el programa es correcto. ¿A qué se debe entonces que puedan salir los mensajes en orden inverso? (AYUDA: observad el método *System.out.println(...)* que hay al final de los métodos *run* de *Productor* y *Consumidor*, que es el que muestra estos mensajes... ¿hay algo que garantice que se ejecute este método en orden?)

PARA ENTREGAR

- Fichero **Ej2.java** con todas las modificaciones indicadas.
- Fichero **Ej3.java** con el problema de los productores y los consumidores resuelto

- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 8

En esta sesión practicaremos algunos conceptos básicos de Entrada/Salida, como la lectura y escritura básica con ficheros, y el uso de ficheros de propiedades, y entrada/salida estándar. Antes de comenzar, lee la introducción del punto **2.3** (*Entrada/Salida*), y los apartados **2.3.1** (*Flujos de Entrada/Salida*), **2.3.2** (*Entrada, salida y salida de error estándar*) y **2.3.3** (*Acceso a ficheros*) del tema 2 de teoría.

1. En este primer ejercicio practicaremos la lectura y escritura básica con ficheros, utilizando las dos posibles alternativas: *Streams* y *Readers/Writers*:

1. Echa un vistazo a la clase *Ej4.java* que se proporciona en la plantilla de la sesión. Verás que hay un constructor vacío, y un campo llamado *cabecera*, que contiene una cadena de texto. También hay dos métodos vacíos, *leeEscribeStream* y *leeEscribeWriter*, y un método *main* que crea un objeto de tipo *Ej4* y llama a estos dos métodos. Lo que vamos a hacer es rellenar esos dos métodos de la forma que se nos indica a continuación.
2. El primero de los métodos *leeEscribeStream* va a leer un fichero de entrada (el fichero *entrada.dat* que se os proporciona en la plantilla), y lo va a volcar a un fichero de salida (fichero *salidaStream.dat*), pero añadiéndole la cadena *cabecera* como encabezado del fichero. Para hacer todo eso empleará flujos de tipo *stream* (*InputStream* para leer, *OutputStream* para escribir, o cualquier subclase derivada de éstas).
    a. Primero obtendremos el flujo de entrada para leer del fichero. Utilizaremos un objeto de tipo *FileInputStream*, que es el stream preparado para leer de ficheros:

    ```
    public void leeEscribeStream()
    {
      FileInputStream in = new FileInputStream("entrada.dat");
    }
    ```

    b. Después obtendremos el flujo de salida, para escribir en el fichero destino. Emplearemos un objeto de tipo *FileOutputStream*, que es el stream preparado para volcar datos a ficheros:

    ```
    public void leeEscribeStream()
    {
      FileInputStream in = new FileInputStream("entrada.dat");
      FileOutputStream out = new FileOutputStream("fstr.dat");
    }
    ```

    c. El siguiente paso es leer el contenido de la entrada, e irlo volcando en la salida. Para leer datos de la entrada emplearemos el método *read()* de *FileInputStream*, que irá leyendo caracteres (transformados en enteros). Para escribir, utilizaremos el método

write() de *FileOutputStream*, que vuelca esos mismos enteros que leemos:

```
public void leeEscribeStream()
{
    FileInputStream in = new FileInputStream("...");
    FileOutputStream out = new FileOutputStream("...");
    int c;

    while ((c = in.read()) != -1)
    {
            out.write(c);
    }
}
```

Echa un vistazo a la documentación sobre el método *read.* ¿Por qué se compara el dato que se lee con -1?

d. Finalmente, lo que nos queda es cerrar tanto el flujo de entrada como el de salida:

```
public void leeEscribeStream()
{
    ...
    in.close();
    out.close();
}
```

e. Compila el programa. Te dará errores porque se deben capturar ciertas excepciones cuando se trabaja con métodos de entrada salida en fichero (*FileNotFoundException* e *IOException*, concretamente).

```
public void leeEscribeStream()
{
    try
    {
            ...
    } catch (FileNotFoundException e) {
            Sytem.err.println ("Fichero no encontrado");
    } catch (IOException e2) {
            System.err.println ("Error ...");
    }
}
```

Captúralas y prueba el resultado.

f. Al ejercicio le falta algo, porque si recuerdas, aparte de leer y volcar el contenido del fichero, debemos añadir a la salida como cabecera el contenido del campo *cabecera*.

Observa en la API que la clase *FileOutputStream* no tiene métodos para escribir directamente una cadena a fichero. Lo que vamos a hacer es convertir la cadena a un array de *bytes*, y luego

utilizar el método *write(byte[ ] b)* para volcarla. Todo esto lo haremos justo antes de empezar a leer el fichero de entrada, y volcar su contenido:

```
public void leeEscribeStream()
{
    try
    {
            FileInputStream in =
                    new FileInputStream("entrada.dat");
            FileOutputStream out =
                    new
FileOutputStream("salidaStream.dat");

            byte[] b = cabecera.getBytes();
            out.write(b);

            int c;
            while ((c = in.read()) != -1)

            ...
}
```

Prueba el método ya completo, y comprueba que el fichero de salida (*salidaStream.dat*) deja algo como:

```
# Esto es la cabecera del fichero que hay que introducir
Hola, este es el texto
del fichero de entrada
que debería copiarse en el fichero de salida
```

3. El segundo método, *leeEscribeWriter*, leerá el mismo fichero de entrada (*entrada.dat*), y lo volcará a otro fichero de salida diferente (*salidaWriter.dat*), empleando flujos de tipo *Reader* y *Writer* (como *FileReader* o *FileWriter*, o cualquier otro subtipo).
    a. Igual que en el método anterior, primero obtendremos las variables para leer de la entrada y escribir en la salida. Para leer podríamos utilizar la clase *FileReader*, pero en su lugar vamos a utilizar la clase *BufferedReader* que nos va a permitir leer líneas enteras del fichero, en lugar de leer carácter a carácter. Para escribir, vamos a utilizar la clase *PrintWriter*, que también nos permitirá escribir líneas enteras en la salida.

```
public void leeEscribeWriter()
{
    BufferedReader br =
      new BufferedReader(new FileReader("entrada.dat"));
    PrintWriter pw =
      new PrintWriter(new FileWriter("salidaWriter.dat"));
}
```

Observad que para construir tanto el *BufferedReader* como el *PritWriter* nos valemos de un objeto *FileReader* o *FileWriter*, respectivamente. Lo que hacemos es simplemente crear un buffer de entrada (*BufferedReader*) o de salida *(PrintWriter*) sobre

el *FileReader* o el *FileWriter* para poder acumular cadenas de texto enteras antes de leerlas o escribirlas.

b. El siguiente paso es leer el contenido de la entrada, e irlo volcando en la salida. Para leer datos de la entrada emplearemos el método *readLine()* de *BufferedReader*, que irá leyendo líneas enteras del fichero. Para escribir, utilizaremos el método *println()* de *PrintWriter*, que vuelca esas mismas líneas que leemos:

```
public void leeEscribeWriter()
{
    BufferedReader br = new ...;
    PrintWriter pw = new ...;

    String linea = "";
    while ((linea = br.readLine()) != null)
    {
            pw.println(linea);
    }
}
```

El uso de *PrintWriter* permite formatear la salida de la misma forma que si la estuviésemos sacando por pantalla, puesto que tiene los mismos métodos que el campo *System.out* (métodos *println, print*, etc).

Echa un vistazo a la documentación sobre el método *readLinea.* ¿Por qué se compara el dato que se lee con *null*?

c. Finalmente, lo que nos queda es cerrar tanto el flujo de entrada como el de salida:

```
public void leeEscribeWriter()
{
    ...
    br.close();
    pw.close();
}
```

d. Compila el programa. Te dará errores porque se deben capturar las mismas excepciones que antes (*FileNotFoundException* e *IOException*).

```
public void leeEscribeWriter()
{
    try
    {
            ...
    } catch (FileNotFoundException e) {
            Sytem.err.println ("Fichero no encontrado");
    } catch (IOException e2) {
            System.err.println ("...");
    }
}
```

Captúralas y prueba el resultado.

e. Para completar el ejercicio, nos falta añadir la cabecera antes de volcar el fichero. Observa que con *PrintWriter* no hace falta que convirtamos la cadena a *bytes* y luego la escribamos, podemos escribir directamente la cadena, antes de empezar a leer el fichero:

```
public void leeEscribeWriter()
{
   try
   {
           BufferedReader br = ...;
           PrintWriter pw = ...;
           pw.print(cabecera);

           String linea = "";
           while ((linea = br.readLine()) != null)

           ...
}
```

Prueba el método ya completo, y comprueba que el fichero de salida (*salidaWriter.dat*) deja el mismo resultado que con el método anterior.

f. NOTA: observa la API de la clase *PrintWriter*, y verás que tiene constructores que permiten crear este tipo de objetos a partir de *Writers* (como hemos hecho aquí) como a partir de *OuputStreams* (como habríamos hecho en el paso 2), con lo que podemos utilizar esta clase para dar formato a la salida de un fichero en cualquiera de los casos.

2. En este segundo ejercicio practicaremos el uso de ficheros de propiedades, y el uso de la entrada y salida estándares.

1. Echa un vistazo a la clase *Ej5.java* que se proporciona en la plantilla de la sesión. Sólo tiene un constructor vacío, y un método *main* que le llama. Vamos a completar el constructor de la forma que veremos a continuación.
2. Lo que vamos a hacer en el constructor es leer un fichero de propiedades (el fichero *prop.txt* que se proporciona en la plantilla), y luego pedirle al usuario que, por teclado, indique qué valores quiere que tengan las propiedades. Una vez establecidos los valores, volveremos a guardar el fichero de propiedades.
   a. Lo primero que vamos a hacer es leer el fichero de propiedades. Para ello utilizaremos un objeto *java.util.Properties*, lo crearemos y llamaremos a su método *load()* para cargar las propiedades del fichero *prop.txt*:

```
public Ej5()
{
```

```
    Properties p = new Properties();
    p.load(new FileInputStream("prop.txt"));
}
```

Observa que para cargar las propiedades, al método *load* le debemos pasar un *InputStream* desde el que leerlas. En este caso le pasamos un *FileInputStream* con el fichero *prop.txt*.

b. Ahora ya tenemos en el objeto *p* todas las propiedades del fichero. Vamos a irlas recorriendo una a una, e indicando al usuario que teclee su valor. Para recorrer las propiedades obtendremos un *Enumeration* con sus nombres, y luego lo iremos recorriendo, y sacándolo por pantalla:

```
public Ej5()
{
    Properties p = new Properties();
    p.load(new FileInputStream("prop.txt"));

    Enumeration en = p.propertyNames();
    while (en.hasMoreElements())
    {
            String prop = (String)(en.nextElement());
            System.out.println("Valor para " + prop);
    }
}
```

Observa el orden en que van mostrándose las propiedades. ¿Es el mismo que el que hay en el fichero? ¿A qué crees que puede deberse? (AYUDA: cuando nosotros *enumeramos* una serie de características, no tenemos que seguir un orden necesariamente. Del mismo modo, cuando introducimos valores en una tabla hash, el orden en que se guardan no es el mismo que el orden en que los introducimos).

c. Lo que hacemos con este bucle es sólo recorrer los nombres de las propiedades y sacarlos por pantalla. Nos falta hacer que el usuario teclee los valores correspondientes. Para ello utilizaremos un objeto de tipo *BufferedReader*, que en este caso leerá líneas de texto que el usuario entre desde teclado:

```
public Ej5()
{
    Properties p = new Properties();
    p.load(new FileInputStream("prop.txt"));

    Enumeration en = p.propertyNames();
    BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));

    while (en.hasMoreElements())
    {
            ...
    }
}
```

observad que construimos el *BufferedReader* para leer de un *InputStream* (no de un *Reader*). Esto lo podemos hacer si nos ayudamos de la "clase puente" *InputStreamReader*, que transforma un tipo de lector en otro.

Lo que nos queda por hacer es pedirle al usuario que, para cada nombre de propiedad, introduzca su valor, y luego asignarlo a la propiedad correspondiente:

```
public Ej5()
{
    ...

    while (en.hasMoreElements())
    {
            String prop = (String)(en.nextElement());
            System.out.println("Valor para " + prop);
            String valor = in.readLine();
            p.setProperty(prop, valor);
    }
}
```

d. Finalmente, cerramos el buffer de entrada, y guardamos las propiedades en el fichero.

```
public Ej5()
{
    ...
    in.close();
    p.store(new FileOutputStream("prop.txt"), "Cabecera");
}
```

e. Compilad y ejecutad el programa. Para que os compile deberéis capturar las excepciones que se os indique en los errores de compilación:

```
public Ej5()
{
    try
    {
            ...
    } catch(...) {}
}
```

3. **(OPTATIVO)** Añadid el código necesario al ejercicio para que, además de poder modificar los valores de las propiedades, podamos añadir por teclado nuevas propiedades al fichero, y guardarlas con las existentes.

PARA ENTREGAR

- Fichero **Ej4.java** y **Ej5.java** con todas las modificaciones indicadas.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 9

En esta sesión practicaremos conceptos algo más avanzados de Entrada/Salida, como la lectura de tokens desde ficheros, y la lectura de objetos complejos.

1. En este primer ejercicio practicaremos la lectura de tokens de un fichero, y su almacenamiento para realizar alguna operación.

1. Antes de comenzar, lee el apartado **2.3.4** (*Lectura de tokens*) del tema 2 de teoría.
2. Echa un vistazo a la clase *Ej6.java* que se proporciona en la plantilla de la sesión. Verás que hay un constructor vacío, y un método *main* que le llama. Rellenaremos el constructor como se indica en los siguientes pasos.
3. Lo que vamos a hacer es que el constructor acceda a un fichero (fichero *matriz.txt*) que tiene una matriz *m x n*. Dicho fichero tiene la siguiente estructura:

```
; Comentario de cabecera
m n
A11 A12 A13...
A21 A22 A23...
...
```

donde *m* son las filas, *n* las columnas, y después aparece la matriz puesta por filas, con un espacio en blanco entre cada elemento.

El ejercicio leerá la matriz (utilizando un *StreamTokenizer* sobre el fichero), construirá una matriz (array) con los datos leídos, después elevará al cuadrado cada componente, y volcará el resultado en un fichero de salida.

   a. Primero obtendremos el flujo de entrada para leer del fichero, y el *StreamTokenizer*:

```
public Ej6()
{
    StreamTokenizer st =
       new StreamTokenizer(new FileReader("matriz.txt"));
}
```

   b. Después establecemos qué caracteres van a identificar las líneas de comentarios. En este caso, los comentarios se identifican por punto y coma:

```
public Ej6()
{
    StreamTokenizer st =
       new StreamTokenizer(new FileReader("matriz.txt"));

    st.commentChar(';');
```

```
}
```

c. Después del comentario irán el número de filas y de columnas. Utilizamos el método *nextToken* del *tokenizer* para leerlos, y luego accedemos al campo *nval* para obtener qué valor numérico se ha leído en cada caso:

```
public Ej6()
{
    StreamTokenizer st =
       new StreamTokenizer(new FileReader("matriz.txt"));

    st.commentChar(';');

    int filas, columnas;

    st.nextToken();
    filas = (int)(st.nval);
    // Filas

    st.nextToken();
    columnas = (int)(st.nval);                    //
Columnas
}
```

NOTA: asumimos que el fichero va a tener un formato correcto, y no tenemos que controlar que haya elementos no deseados por enmedio.

¿Qué se habría leído en primer lugar si no hubiésemos identificado la primera línea como comentario? ¿Dónde podríamos haber consultado ese valor leído?

d. Lo siguiente es ir leyendo los elementos de la matriz. Construimos un array de enteros de *filas* x *columnas*, y luego lo vamos rellenando con los valores que nos dé el *StreamTokenizer*:

```
public Ej6()
{
    ...
    int[][] matriz = new int[filas][columnas];
    int t;

    for (int i = 0; i < filas; i++)
            for (int j = 0; j < columnas; j++)
            {
                    t = st.nextToken();
                    if (t != StreamTokenizer.TT_EOF)
                    {
                            matriz[i][j] = (int)(st.nval);
                    }
            }
}
```

e. Por último, calculamos el cuadrado de cada elemento de la matriz (utilizamos el método *pow* de la clase *Math*), y guardamos la matriz resultado en otro fichero de salida (*matrizSal.txt*), con el mismo formato que el de entrada:

```
public Ej6()
{
    ...

    for (int i = 0; i < filas; i++)
            for (int j = 0; j < columnas; j++)
            {
             matriz[i][j]=(int)(Math.pow(matriz[i][j], 2));
            }

    // Volcamos la salida a fichero

    PrintWriter pw =
      new PrintWriter (new FileWriter("matrizSal.txt"));
    pw.println ("; Matriz resultado");
    pw.println ("" + filas + " " + columnas);
    for (int i = 0; i < filas; i++)
    {
            for (int j = 0; j < columnas; j++)
            {
                    pw.print("" + matriz[i][j] + " ");
            }
            pw.println();
    }
    pw.close();
}
```

f. Compila y ejecuta el programa (captura las excepciones adecuadas para que te compile bien). Comprueba que el fichero de salida genera el resultado adecuado:

```
; Matriz resultado
3 3
1 4 9
16 25 36
49 64 81
```

Prueba también a pasarle este mismo fichero como entrada al programa, y que genere otro fichero de salida diferente.

4. **(OPTATIVO)** Supongamos que tenemos un fichero de entrada cuya estructura debe ser una alternancia de palabras y números, es decir, debe haber una palabra seguida siempre de un número:

```
hola 1 pepe 2 otra 53 adios 877
```

Construye varios ficheros ejemplo de entrada, e implementa la clase **LeeFicheroAlternado**, que lea estos ficheros mediante un *StreamTokenizer*. La clase deberá lanzar una excepción del tipo

adecuado, indicando error de sintaxis, si el fichero no tiene la estructura adecuada. En caso de ser correcto, mostrará su contenido en pantalla.

2. En este segundo ejercicio practicaremos cómo utilizar los ficheros para almacenar y leer objetos complejos. Hasta ahora sólo hemos trabajado con enteros o cadenas, y para leerlos basta con leer un stream de bytes, o utilizar un *tokenizer* y procesar el fichero de la forma que nos convenga.

Imaginemos que trabajamos con un objeto complejo que encapsula diferentes tipos de datos (enteros, cadenas, vectores, etc). A la hora de guardar este elemento en fichero, se nos plantea el problema de cómo representar su información para volcarla. De la misma forma, a la hora de leerlo, también debemos saber cómo extraer y recomponer la información del objeto. Veremos que hay clases Java que hacen todo este trabajo mucho más sencillo.

1. Antes de comenzar, lee los apartados **2.3.5** (*Acceso a ficheros o recursos dentro de un JAR*), **2.3.6** (*Codificación de datos*) y **2.3.7** (*Serialización de objetos*) del tema 2 de teoría.
2. Echa un vistazo al fichero *Ej7.java* que se proporciona en la plantilla de la sesión. Tiene una clase principal (*Ej7*), con un constructor vacío y un método *main* que le llama.

   También tiene una clase interna llamada *ObjetoFichero*. Observa que dicha clase interna tiene diferentes tipos de campos: una cadena, un entero, un double y un Vector. Tiene un constructor que asigna valores a los campos, y luego dos métodos: uno *addCadena* que añade cadenas al Vector, y otro *imprimeObj* que devuelve una cadena que representa los valores de los campos del objeto. Es **importante** también resaltar que esta clase es **Serializable**, es decir, implementa la interfaz *Serializable*, lo que permitirá que se pueda guardar y leer de flujos o ficheros como un objeto complejo en bloque.

3. Lo que vamos a hacer en el constructor de *Ej7* es crear varios objetos de tipo *ObjetoFichero*, y luego guardarlos en un fichero de salida (*ficheroObj.dat*). Finalmente, leeremos los objetos de ese fichero de salida y mostraremos por pantalla los valores de sus campos, para comprobar que se han guardado y leído de forma correcta.
   a. Lo primero es crear varios objetos (por ejemplo, dos) de tipo *ObjetoFichero*, cada uno con valores diferentes:

```
public Ej7()
{
    ObjetoFichero of = new ObjetoFichero ("cad1", 1, 1.5);
    of.addCadena("cad2");
    of.addCadena("cad3");

    ObjetoFichero of2 = new ObjetoFichero ("ca1b", 2, 2.5);
    of2.addCadena("cad2b");
    of2.addCadena("cad3b");
}
```

El vector de elementos de cada objeto tiene 3 cadenas de texto, más aparte los valores que le hemos dado a los otros campos de la clase.

b. Ahora guardaremos en el fichero de salida (*ficheroObj.dat*) estos dos objetos creados. Vamos a utilizar la clase *ObjectOutputStream* que permite abrir un flujo de salida y meter en él objetos complejos, siempre que sean *Serializables*, como el nuestro. Simplemente basta con utilizar su método *writeObject*, y pasarle el objeto que queremos guardar:

```
public Ej7()
{
    ...

    ObjectOutputStream oos =
       new ObjectOutputStream(new FileOutputStream("..."));
    oos.writeObject(of);
    oos.writeObject(of2);
    oos.close();
}
```

Observa lo sencillo que resulta guardar objetos complejos de esta forma. La única condición que deben cumplir es que deben ser *Serializables*.

c. Finalmente, leeremos los objetos guardados del fichero que hemos generado, y sacaremos por pantalla los valores de sus campos. Para leer los ficheros utilizaremos el análogo a la clase anterior, es decir, un objeto de tipo *ObjectInputStream*, que permite leer objetos complejos enteros (siempre que sean serializables), desde flujos de entrada. Sólo hay que utilizar su método *readObject*, que extraerá cada objeto de ese tipo que tengamos en el flujo.

```
public Ej7()
{
    ...

    ObjectInputStream ois =
       new ObjectInputStream(new FileInputStream("..."));
    ObjetoFichero ofLeido1 =
      (ObjetoFichero)(ois.readObject());
    ObjetoFichero ofLeido2 =
      (ObjetoFichero)(ois.readObject());
}
```

Es importante hacer notar que el método *readObject* devuelve un objeto de tipo *Object*, que luego nosotros debemos convertir (con un *cast*) al tipo de datos que necesitemos.

Para sacar el valor de los campos por pantalla, recordemos que cada objeto de tipo *ObjetoFichero* tiene un método llamado

imprimeObj que devuelve una cadena que representa su contenido. Así que basta con imprimir esa cadena:

```
public Ej7()
{
    ...
    System.out.println(ofLeido1.imprimeObj());
    System.out.println(ofLeido2.imprimeObj());
}
```

d. Compilad y ejecutad el programa (capturad las excepciones necesarias para que compile), y observad si se muestran por pantalla los valores correctos. Eso será prueba de que los objetos se han guardado y leído bien.

   ¿Qué pasaría si *ObjetoFichero* no implementase la interfaz *Serializable*? ¿Qué excepción saltaría al ejecutar?

4. **(OPTATIVO)** Crea una nueva clase **VectorObjetoFichero** que internamente sea una lista de elementos de tipo *ObjetoFichero*. Puedes utilizar cualquier tipo de colección de Java (*Vector, List, ArrayList*, etc). Después, haz que la clase sea Serializable, y prueba a guardar y leer objetos de la misma en diferentes ficheros.

PARA ENTREGAR

- Fichero **Ej6.java** y **Ej7.java** con todas las modificaciones indicadas.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 10

En esta sesión vamos a hacer un ejercicio práctico que englobe varios de los conceptos vistos hasta ahora.

1. Vamos a construir una aplicación que permita definir distintos tipos de figuras geométricas, y poderlas leer y guardar en ficheros. Antes de explicar los pasos a seguir, daremos un rápido vistazo a la estructura de la aplicación, para entender qué ficheros y directorios tenéis en la plantilla.

En la plantilla tenemos dos paquetes de clases:

- **paquete geom**: dedicado a distintos tipos de figuras geométricas. Contiene una clase padre abstracta llamada *Figura*, con un método abstracto llamado *imprime*. Esta clase tendrá una serie de subclases, que son los ficheros *Circulo.java*, *Rectangulo.java* y *Linea.java* que deberéis completar. De esta forma lo que tenemos es un conjunto de figuras geométricas de distintos tipos, y todas ellas subtipos de una clase padre genérica *Figura*.
- **paquete io**: aquí tendremos clases relacionadas con tareas de entrada/salida: la clase *EntradaTeclado* deberá contener el código necesario para recoger los datos que el usuario introduce por teclado. La clase *IOFiguras* tendrá métodos para leer de un fichero un conjunto de figuras geométricas (del paquete *geom* anterior), y para guardar un conjunto de figuras en un fichero.

Finalmente, tenemos una clase principal en el directorio raíz, llamada *AplicGeom*, que leerá las figuras que haya en un fichero que se le pase como parámetro, y mostrará un menú de opciones para que el usuario pueda:

1. Crear una nueva figura geométrica
2. Borrar una figura geométrica de la lista existente
3. Guardar la lista de figuras actual en el fichero de entrada
4. Salir del programa

Veremos ahora qué pasos seguir para construir todo esto.

1. En primer lugar, construiremos el paquete **geom**. La clase *Figura* ya está completa, así que rellenaremos las otras:
    a. La clase **Circulo** deberá tener dos campos enteros *x*, *y*, que indiquen la posición de su centro, y luego un campo entero *radio*, que indique la longitud de su radio. Haremos un constructor que tome estos tres campos y los asigne, y luego métodos *getXXX* y *setXXX* para obtener y cambiar el valor de cada campo. Deberemos rellenar también el método *imprime*, ya que es abstracto en la superclase. Dicho método devolverá una cadena con información de los campos de la figura:

```
package geom;
```

```
public class Circulo extends Figura implements
java.io.Serializable
{
    int x;
    int y;
    int radio;

    public Circulo(int x, int y, int radio)
    {
            ...
    }

    public int getX()
    {
            ...
    }

    public int getY()
    {
            ...
    }

    public int getRadio()
    {
            ...
    }

    public void setX(int x)
    {
            ...
    }

    public void setY(int y)
    {
            ...
    }

    public void setRadio(int radio)
    {
            ...
    }

    public String imprime()
    {
    return "CIRCULO: (" + x + ", " + y + "), r = " + radio;
    }
}
```

b. De forma similar construimos las clases **Rectangulo** y **Linea**. Estas clases tendrán cada una cuatro campos: *x1, y1, x2* e *y2*. En el caso del *Rectangulo*, (*x1, y1*) indican la posición del vértice superior izquierdo, y (*x2, y2*) la del vértice inferior derecho. En el caso de la *Linea*, (*x1, y1*) indican un extremo de la línea, y (*x2, y2*) el otro extremo.

```
...
public class Rectangulo extends Figura implements
java.io.Serializable
```

```
{
    int x1;
    int y1;
    int x2;
    int y2;
...
...
public class Linea extends Figura implements
java.io.Serializable
{
    int x1;
    int y1;
    int x2;
    int y2;
...
```

Recordad definir también el método *imprime* para que saque una cadena de texto con información relativa a cada figura.

En el caso del *Rectangulo*, sería algo como:

```
"RECTANGULO: (" + x1 + ", " + y1 + ") - (" + x2 + ", " +
y2 + ")"
```

Y para la *Linea*:

```
"LINEA: (" + x1 + ", " + y1 + ") - (" + x2 + ", " + y2 +
")"
```

NOTA IMPORTANTE: Observad que tanto la clase padre *Figura* como todas las subclases definidas son **Serializables** (implementan la interfaz *Serializable*). Esto nos permitirá después guardarlas y leerlas de ficheros como objetos completos. Recordad también poner la directiva *package* al principio del fichero, indicando el paquete al que pertenecen las clases.

2. Completaremos ahora el paquete **io**.
    a. Comenzamos por la clase **EntradaTeclado**. Simplemente debe recoger los datos que el usuario introduzca por teclado, así que declararemos un campo de tipo *BufferedReader*:

```
package io;

import java.io.*;

public class EntradaTeclado
{
    BufferedReader in = null;
    ...
```

Después, en el constructor hacemos que dicho buffer lea de la entrada estándar (*System.in*):

```
    ...
    public EntradaTeclado()
```

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

Finalmente, definimos un método *leeTeclado* que devuelva una cadena con cada línea que el usuario ha escrito hasta pulsar Intro. Esta cadena la utilizaremos después desde el programa principal, para recoger todas las órdenes del usuario (capturamos las excepciones necesarias, y devolvemos *null* si se produce algún error):

```
    ...
    public String leeTeclado()
    {
            try
            {
                    return in.readLine();
            } catch (Exception e) {
                    return null;
            }
    }
}
```

b. Por otro lado, completamos la clase **IOFiguras**. Esta clase tendrá dos métodos **estáticos** (para llamarlos sin tener que crear un objeto de la clase): uno servirá para leer un conjunto de figuras de un fichero, y el otro para guardarlas en fichero.

```
package io;

import geom.*;
import java.io.*;
import java.util.*;

public class IOFiguras
{
    public static Figura[] leeFiguras(String fichero)
    {
            ...
    }
    public static void guardaFiguras(Figura[] figs, String
fich)
    {
            ...
    }
}
```

i. Para el método **leeFiguras**, haremos que abra un *ObjectInputStream* contra el fichero *fichero*, y que vaya leyendo objetos de tipo *Figura* y metiéndolos en una lista (*ArrayList*), hasta que salte la excepción que indique el fin de fichero. Después, construiremos un array con la lista, y lo devolveremos. En el caso de que no haya elementos en la lista, devolveremos *null*.

```
public static Figura[] leeFiguras(String fichero)
{
            ArrayList alAux = new ArrayList();
            Figura f;

            try
            {
                    ObjectInputStream oin =
                      new ObjectInputStream (new FileInputStream(fichero));

                    ...      //      leer      figuras      del      fichero

                    oin.close();

            } catch (Exception e) {
                    // Se pueden producir 2 excepciones:
                    // FileNotFoundException si no encuentra el fichero
                    // IOException cuando llegue a fin de fichero
                    // DA IGUAL QUE SE PRODUZCA UNA U OTRA,
                    // LA CAPTURAMOS Y NO HACEMOS NADA MAS
            }


            // Si no había fichero, o no tenía figuras, la lista estará
            // vacía (se inicializa al principio del método, pero no llega
            // a entrar en el "while"). Entonces devolvemos null
            if (alAux.size() == 0)
                    return null;

            // Si había figuras, devolvemos un array con ellas
            return((Figura[])(alAux.toArray(new Figura[0])));
}
```

Observa que leemos objetos de tipo *Figura* (genéricos), sin discriminar si son líneas, círculos o rectángulos. El objeto se recupera tal cual, y del tipo que sea, aunque se trate como una figura genérica. Observa también que no hay forma de detectar el final del fichero. Se lanzará una IOException cuando lleguemos. Así que hay que meter la lectura de objetos en un bucle (infinito), y el bucle en un *try*, de forma que cuando llegue al final del fichero lanzará la excepción y saldremos del bucle, con todo el fichero ya leído. Por último, observa que **no pasa nada si el fichero no existe, o si se lanza cualquier excepción**. El método irá añadiendo figuras conforme las vaya leyendo: si no hay fichero, o no hay figuras en él, la lista quedará vacía (nunca entrará en el "while"), y se devolverá *null*. Si hay figuras, se irán colocando en la lista y se devolverá la lista al final. Después se trata de tomar lo que devuelva la llamada al método y hacer lo que corresponda:

```
Figura[] fig = IOFiguras.leeFiguras(nombre_fichero);
if (fig != null)
{
        // ... Procesar el array de figuras como se necesite
        // Por ejemplo, para meterlas en una lista:
```

```
        ArrayList figuras = new ArrayList();
        for (int i = 0; i < fig.length; i++)
                figuras.add(fig[i]);
} else {
        // ... No hay figuras que procesar
}
```

Captura las excepciones adecuadas para poderlo compilar. No es necesario que hagas nada en el bloque *catch* de la captura.

ii. Para el método de guardado, **guardaFiguras**, abrimos un *ObjectOutputStream* contra el fichero *fichero*, luego vamos recorriendo el array *figuras* y metiendo cada una en el fichero, con un *writeObject*:

```
public static void guardaFiguras(Figura[] figuras, String fich)
{
           try
           {
                   ObjectOutputStream oout =
                      new ObjectOutputStream (new FileOutputStream(fich));

                   ... // guardar figuras

                   oout.close();

           } catch (Exception e) {}
}
```

Captura las excepciones adecuadas, para poderlo compilar. No es necesario que hagas nada en el bloque *catch* de la captura.

3. Lo que nos queda es completar el programa principal **AplicGeom**. Veréis que la clase tiene un campo de tipo *EntradaTeclado* para leer las órdenes del usuario, otro para guardar el nombre del fichero con que trabajar, y otro que es una lista (*ArrayList*) con las figuras que haya en cada momento.

   Tenemos también un constructor al que se le pasa el nombre del fichero y lo asigna. Deberemos también inicializar la *EntradaTeclado* en el constructor, y leer del fichero las figuras (ayudándonos de la clase *IOFiguras*) y guardarlas en la lista:

```
public AplicGeom(String fichero)
{
  this.fichero = fichero;
  et = new EntradaTeclado();
  figuras = new ArrayList();
  Figura[] fig = IOFiguras.leeFiguras(fichero);

  // ... Aquí iría el resto del código: un bucle para
  // recorrer las figuras del array "fig" (si no es null)
  // y meterlas en la lista "figuras"
```

```
}
```

Después vemos que hay un método *main* que recoge el nombre del fichero de los parámetros de entrada, y crea un objeto de tipo *AplicGeom* con él. Finalmente, llama al método *iniciar* de la clase, que está vacío, pero deberá contener toda la lógica del programa. Deberéis completarlo para que haga una iteración continua en la que:

- En cada iteración:
  - Primero mostrará un listado con las figuras que hay actualmente en la lista *figuras*. Recorrerá dicho *ArrayList* y llamará al método *imprime* de cada figura, sacando la información que devuelva por pantalla
  - A continuación, mostrará un menú con las 4 opciones disponibles:
    - Crear una nueva figura
    - Borrar una figura existente
    - Guardar datos en fichero
    - Salir
  - Si el usuario elige *Salir*, se termina la ejecución (*System.exit(0)*)
  - Si elige *Crear una nueva figura*:
    - Se le pedirá que elija entre las figuras disponibles (Circulo, Rectángulo o Linea)
    - Una vez elegida, se le pedirá que introduzca uno a uno los valores de los campos para dicha figura (estos campos dependerán de la figura elegida).
    - Se recogerán todos ellos. Como se recogen como *String*, se deberán convertir luego al valor adecuado (es recomendable utilizar el método *Integer.parseInt(String valor)* para convertir una cadena a entero). Una vez recogidos, se creará un objeto del tipo elegido (Circulo, Rectangulo o Linea) y se añadirá a la lista *figuras* que tenemos como campo global
  - Si elige *Borrar una figura existente*:
    - Se mostrará un listado con las figuras que haya actualmente en la lista *figuras*
    - Se pedirá al usuario que elija el número de la figura que quiere borrar
    - Se borrará de la lista la figura de la posición elegida
  - Si elige *Guardar datos*:
    - Se utilizará la clase *IOFiguras*, y su método *guardaFiguras* para guardar la lista de figuras (habrá que convertirla a array), en el fichero que indicamos al arrancar el programa.

4. Compila y ejecuta la aplicación completa, verificando que funciona correctamente.
5. Una vez finalizado el ejercicio, y visto la forma en que está estructurado...

- ¿Qué ventajas ves en definir una clase padre que englobe a todas las figuras (ventajas en cuanto a tratamiento de las figuras, lectura/escritura, etc)?
- ¿Por qué no ha hecho falta en ningún momento del ejercicio (menos al añadir figuras a la lista) distinguir de qué figura se trata cada vez? Es decir, observa que siempre hemos tratado las figuras como "Figura", y no como casos particulares "Rectangulo", "Circulo" o "Linea". ¿A qué se debe esto? (AYUDA: tiene que ver con el uso de clases abstractas como superclases, y métodos abstractos dentro de ellas).

PARA ENTREGAR

- Todos los ficheros **.java** que se dan en la plantilla, debidamente rellenos y compilados para que la aplicación funcione.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.

ENTREGA FINAL DEL BLOQUE 2

- Como entrega **única** de TODOS los ejercicios del bloque 2, deberéis hacer un fichero ZIP (***bloque2.zip***), con una carpeta para cada sesión (*sesion6, sesion7, sesion8, sesion9* y *sesion10*), y dentro de cada carpeta copiar los ficheros que se os pidan en cada sesión.

Sesión 11

1. En esta sesión vamos a realizar dos ejemplos con AWT. El primero de ellos va a simular una calculadora sencilla, con este aspecto:

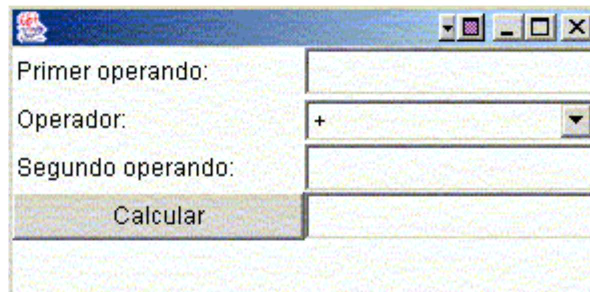


Figura 1. Apariencia de la calculadora

En el primer cuadro de texto ponemos el primer operando, luego elegimos la operación a realizar (suma '+', resta '-', o multiplicación '*'), y en el segundo cuadro de texto ponemos el segundo operando. Pulsando en *Calcular* mostraremos el resultado en el tercer cuadro de texto.

1. Antes de comenzar, lee los apartados **3.1.1** (*Introducción a AWT*) y **3.1.2** (*Gestores de disposición*) del tema 3 de teoría.
2. Echa un vistazo a la clase *Calculadora.java* que se proporciona en la plantilla de la sesión. Verás que es un subtipo de *Frame*, y tiene un constructor vacío, y un método *main* que crea un objeto de ese tipo y lo muestra (método *show*). Nos falta completar el constructor para definir la ventana que se mostrará.
3. Lo primero de todo es definir la ventana sobre la que van a ir los controles: crearemos una ventana de 300 de ancho por 150 de alto, con un gestor de tipo *GridLayout*, con 4 filas y 2 columnas (como se ve en la figura superior):

```
public Calculadora()
{
  setSize(300, 150);
  setLayout(new GridLayout(4, 2));
```

4. Después colocamos los componentes, por orden, en la rejilla: primero la etiqueta y el cuadro de texto del primer operando, después la etiqueta y el desplegable... etc:

```
public Calculadora()
{
  ...

  // Primer operando
  Label lblOp1 = new Label("Primer operando:");
  TextField txtOp1 = new TextField();
  add(lblOp1);
  add(txtOp1);
```

```
  // Operador
  Label lblOper = new Label ("Operador:");
  Choice operadores = new Choice();
  operadores.addItem("+");
  operadores.addItem("-");
  operadores.addItem("*");
  add(lblOper);
  add(operadores);

  // Segundo operando
  Label lblOp2 = new Label("Segundo operando:");
  TextField txtOp2 = new TextField();
  add(lblOp2);
  add(txtOp2);

  // Boton de calcular y cuadro de resultado
  Button btnRes = new Button ("Calcular");
  TextField txtRes = new TextField();
  add(btnRes);
  add(txtRes);
}
```

5. Llegados a este punto, compila y ejecuta el programa, para comprobar que no hay errores en el código, y para asegurarte de que el programa va a tener la misma apariencia que el de la figura 1 (lógicamente el programa aún no hará nada, sólo verás la ventana).
6. Lo que nos queda es "hacer que el programa haga algo". Para ello vamos a definir los eventos. Antes de seguir, lee el apartado **3.1.3** (*Modelo de eventos en Java*) del tema 3 de teoría.
   a. Definimos un evento sobre el botón, para que, al pulsarlo, tome los dos operandos y el operador seleccionado, y muestre el resultado en el cuadro correspondiente:

```
public Calculadora()
{
    ...
    // Evento sobre el botón
    btnRes.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            int op1, op2;
            try
            {
               // Tomar los dos operandos
                op1 = Integer.parseInt(txtOp1.getText());
                op2 = Integer.parseInt(txtOp2.getText());

               // Hacer la operacion seleccionada
                if
 (((String)(operadores.getSelectedItem())).equals("+"))
                    txtRes.setText("" + (op1 + op2));
                else if
 (((String)(operadores.getSelectedItem())).equals("-"))
                    txtRes.setText("" + (op1 - op2));
                   else if
 (((String)(operadores.getSelectedItem())).equals("*"))
```

```
                        txtRes.setText("" + (op1 * op2));
            } catch (Exception ex) {
                txtRes.setText("ERROR EN LOS OPERANDOS");
            }
        }
    });
}
```

b. El otro evento lo definimos sobre la ventana (el *Frame*) para hacer que se cierre y termine el programa cuando pulsemos el botón de cerrar:

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

c. Compila el programa... te dará errores de compilación.
d. Los errores del paso anterior se deben a que, si accedemos a un control desde dentro de un evento (como por ejemplo a los controles *txtOp1*, *txtOp2*, *txtRes* o la lista *operadores*, en el evento del botón), dichos controles no pueden ser variables locales normales. El error de compilación dice que deben declararse variables finales, u otra posibilidad es ponerlas como variables globales de la clase. Es decir, podemos hacer lo siguiente:

   Sustituir estas líneas:

```
public Calculadora()
{
    ...
    TextField txtOp1 = new TextField();
    ...
    Choice operadores = new Choice();
    ...
    TextField txtOp2 = new TextField();
    ...
    TextField txtRes = new TextField();
    ...
}
```

   Por estas:

```
public Calculadora()
{
    ...
    final TextField txtOp1 = new TextField();
    ...
```

```
    final Choice operadores = new Choice();
    ...
    final TextField txtOp2 = new TextField();
    ...
    final TextField txtRes = new TextField();
    ...
}
```

O bien colocarlas fuera del constructor, como variables globales:

```
public class Calculadora ...
{
    TextField txtOp1 = new TextField();
    Choice operadores = new Choice();
    TextField txtOp2 = new TextField();
    TextField txtRes = new TextField();

    public Calculadora()
    {
            ...
    }
    ...
}
```

7. Compila y ejecuta el programa. Prueba su funcionamiento con algunos ejemplos que se te ocurran. Observa también los ficheros *.class* que se generan: además del principal (*Calculadora.class*), aparecen dos más (*Calculadora$1.class* y *Calculadora$2.class*). ¿Sabrías decir qué son? (AYUDA: observa que aparecen tantos ficheros adicionales como eventos has definido en la aplicación...)
8. **(OPTATIVO)** Implementa otra clase *CalculadoraProfesional*, que tenga la apariencia de la calculadora de Windows, pero en versión reducida: en ella tendremos los números en diferentes botones, operaciones para sumar, restar, multiplicar y dividir, cada una en un botón, un botón "=" para calcular el resultado, y un cuadro de texto que simule la pantalla de la calculadora. El funcionamiento es el convencional: pulsando los botones, se actualizará el cuadro de texto, con los números introducidos, y realizando las operaciones indicadas.

2. Ahora vamos a construir otra aplicación desde cero. Para resumir los pasos que hemos dado en el ejercicio anterior, lee el apartado **3.1.4** (*Pasos generales para construir una aplicación gráfica con AWT*) del tema 3 de teoría.

La aplicación que vamos a construir ahora no es más que un bloc de notas "reducido", donde podremos cambiar algunas características de la fuente con que escribimos. La ventana tendrá una apariencia como la siguiente:

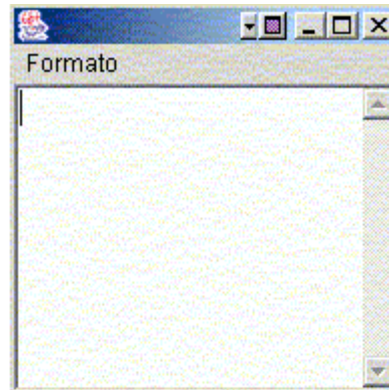


Figura 2. Apariencia de nuestro "bloc de notas"

Vemos que sólo hay un cuadro de texto grande donde escribir, y un menú llamado *Formato* donde irán todas las opciones para cambiar el formato de la fuente. Dichas opciones serán cambiar el color entre rojo y negro, y cambiar el estilo entre normal, cursiva y/o negrita.

1. Echa un vistazo a la clase *Formatos.java* que se proporciona en la plantilla de la sesión. Verás que es un subtipo de *Frame*, y tiene un constructor vacío, y un método *main* que crea un objeto de ese tipo y lo muestra (método *show*). Nos falta completar el constructor para definir la ventana que se mostrará.
2. Lo primero de todo es importar los paquetes necesarios: el de AWT y el de eventos:

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

public class Formatos extends Frame
{
  ...
```

3. Después establecemos el tamaño de la ventana y colocamos los componentes: el menú *Formato* con sus opciones, y el cuadro de texto:

```
public class Formatos extends Frame
{
  TextArea txt;
  boolean negrita = false;
  boolean cursiva = false;

  public Formatos()
  {
          setSize(200, 200);

          // Menú

          MenuBar mb = new MenuBar();

          Menu m = new Menu("Formato");

          MenuItem mi = new MenuItem("Color Negro");
```

```
          mi.addActionListener(this);
          m.add(mi);
          mi = new MenuItem("Color Rojo");
          mi.addActionListener(this);
          m.add(mi);

          CheckboxMenuItem cmi = new CheckboxMenuItem("Negrita");
          cmi.addItemListener(this);
          m.add(cmi);
          cmi = new CheckboxMenuItem("Cursiva");
          cmi.addItemListener(this);
          m.add(cmi);

          mb.add(m);

          setMenuBar(mb);

          // Area de texto

          txt = new TextArea();
          txt.setFont(new Font("Arial", Font.PLAIN, 16));
          add(txt, BorderLayout.CENTER);
  }
}
```

Las variables globales *negrita* y *cursiva* de momento no las utilizamos, pero más adelante las emplearemos para guardar en todo momento si están marcadas las opciones de letra negrita o cursiva, respectivamente. En el cuadro de texto establecemos una fuente Arial de 16 puntos, y sin estilo (sin negrita ni cursiva, indicado en la constante *Font.PLAIN*).

4. Observa que hay dos tipos de *menu items*: unos de tipo *MenuItem* para cambiar el color entre rojo y negro, y otros de tipo *CheckboxMenuItem* para establecer o no la cursiva y la negrita. Este tipo de items son de los que se marcan y se desmarcan cada vez que se pulsan.

   Observa también que definimos dos tipos de eventos sobre estos items de menú: un evento de tipo *ActionListener* sobre los *MenuItem*, y uno de tipo *ItemListener* para los *CheckboxMenuItem*. Esto se debe a que, al ser tipos diferentes de items, en AWT responden a dos tipos diferentes de eventos: los primeros son eventos de acción (al pulsar sobre el menú, es como si pulsáramos un botón), y los segundos son de cambio de estado (al pulsar sobre el item, es como si lo marcáramos o desmarcáramos cada vez).

   Por último, observa que los *ActionListener* e *ItemListener* que añadimos tienen como parámetro *this*, es decir, **la propia clase** debe implementar las interfaces *ActionListener* e *ItemListener*:

```
public class Formatos extends Frame implements ActionListener,
ItemListener
{
  ...
```

a. Para la interfaz *ActionListener*, debemos definir un método *actionPerformed*, que será el que se ejecute cuando pulsemos sobre la opción *Color Rojo* o *Color Negro* del menú:

```
public class Formatos extends Frame implements
ActionListener, ItemListener
{
    ...
    public Formatos()
    {
            ...
    }

    public void actionPerformed(ActionEvent e)
    {
            if (e.getActionCommand().equals("Color Negro"))
            {
                    txt.setForeground(Color.black);
            } else if (e.getActionCommand().equals(...)) {
                    txt.setForeground(Color.red);
            }
    }
```

¿Cómo hacemos para distinguir qué opción del menú se ha pulsado en el *actionPerformed*? ¿Qué método se utiliza para cambiar el color de la fuente de un cuadro de texto? ¿Qué error daría si no hubiésemos puesto la variable *txt* del área de texto como global?

b. Para la interfaz *ItemListener* debemos definir un método *itemStateChanged*, que será el que se ejecute cuando pulsemos sobre *Negrita* o *Cursiva* en el menú:

```
public class Formatos extends Frame implements
ActionListener, ItemListener
{
    ...
    public Formatos()
    {
            ...
    }
    ...
    public void itemStateChanged(ItemEvent e)
    {
            String item = (String)(e.getItem());
            if (item.equals("Negrita"))
                    negrita = !negrita;
            else if (item.equals("Cursiva"))
                    cursiva = !cursiva;

            txt.setFont(new Font("Arial",
                        (negrita?Font.BOLD:0) |
                        (cursiva?Font.ITALIC:0), 16));
    }
```

Observa que seguimos manteniendo, como en el constructor, una fuente Arial de 16 puntos. También observa que utilizamos las variables globales *negrita* y *cursiva* definidos al principio de la clase. Según si estos flags son verdaderos o falsos, establecemos una combinación de los mismos (con *Font.BOLD* y *Font.ITALIC*).

¿Cómo hacemos aquí para distinguir qué opción del menú se ha pulsado? ¿Qué método se utiliza para cambiar la fuente de un cuadro de texto? ¿Qué significa el segundo parámetro de dicho método?

5. Lo último que nos queda por hacer es definir, en el constructor, el evento sobre la ventana (el *Frame*) para hacer que se cierre y termine el programa cuando pulsemos el botón de cerrar:

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

6. Compila el programa para corregir posibles erratas, y ejecútalo para comprobar que funciona correctamente. Observa que, al cambiar el formato del texto, se cambia para TODO el texto del cuadro, no sólo para lo que vayamos a escribir a continuación.

PARA ENTREGAR

- Fichero **Calculadora.java** y **Formatos.java** con todas las modificaciones indicadas.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 12

1. Vamos a realizar un par de ejercicios con la librería Swing, para ver qué diferencias existen con AWT, y qué posibilidades nuevas ofrece.

1. Antes de comenzar con estos ejercicios, lee entero el apartado **3.2** (*Swing*) del tema 3 de teoría.
2. Echa un vistazo a la clase *JCalculadora.java* que se proporciona en la plantilla de la sesión. Verás que es una versión "resuelta" del ejercicio de la *Calculadora* propuesto en la sesión de AWT anterior. Lo que vamos a hacer ahora es transformar este ejemplo en un programa Swing, para ver qué diferencias hay.
3. Lo primero que hay que hacer es importar el paquete adecuado (además de los de AWT, que NO hay que quitar, porque el modelo de eventos y los gestores de disposición son los mismos).

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class JCalculadora ...
```

4. Después vamos cambiando los componentes de AWT por los correspondientes de Swing.
   a. En primer lugar, la clase ya no heredará de *Frame*, sino de su homólogo *JFrame*

   ```
   public class JCalculadora extends JFrame
   {
       ...
   ```

   b. Después sustituimos cada control de AWT por el correspondiente de Swing, es decir, las líneas:

   ```
   TextField txtOp1 = new TextField();
   ...
   Choice operadores = new Choice();
   ...
   TextField txtOp2 = new TextField();
   ...
   TextField txtRes = new TextField();
   ...
   Label lblOp1 = new Label("Primer operando:");
   ...
   Label lblOp2 = new Label("Segundo operando:");
   ...
   Label lblOper = new Label("Operador:");
   ...
   Button btnRes = new Button("Primer operando:");
   ...
   ```

   Por las correspondientes clases Swing:

```
JTextField txtOp1 = new JTextField();
...
JComboBox operadores = new JComboBox();
...
JTextField txtOp2 = new JTextField();
...
JTextField txtRes = new JTextField();
...
JLabel lblOp1 = new JLabel("Primer operando:");
...
JLabel lblOp2 = new JLabel("Segundo operando:");
...
JLabel lblOper = new JLabel("Operador:");
...
JButton btnRes = new JButton("Primer operando:");
...
```

c. Prueba a compilar y ejecutar la clase... dará error. ¿A qué se debe el error?
d. Como se explica en la parte de teoría, en Swing algunos métodos de *JFrame* no pueden ser accedidos directamente, como ocurría con *Frame* en AWT. Estos métodos son, entre otros, *setLayout* y *add*. Así, para solucionar el error anterior, deberás anteponer el método *getContentPane()* antes de cada método *setLayout* o *add* del *JFrame*:

```
getContentPane().setLayout(new GridLayout(4, 2));
...
getContentPane().add(lblOp1);
getContentPane().add(txtOp1);
...
getContentPane().add(lblOper);
getContentPane().add(operadores);
...
getContentPane().add(lblOp2);
getContentPane().add(txtOp2);
...
getContentPane().add(btnRes);
getContentPane().add(txtRes);
...
```

e. Compila y comprueba que el funcionamiento del programa es el mismo, aunque su apariencia sea distinta:

Figura 1. Apariencia de la calculadora en Swing

2. En este segundo ejercicio probaremos algunas cosas nuevas que ofrece Swing, partiendo de una aplicación ya hecha.

1. Echa un vistazo a la clase *JFormatos.java* que se proporciona en la plantilla de la sesión. Verás que es una versión en Swing del ejercicio de *Formatos* que se pedía en la sesión de AWT anterior.
2. Compílala y ejecútala para comprobar que funciona correctamente.
3. Antes de seguir, vamos a echar un vistazo a las cosas que cambian entre la aplicación en AWT y esta versión de Swing:
    - Observa, como en el ejercicio anterior, que cada clase AWT deja paso a su correspondiente de Swing:

```
Frame -> JFrame
TextArea -> JTextArea
MenuBar -> JMenuBar
Menu -> JMenu
MenuItem -> JMenuItem
CheckboxMenuItem -> JCheckBoxMenuItem
```

    - Un cambio importante lo tenemos en los controles de tipo *JCheckBoxMenuItem*. Observa que ahora ya no lanzan *ItemListeners*, sino *ActionListeners*, como los menús normales (*MenuItems*). Ello se debe a que la clase *JCheckBoxMenuItem* ya tiene disponible el método *addActionListener*, que no tenía su predecesora en AWT. Esto nos permite juntar todo el código de los menús en un sólo evento *actionPerformed*, y no tenerlo separado en dos (*actionPerformed* e *itemStateChanged*, como ocurría en AWT).
4. Sobre esta aplicación vamos a añadir dos cambios:
    - Un temporizador que cada 10 segundos guarde el texto que haya escrito en un fichero determinado (indicado por una variable)
    - Un botón con un icono de guardar (imagen *save.jpg* de la plantilla), que al pulsarlo nos abra un diálogo para que elijamos en qué fichero guardar el texto.

   Para el temporizador, primero importamos el paquete de entrada salida, para poder utilizar clases de acceso a ficheros:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
```

   También añadimos una variable global llamada *fichero* que tenga el nombre del fichero donde guardar el contenido del cuadro de texto:

```
public class JFormatos extends JFrame implements ActionListener
{
  JTextArea txt;
  boolean negrita = false, cursiva = false;
  String fichero = "guardar.txt";
  ...
```

Después, en el constructor, añadimos al final el temporizador, que utilizará un *PrintWriter* para volcar al fichero *fichero* el contenido del cuadro de texto:

```
public JFormatos()
{
  ...
  // Timer

  Timer tim = new Timer (10000, new ActionListener()
  {
          public void actionPerformed(ActionEvent e)
          {
                  try
                  {
                          PrintWriter pw =
                             new PrintWriter(
                                new FileWriter(fichero));
                          pw.println(txt.getText());
                          pw.close();
                  } catch (Exception ex) {}
          }
  });
  tim.setRepeats(true);
  tim.start();
}
```

¿Qué ocurriría si no pusiéramos el método *setRepeats(true)*?

Para el botón de guardar, primero declaramos una variable global que haga referencia al *JFrame* actual (luego veremos para qué sirve):

```
public class JFormatos extends JFrame implements ActionListener
{
  JTextArea txt;
  boolean negrita = false, cursiva = false;
  String fichero = "guardar.txt";
  JFrame frame = this;
  ...
```

Después añadimos el botón al final del constructor, con el icono *save.jpg* que se tiene en la plantilla, y su evento correspondiente:

```
public JFormatos()
{
  ...
  // Icono

  JButton btnSave =
    new JButton("Guardar", new ImageIcon("save.jpg"));
  btnSave.addActionListener(new ActionListener()
  {
          public void actionPerformed(ActionEvent e)
          {
                  JFileChooser jfc = new JFileChooser(".");
                  int res = jfc.showSaveDialog(frame);
                  if (res == JFileChooser.APPROVE_OPTION)
```

```
                    fichero =
                       jfc.getSelectedFile().getAbsolutePath();
          }
  });
  getContentPane().add(btnSave, BorderLayout.SOUTH);

}
```

Observa que para añadir el icono al botón, utilizamos un constructor de *JButton* con los parametros *JButton(String nombre, Icon icono*), y creamos el icono con la clase *ImageIcon* de Swing.

En cuanto al evento del botón, creamos un diálogo de tipo *JFileChooser*, que se abrirá en el directorio actual (indicado por "."). Observa que al llamar a *showSaveDialog* se abre un diálogo como los de *Guardar como...* de Windows, y este diálogo devuelve un entero, que es el resultado tras cerrarlo. Si dicho resultado es igual a la constante *JFileChooser.APPROVE_OPTION* indica que el usuario ha elegido un fichero, con lo que sólo nos queda tomar el fichero seleccionado (*getSelectedFile*) y su ruta absoluta (*getAbsolutePath*). A partir de entonces, el fichero donde guarde el temporizador será el nuevo que hemos asignado.

Observa también que la variable *frame* que hemos creado antes, la utilizamos para crear el diálogo, indicando de qué ventana principal depende. En realidad, podríamos haber utilizado cualquier componente de nuestra ventana principal.

Si hubiésemos querido abrir un fichero para leerlo, en lugar de para guardarlo... ¿qué método de *JFileChooser* habría sido más adecuado, en lugar de *showSaveDialog*? (consulta la API de esta clase para averiguarlo).

3. Para terminar esta sesión, vamos a dar un rápido vistazo a los Applets. Lee primero el apartado **3.3** (*Applets*) del tema 3 de teoría.

1. Echa un vistazo a la clase *CalcApplet.java* que se proporciona en la plantilla de la sesión. Verás que es otra copia del ejercicio de la *Calculadora* en AWT. Ahora vamos a convertirla en Applet.
2. Lo primero que haremos será importar el paquete para trabajar con applets, manteniendo los que ya hay de AWT, que los necesitaremos para los eventos, controles, y demás:

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

public class CalcApplet ...
```

3. Después hacemos que la clase herede de *Applet*, no de *Frame*:

```
public class CalcApplet extends Applet
{
  ...
```

4. A continuación, sustituimos el constructor por el método *init*, simplemente cambiando el nombre:

```
public CalcApplet()
{
  ...
```

por:

```
public void init()
{
  ...
```

5. Por último, eliminamos el método *main* entero (al ser un applet, no lo necesita), y compilamos el programa para depurar posibles erratas.
6. Lo que nos queda por hacer es una página HTML desde donde cargar el applet. En la plantilla se proporciona la página *CalcApplet.html*, con un *body* vacío, sólo hace falta añadirle una etiqueta APPLET con el applet que queramos ejecutar:

```
<html>
<body>

<APPLET CODE=CalcApplet.class WIDTH=400 HEIGHT=200>
</APPLET>

</body>
</html>
```

7. Abre la página desde cualquier navegador, y comprueba que el applet funciona correctamente. También puedes abrirla desde la herramienta *appletviewer* que viene con JDK, pasándole la página a abrir, desde la ventana de MS-DOS:

```
appletviewer CalcApplet.html
```

8. **(OPTATIVO)** Aunque te parezca increíble, se puede hacer que una misma aplicación AWT funcione como applet y como aplicación normal. Trata de modificar el applet que acabas de construir para hacer que se pueda cargar tanto a través de una página HTML como ejecutándolo automáticamente.

PARA ENTREGAR

- Fichero **Calculadora.java**

- Ficheros **JFormatos.java** y **save.jpg**, con la aplicación de *JFormatos* modificada: deberá tener el timer, el botón con el icono, y el evento para abrir el diálogo para elegir fichero.
- Fichero **CalcApplet.java** con la calculadora transformada en applet, y la página **CalcApplet.html** para poder cargar el applet.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 13

1. Vamos a realizar un ejercicio que nos permita practicar cómo dibujar en Java, y posteriormente realizar animaciones con esos dibujos.

1. Antes de comenzar, lee la introducción al apartado **3.4** (*Gráficos y animación*), y los subapartados **3.4.1** (*Gráficos en AWT*), y **3.4.2** (*Contexto gráfico: Graphics*) entero, incluyendo sus subapartados, del tema 3 de teoría.
2. Echa un vistazo a la clase *Ej1.java* que se proporciona en la plantilla de la sesión. Verás que es un subtipo de *JFrame*, de 300 de ancho por 300 de alto, y que tiene un *GridLayout* de 2 filas y 1 columna.

   Verás también que dentro de ese *GridLayout* coloca dos componentes. Uno de tipo *MiPanel* (una clase interna definida dentro de *Ej1*), y otro de tipo *MiCanvas* (también otra clase interna de *Ej1*). Observa que el primero es un subtipo de *JPanel*, y el segundo un subtipo de *Canvas*.
3. Compila y ejecuta la clase para comprobar que, de momento, no hay errores. Aparecerá una ventana de 300 x 300 donde no se ve nada (están añadidos el canvas y el panel, pero al estar vacíos no se ve nada).
4. Vamos a hacer que tanto en *MiCanvas* como en *MiPanel* se dibujen un rectángulo azul y un círculo verde. Para ello, vamos a cada clase y definimos dentro un método *paint* propio, que será el encargado de dibujar estos elementos:

```
...
class MiPanel extends JPanel
{
  ...

  public void paint(Graphics g)
  {
          g.setColor(Color.blue);
          g.fillRect(5, 0, 100, 25);
          g.setColor(Color.green);
          g.fillOval(125, 0, 100, 50);
  }
}

class MiCanvas extends Canvas
{
  ...
  public void paint(Graphics g)
  {
          g.setColor(Color.blue);
          g.fillRect(5, 0, 100, 25);
          g.setColor(Color.green);
          g.fillOval(125, 0, 100, 50);
  }
}
```

Observa que los dos (panel y canvas) dibujan las figuras en sus mismas coordenadas (relativas a cada uno), de forma que la apariencia en los dos es la misma.

5. Compila y ejecuta ahora la aplicación. Quedaría algo como esto:

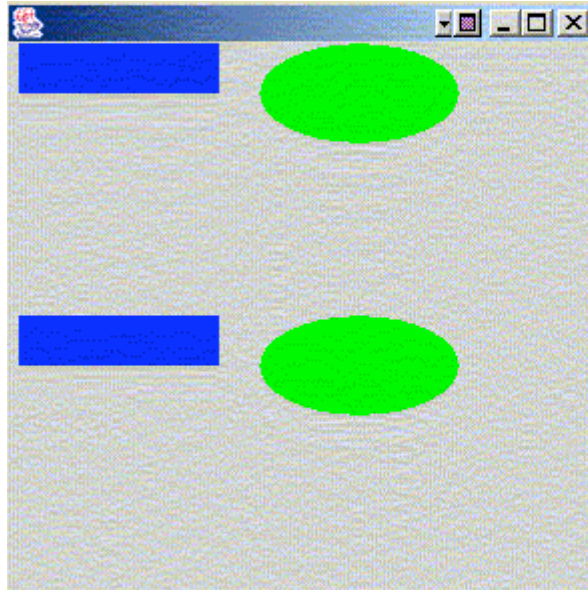


Figura 1. Apariencia de la aplicación de dibujar. El panel superior es MiPanel, y el inferior es MiCanvas

6. Vamos a introducir ahora algunas animaciones sobre estas figuras. Antes de seguir, lee entero el apartado **3.4** (*Animaciones*) del tema 3 de teoría.
7. Vamos a mover todas las figuras desde una coordenada y = 0 hasta otra y = 100.
    a. Lo primero, definimos una variable global en la clase *Ej1*, que guarde dicha coordenada y:

```
public class Ej1 extends JFrame
{
    MiPanel mp = new MiPanel();
    MiCanvas mc = new MiCanvas();
    int yObj = 0;
    ...
```

    b. Para hacer las animaciones, como se explica en el apartado de teoría, es importante hacerlas desde un hilo aparte, para no bloquear el programa principal. Así que definimos un hilo en nuestra clase principal, y hacemos que ésta implemente la interfaz *Runnable*:

```
public class Ej1 extends JFrame implements Runnable
{
    MiPanel mp = new MiPanel();
    MiCanvas mc = new MiCanvas();
    int yObj = 0;
```

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

Al final del constructor, iniciamos el hilo:

```
public Ej1()
{
    ...
    t.start();
}
```

Finalmente, definimos un método *run* en la clase *Ej1*, que haga las animaciones. Lo que va a hacer es incrementar la *yObj* de uno en uno, hasta 100, y repintar el panel y el canvas en cada iteración. Dormirá 100 ms entre cada iteración para permitir que el proceso principal (la ventana) siga su curso.

```
public void run()
{
    while (yObj < 100)
    {
            yObj++;
            mp.repaint();
            mc.repaint();
            try
            {
                    Thread.currentThread().sleep(100);
            } catch (Exception ex) {}
    }
}
```

Finalmente, nos queda modificar un poco los métodos *paint* de *MiCanvas* y *MiPanel*, para hacer que la coordenada Y donde se dibujan las figuras no sea fija, sino que sea la variable *yObj*:

```
...
class MiPanel extends JPanel
{
    ...

    public void paint(Graphics g)
    {
            g.setColor(Color.blue);
            g.fillRect(5, yObj, 100, 25);
            g.setColor(Color.green);
            g.fillOval(125, yObj, 100, 50);
    }
}

class MiCanvas extends Canvas
{
    ...
    public void paint(Graphics g)
    {
            g.setColor(Color.blue);
            g.fillRect(5, yObj, 100, 25);
            g.setColor(Color.green);
```

```
            g.fillOval(125, yObj, 100, 50);
    }
}
```

c. Prueba a compilar y ejecutar la clase... ¿Qué efecto no deseable observas en el panel superior (*MiPanel*)? ¿Y en el inferior (*MiCanvas*)?

8. Vamos a corregir los dos efectos no deseables que ocurrían en el paso anterior.
   - En el caso de *MiPanel*, el efecto de rastro se subsana definiendo un método *update* que llame a *paint*, y haciendo que en el método *paint* se limpie el área de dibujo antes de volver a dibujar (con un *clearRect*):

```
class MiPanel extends JPanel
{
    ...

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

    public void paint(Graphics g)
    {
            g.clearRect(0, 0, getWidth(), getHeight());
            g.setColor(Color.blue);
            g.fillRect(5, yObj, 100, 25);
            g.setColor(Color.green);
            g.fillOval(125, yObj, 100, 50);
    }
}
```

   - En el caso de *MiCanvas* el parpadeo (*flicker*) se corrige empleando la técnica del doble buffer. Definimos un campo de tipo *Image* en la clase *MiCanvas*, que utilizaremos como *backbuffer*. Haremos un método *update* que llame a *paint*, como antes, y luego en el paint dibujaremos sobre este *backbuffer*, y volcaremos el resultado entero en pantalla:

```
class MiCanvas extends Canvas
{
    Image backbuffer = null;

    public MiCanvas()
    {
    }

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

    public void paint(Graphics g)
    {
            if(backbuffer == null)
```

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

            // Dibujamos los gráficos en el backbuffer

            Graphics off_g = backbuffer.getGraphics();
            off_g.clearRect(0, 0, getWidth(), getHeight());
            off_g.setColor(Color.blue);
            off_g.fillRect(5, yObj, 100, 25);
            off_g.setColor(Color.green);
            off_g.fillOval(125, yObj, 100, 50);

            // Volcamos el backbuffer a pantalla

            g.drawImage(backbuffer, 0, 0,
                        getWidth(), getHeight(), this);
            g.dispose();
    }
}
```

Observa como hacemos el mismo dibujo, pero sobre los gráficos del *backbuffer (off_g)*, no sobre los del canvas (*g*). Después volcamos la imagen entera. Observa también que es necesario limpiar el *backbuffer* (con un *clearRect*) antes de dibujar, puesto que de lo contrario dejaríamos el rastro visto en *MiPanel*.

9. Compila y ejecuta la aplicación, para ver que funciona ya correctamente. ¿Por qué en el caso de *MiPanel* no ha sido necesario hacer el doble buffer, y sí lo hemos tenido que hacer en *MiCanvas*?
10. **(OPTATIVO)** Trata de añadir al código de este ejercicio controles por teclado, para hacer que las figuras se muevan en la dirección del cursor que se pulse (arriba, abajo, izquierda o derecha). Necesitarás añadir eventos de teclado (*KeyListener*), y manipular las figuras según lo que se pulse. Si no consigues hacerlo no te preocupes, en la siguiente sesión veremos ejemplos de cómo hacer este tipo de cosas.

PARA ENTREGAR

- Fichero **Ej1.java** con todos los cambios propuestos
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.

Sesión 14

1. En este ejercicio vamos a practicar con el modo a pantalla completa de Java, para ver que no resulta tan complicado como pueda parecer el trabajar con él.

1. Antes de comenzar, lee los subapartados **3.4.4** (*API de Java 2D*), y **3.4.5** (*Modo a pantalla completa*) del tema 3 de teoría.
2. Echa un vistazo a la clase *Ej2.java* que se proporciona en la plantilla de la sesión. Verás que es una versión reducida del ejemplo de animación hecho en la sesión anterior, donde sólo se muestra el Canvas, y dentro el rectángulo y círculo moviéndose.
3. Lo primero que vamos a hacer es pasar este *JFrame* a pantalla completa. Para ello es imprescindible elegir el modo gráfico que queremos (el *DisplayMode*). En la plantilla se os proporciona también un fichero llamado *DlgModos.java*. Echadle un vistazo, y veréis que es un tipo de cuadro de diálogo (*JDialog*). En él se obtienen los modos de pantalla compatibles con vuestro monitor, y se muestran en una *JList*, para que elijamos uno. Una vez elegido, pulsando el botón de *Aceptar* se cierra el cuadro de diálogo, quedándose el modo seleccionado en el campo *modoSeleccionado*.

   Vamos a incorporar este cuadro de diálogo a nuestro *JFrame*, para que se muestre antes de mostrar la ventana principal, y así elegir el modo gráfico que queramos. Añadimos al final del constructor lo siguiente:

```
public Ej2()
{
  ...
  // Tomamos el dispositivo grafico
  GraphicsEnvironment                    ge                    =
GraphicsEnvironment.getLocalGraphicsEnvironment();
  GraphicsDevice gd = ge.getDefaultScreenDevice();

  // Mostramos dialogo para elegir un modo grafico
  DlgModos dlg = new DlgModos(gd, this);
  dlg.show();

  // Activamos la pantalla completa, con el modo seleccionado
  gd.setFullScreenWindow(this);
  gd.setDisplayMode(dlg.modoSeleccionado);
}
```

   Compilad y ejecutad el programa. Es **IMPORTANTE** elegir un modo gráfico compatible con nuestra pantalla, pues realmente todos los que muestra no lo son. Para evitar pantallazos y cambios de configuración, podéis elegir el mismo modo que tenéis activo para Windows (normalmente, unos 1024 x 768 con 16 o 32 bits de profundidad de color). Si os fuese demasiado lento, podéis elegir uno de menor resolución (320 x 240, por ejemplo), aunque procurad mantener la misma profundidad de color que tengáis. En general, es cuestión de ir probando hasta que alguno se vea.

Veréis que el frame pasa a pantalla completa (aunque todavía se verá la barra superior de la ventana, eso lo solucionaremos luego).

4. Observad que, si elegís una resolución mayor que 320 x 240, el gráfico no ocupa toda la pantalla, sino la parte proporcional a esas dimensiones. Ello se debe a que estamos dibujando una imagen de 320 x 240 dentro de una ventana de resolución mayor. Lo que tenemos que hacer es escalar esa imagen a dibujar, para que ocupe toda la extensión del modo gráfico seleccionado.
   a. Primero declaramos dos variables globales que indicarán el ancho y alto del modo gráfico seleccionado:

```
public class Ej2 extends JFrame implements Runnable
{
    public static final int ANCHO = 320;
    public static final int ALTO = 240;
    MiCanvas mc = new MiCanvas();
    Thread t = new Thread(this);
    int yObj = 0;
    int anchoPantalla;
    int altoPantalla;
```

   b. Después, una vez elegido el modo de pantalla, asignamos la anchura y altura elegidas a dichas variables. Para ello, cuando se cierre el diálogo del modo gráfico, ponemos:

```
public Ej2()
{
    ...
    DlgModos dlg = new DlgModos(gd, this);
    dlg.show();
    anchoPantalla = dlg.modoSeleccionado.getWidth();
    altoPantalla = dlg.modoSeleccionado.getHeight();

    ...
}
```

   c. Finalmente, en el método *paint* de *MiCanvas*, hacemos que el backbuffer se dibuje escalado a las dimensiones de la pantalla, y no a ANCHO x ALTO como estaba antes:

```
public void paint(Graphics g)
{
    if(backbuffer == null)
            backbuffer = createImage(ANCHO, ALTO);

    // Dibujamos los gráficos en el backbuffer

    Graphics off_g = backbuffer.getGraphics();
    off_g.clearRect(0, 0, ANCHO, ALTO);
    ...

    // Volcamos el backbuffer a pantalla, segun su tamaño

    g.drawImage(backbuffer, 0, 0,
```

```
                anchoPantalla, altoPantalla, this);
    g.dispose();
}
```

Observad cómo la imagen del backbuffer siempre será de 320 x 240, tal y como la definimos en *createImage.* Esto nos permite dibujar independientemente de la resolución final, y luego nosotros escalaremos el resultado según la resolución que tengamos elegida.

5. La barra superior no la hemos quitado aún porque si no no habría forma de cerrar la ventana. Ahora lo que vamos a hacer es quitarla, y sustituir la animación automática por eventos de teclado, de forma que podamos nosotros mover las figuras (arriba y abajo, por ejemplo), desde las flechas del teclado

   Para quitar la barra, colocamos en el constructor (al principio, por ejemplo), el siguiente método:

```
public Ej2()
{
  this.setUndecorated(true);
  ...
}
```

que hace que la ventana no tenga decoración, y con eso se quita la barra.

Después, añadimos en ese mismo constructor un evento de teclado (*KeyListener*) que recogerá las siguientes pulsaciones de teclas:

- Si pulsamos la FLECHA HACIA ARRIBA del teclado, moverá las figuras hacia arriba (disminuirá su coordenada Y), hasta llegar al tope (Y = 0)
- Si pulsamos la FLECHA HACIA ABAJO del teclado, moverá las figuras hacia abajo (aumentará su coordenada Y), hasta llegar al tope (Y = altura del área de dibujo)
- Si pulsamos ESCAPE cerrará la aplicación.

Añadimos un evento con todo lo anterior:

```
public Ej2()
{
  ...
  addKeyListener(new KeyAdapter()
  {
          public void keyPressed(KeyEvent e)
          {
                  if (e.getKeyCode()==KeyEvent.VK_ESCAPE)
                  {
                          System.exit(0);
                  } else if (e.getKeyCode()==KeyEvent.VK_UP) {
                          if (yObj > 0)
```

```
                    yObj--;
            } else if (e.getKeyCode()==KeyEvent.VK_DOWN) {
                if (yObj < ALTO)
                    yObj++;
            }
        }
    });
}
```

¿Qué método utilizamos en el código para saber qué tecla se ha pulsado? ¿Con qué lo comparamos en cada caso para saber si se ha pulsado la tecla indicada?

Como ahora las figuras las movemos por teclado, quitamos la línea que aumenta la coordenada Y automáticamente en el método run, y hacemos que el bucle sea infinito:

```
public void run()
{
  while (true)
  {
        // Quitamos esta linea: yObj++;
        mc.repaint();
        try
        {
            Thread.currentThread().sleep(100);
        } catch (Exception ex) {}
  }
}
```

6. Compilad y ejecutad la aplicación, para comprobar que funciona correctamente. Probad los eventos del teclado para mover las figuras.
7. Por último, vamos a quitar las figuras que dibujábamos, y en su lugar vamos a colocar una imagen ya predefinida, que podamos mover con teclado. Tenéis una llamada *figura.png* en la plantilla.

    a. En primer lugar, definimos una variable global que guardará la imagen:

    ```
    public class Ej2 extends JFrame implements Runnable
    {
        public static final int ANCHO = 320;
        public static final int ALTO = 240;
        MiCanvas mc = new MiCanvas();
        Thread t = new Thread(this);
        int yObj = 0;
        int anchoPantalla;
        int altoPantalla;
        Image figura;
    ```

    b. Después, en el constructor, obtenemos la imagen, utilizando la clase *Toolkit* y luego su método *createImage*:

    ```
    public Ej2()
    {
    ```

```
    ...
    Toolkit tk = Toolkit.getDefaultToolkit();
    figura = tk.createImage("figura.png");
    ...
}
```

c. Finalmente, en el método *paint* de *MiCanvas*, quitamos los dibujos de las figuras, y simplemente dibujamos esta imagen:

```
public void paint(Graphics g)
{
    if(backbuffer == null)
            backbuffer = createImage(ANCHO, ALTO);

    // Dibujamos los gráficos en el backbuffer

    Graphics off_g = backbuffer.getGraphics();
    off_g.clearRect(0, 0, ANCHO, ALTO);
    off_g.drawImage(figura, 0, yObj,
                        figura.getWidth(this),
figura.getHeight(this), this);

    // Quitamos estas lineas, ya no las necesitamos:
    // off_g.setColor(Color.blue);
    // off_g.fillRect(5, yObj, 100, 25);
    // off_g.setColor(Color.green);
    // off_g.fillOval(125, yObj, 100, 50);

    // Volcamos el backbuffer a pantalla, segun el tamaño
de la misma

    g.drawImage(backbuffer,    0,    0,    anchoPantalla,
altoPantalla, this);
    g.dispose();
}
```

Vemos que dibujamos la figura en la coordenada X = 0, y variaremos su Y desde las flechas del teclado, como antes.

8. Compila y ejecuta ahora la aplicación. Comprueba que la figura se carga bien, y la puedes mover desde el teclado.
9. **(OPTATIVO)** Esta parte hazla únicamente si tienes tiempo y quieres, no es necesario entregarla:
   - El último apartado de teoría del tema 3 (apartado **3.4.6** (*Sonido y música. Java Sound*)), no lo hemos visto. Si te interesa, échale un vistazo.
   - En la plantilla se tiene también una imagen llamada *yogi.png*. Verás que tiene sprites (figuras) del oso Yogi, moviéndose a izquierda y derecha. Intenta sustituir el ejercicio que hemos hecho por otro donde la figura se mueva de izquierda a derecha (en lugar de arriba y abajo), y trata de animar un poco el movimiento cambiando la imagen a mostrar tras cada paso que se dé (eligiendo de entre las que aparecen en *yogi.png*). AYUDA: en la siguiente sesión veremos un ejemplo de cómo hacer esto, explicando la técnica a seguir en estos casos.

PARA ENTREGAR

- Fichero **Ej2.java** con todos los cambios propuestos (salvo los del punto 9, que son optativos), incluyendo también el fichero **DlgModos.java**.
- Fichero de texto **respuestas.txt** contestando a todas las preguntas formuladas.
- Contenidos optativos que hayas podido realizar (opcional)

Sesión 15

En esta última sesión vamos a hacer una aplicación que reúna varios de los conceptos importantes vistos hasta ahora. Debéis elegir **una** de las dos propuestas siguientes:

- Realizar una aplicación Swing que simule un **Paint simplificado**, donde sólo se podrán dibujar círculos, rectángulos y líneas, y se podrán guardar los datos en archivos, y recuperarlos. Si elegís esta opción, deberéis hacer el Ejercicio 1

- Realizar un **juego Java** a pantalla completa. Si elegís esta otra opción, deberéis hacer el Ejercicio 2.

Ejercicio 1. Vamos a construir un **Paint** en versión simplificada. En dicho *Paint* deberemos poder dibujar tres tipos de figuras: círculos, rectángulos y líneas rectas. También deberemos poder guardar figuras en ficheros, y recuperarlas después desde la aplicación.

Una apariencia general de la aplicación podría ser esta:

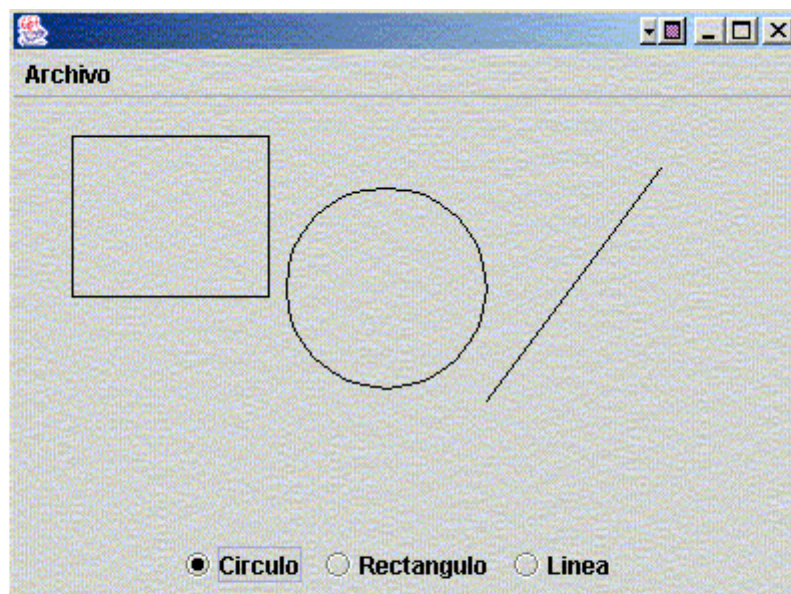


Figura 1. Apariencia de la aplicación Paint

Para hacer este ejercicio deberéis tomar el ejercicio que hicisteis en la **sesión 10**, donde se os pedía que, en modo texto, indicarais qué figuras geométricas añadir o quitar de una lista.

La estructura de clases y paquetes que vamos a necesitar es la misma que la de aquella sesión:

- **paquete geom**: con las figuras geométricas: contendrá la clase padre abstracta *Figura*, y las subclases *Circulo.java*, *Rectangulo.java* y *Linea.java* .
- **paquete io**: aquí tendremos la clase *EntradaTeclado* (que no utilizaremos en esta ocasión) y la clase *IOFiguras* para guardar y recuperar un conjunto de figuras en un fichero. Esta otra clase sí nos será útil.

La clase principal de aquella aplicación (la de la sesión 10) era *AplicGeom.* La borraremos, y sustituiremos por la versión en modo gráfico, la clase *JAplicGeom* que se os da en la plantilla, para que la completéis.

PREPARATIVOS PREVIOS

Antes de empezar, debéis realizar un cambio en la clase **geom.Figura**, y es añadirle un método abstracto más, llamado *dibuja*:

```
package geom;

import java.awt.Graphics;

public abstract class Figura implements java.io.Serializable
{
        public abstract String imprime();
        public abstract void dibuja(Graphics g);
}
```

Este método indicará cómo se debe dibujar cada subtipo de figura. Por tanto, deberéis añadir este método también en las tres subclases (**Circulo, Rectangulo** y **Linea**), indicando cómo se dibujaría cada una:

- En el caso del **Circulo**, habría que usar el método **drawOval** del parámetro *Graphics*, para indicarle que dibuje una elipse, que en realidad será un círculo centrado en (*x, y*), y con radio *radio*, es decir, una elipse que empiece en (*x - x/2, y - y/2*), y que tenga anchura = *radio*, altura = *radio*:

```
package geom;

import java.awt.Graphics;

public class Circulo extends Figura implements
java.io.Serializable
{
  ...
  public void dibuja(Graphics g)
  {
        g.drawOval(x - x/2, y - y/2, radio, radio);
  }
}
```

- En el caso del **Rectángulo**, indicamos con **drawRect** que dibuje un rectángulo con inicio en (*x1, y1*), y con anchura *x2 - x1* y altura *y2 - y1*. (NOTA: se asume que x2 > x1, e y2 > y1):

```
package geom;

import java.awt.Graphics;

public class Rectangulo extends Figura implements
java.io.Serializable
{
  ...
  public void dibuja(Graphics g)
  {
          g.drawRect(x1, y1, x2 - x1, y2 - y1);
  }
}
```

- En el caso de la **Linea**, indicamos con **drawLine** que dibuje una línea con inicio en (*x1, y1*) y fin en (*x2, y2*):

```
package geom;

import java.awt.Graphics;

public class Rectangulo extends Figura implements
java.io.Serializable
{
  ...
  public void dibuja(Graphics g)
  {
          g.drawLine(x1, y1, x2, y2);
  }
}
```

Con esto ya tenemos preparadas nuestras figuras para dibujarse. Nos faltará únicamente completar el programa principal (*JAplicGeom*) para indicar cómo dibujarlas, guardarlas y recuperarlas.

OBJETIVOS QUE DEBE CUMPLIR EL PROGRAMA

Se os da libertad para que cada uno implemente el programa como quiera, siempre que se cumplan los siguientes **requisitos**:

- Se deben poder dibujar los tres tipos de figuras: círculos, rectángulos y líneas
- Se deben poder guardar las figuras en un fichero, y recuperarlas después si se quiere.

NOTAS ACERCA DE LA IMPLEMENTACIÓN

Vamos a ver ahora cómo hacer los apartados que más os puedan costar.

a) Cómo construir la ventana

Como veis en la figura 1, la ventana de la aplicación tiene un área de dibujo en la parte superior, y tres botones de radio en la inferior, que indican qué figura dibujar, además de los menús.

Para la parte superior, podemos crearnos una clase interna *MiAreaDibujo*, que sea un subtipo de *JPanel*, y que podamos colocar en la parte superior para dibujar. Dicha clase tendrá un método *update* y otro *paint*, donde pondremos el código necesario para dibujar las figuras:

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

public class JAplicGeom extends JFrame
{
        class MiAreaDibujo extends JPanel
        {
                public MiAreaDibujo()
                {
                }

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

                public void paint(Graphics g)
                {
                        // Borrar el área de dibujo
                        g.clearRect(0, 0, getWidth(), getHeight());

                        // Dibujar las figuras (más adelante)...
                }
        }
}
```

Luego añadiríamos un objeto de este tipo en la clase principal:

```
...
public class JAplicGeom extends JFrame
{
        MiAreaDibujo mad;

        public JAplicGeom()
        {
                ...
                mad = new MiAreaDibujo();
        }
        ...
}
```

Para la parte inferior, podemos crear tres *JRadioButtons*, y añadirlos a un *JPanel*.

```
...
public class JAplicGeom extends JFrame
{
```

```
        MiAreaDibujo mad;
        JRadioButton btnCirculo;
        JRadioButton btnRectangulo;
        JRadioButton btnLinea;

        public JAplicGeom()
        {
                ...
                mad = new MiAreaDibujo();

                btnCirculo = new JRadioButton("Circulo", true);
                btnRectangulo = new JRadioButton("Rectangulo");
                btnLinea = new JRadioButton("Linea");

                JPanel panelBotones = new JPanel();
                panelBotones.add(btnCirculo);
                panelBotones.add(btnRectangulo);
                panelBotones.add(btnLinea);
        }
        ...
}
```

Finalmente, basta con colocar el área de dibujo arriba, y el panel de botones abajo:

```
...
public class JAplicGeom extends JFrame
{
        MiAreaDibujo mad;
        JRadioButton btnCirculo;
        JRadioButton btnRectangulo;
        JRadioButton btnLinea;

        public JAplicGeom()
        {
                ...
                mad = new MiAreaDibujo();

                btnCirculo = new JRadioButton("Circulo", true);
                btnRectangulo = new JRadioButton("Rectangulo");
                btnLinea = new JRadioButton("Linea");

                JPanel panelBotones = new JPanel();
                panelBotones.add(btnCirculo);
                panelBotones.add(btnRectangulo);
                panelBotones.add(btnLinea);

                getContentPane().add(mad, BorderLayout.CENTER);
                getContentPane().add(panelBotones, BorderLayout.SOUTH);
        }
        ...
}
```

Para los menús, definimos un *JMenuBar*, un *JMenu* llamado "Archivo", y dentro tres items: "Abrir" (para abrir un fichero de figuras), "Guardar" (para guardar las figuras actuales en fichero) y "Salir" (para cerrar la aplicación):

```
...
```

```
public class JAplicGeom extends JFrame
{
        ...
        public JAplicGeom()
        {
                ...
                JMenuBar mb = new JMenuBar();
                JMenu m = new JMenu("Archivo");

                JMenuItem mAbrir = new JMenuItem("Abrir");
                m.add(mAbrir);
                JMenuItem mGuardar = new JMenuItem("Guardar");
                m.add(mGuardar);
                JMenuItem mSalir = new JMenuItem("Salir");
                m.add(mSalir);

                mb.add(m);

                this.setJMenuBar(mb);
        }
        ...
}
```

Definimos un "main" donde creamos la ventana, le damos tamaño, le ponemos el evento de cerrarse, y la mostramos:

```
...
public class JAplicGeom extends JFrame
{
        ...
        public static void main(String[] args)
        {
                JAplicGeom jag = new JAplicGeom();

                jag.setSize(400, 300);

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

                jag.show();
        }
}
```

Probad a compilar y ejecutar el programa ahora, para depurar erratas y ver que se os muestra la ventana como la de la figura 1 (deberéis importar los paquetes necesarios en el fichero *JAplicGeom*).

Volver

b) Cómo gestionar las figuras

Para llevar un seguimiento de la lista de figuras, podemos definir una variable global que sea un **ArrayList**, donde almacenemos las figuras que vamos

dibujando. Sería buena idea también definir una variable **Figura** que almacene la figura que se está dibujando actualmente (antes de añadirla a la lista):

```
...
public class JAplicGeom extends JFrame
{
        ArrayList figuras = new ArrayList();
        Figura figActual = null;
        ...
}
```

Cuando empecemos a dibujar una figura, ésta se almacenará temporalmente en *figActual*, y cuando terminemos de dibujarla, la añadiremos a la lista *figuras*, y volveremos a dejar vacía *figActual*. En otros apartados se explica cómo cargar una lista de fichero, o cómo ir actualizando la lista con nuevas figuras dibujadas.

Volver

c) Cómo abrir y guardar archivos de figuras

En primer lugar vamos a ver qué hacer cuando elijamos *Archivo - Abrir* o *Archivo - Guardar*. Para leer o guardar una lista de figuras, tenemos los métodos *leeFiguras* y *guardaFiguras*, que implementamos en la clase *io.IOFiguras*, en la sesión 10.

Para **elegir el fichero del que leer**, o donde guardar, utilizaremos la clase **JFileChooser** de Swing.

De esta forma, si quiero **leer figuras de un fichero**, y guardarlas en el campo *figuras* definido antes, haré algo como:

```
JFileChooser jfc = new JFileChooser(".");
int result = jfc.showOpenDialog(this);
if (result == JFileChooser.APPROVE_OPTION)
{
        String fichero = jfc.getSelectedFile().getAbsolutePath();
        Figura[] figs = IOFiguras.leeFiguras(fichero);
        figuras = new ArrayList();
        if (figs != null)
                for (int i = 0; i < figs.length; i++)
                        figuras.add(figs[i]);
        repaint();
}
```

donde primero muestro el diálogo con *showOpenDialog* (*this* sería la ventana actual, que actúa como ventana padre del diálogo), recojo la respuesta, y si es APPROVE_OPTION quiere decir que he elegido un fichero. En ese caso cojo la ruta y el nombre del fichero (*getAbsolutePath*), y llamo a *IOFiguras.leeFiguras* para obtener las figuras. Luego las cargo en el campo *figuras*, y redibujo la ventana.

Por otro lado, si quiero **guardar las figuras en un fichero**, haré algo como:

```
JFileChooser jfc = new JFileChooser(".");
int result = jfc.showSaveDialog(this);
if (result == JFileChooser.APPROVE_OPTION)
{
        String fichero = jfc.getSelectedFile().getAbsolutePath();
        Figura[] figs = (Figura[])(figuras.toArray(new Figura[0]));
        IOFiguras.guardaFiguras(figs, fichero);
}
```

Es muy parecido a lo anterior: muestro el diálogo (ahora con *showSaveDialog*), recojo el fichero elegido, y llamo a *IOFiguras.guardaFiguras* para guardar la lista de figuras en el fichero.

NOTA: estos dos bloques tendréis que colocarlos donde toque, es decir, el primero lo pondréis donde se dé la orden de abrir un fichero, y el segundo donde se dé la orden de guardar.

Volver

d) Cómo responder a peticiones sobre menús y botones

Distinguimos dos tipos de eventos en el programa: los que se producen al elegir opciones de menú, o pulsar botones de radio, y los que se producirán cuando dibujemos figuras en el área de dibujo. Aquí trataremos los primeros.

Notemos que necesitamos distinguir 6 tipos de opciones diferentes, entre menús y botones de radio:

- Pulsar **Archivo - Abrir**
- Pulsar **Archivo - Guardar**
- Pulsar **Archivo - Salir**
- Pulsar botón de **Circulo**
- Pulsar botón de **Rectangulo**
- Pulsar botón de **Linea**

Podríamos definir un *ActionListener* para cada uno de estos elementos. Sin embargo, nos es más fácil hacer que la clase principal implemente *ActionListener*, y juntar todos los eventos en un solo método:

```
...
public class JAplicGeom extends JFrame implements ActionListener
{
        ...
        public JAplicGeom()
        {
                ...
                btnCirculo.addActionListener(this);
                btnRectangulo.addActionListener(this);
                btnLinea.addActionListener(this);
                mAbrir.addActionListener(this);
                mGuardar.addActionListener(this);
                mSalir.addActionListener(this);
        }
```

```
        public void actionPerformed(ActionEvent e)
        {
        }
}
```

En el *actionPerformed* que hemos añadido, pondremos el código que controle TODAS esas acciones. Para distinguir qué acción se ha pedido, el parámetro *ActionEvent* tiene un método *getActionCommand*, que permite comparar y ver de qué opción se trata:

```
        ...
        public void actionPerformed(ActionEvent e)
        {
                if (e.getActionCommand().equals("Abrir"))
                {
                        //... Qué hacer al elegir "Archivo - Abrir"
                } else if (e.getActionCommand().equals("Guardar")) {
                        //... Qué hacer al elegir "Archivo - Guardar"
                } else if (e.getActionCommand().equals("Salir")) {
                        //... Qué hacer al elegir "Archivo - Salir"
                } else if (e.getActionCommand().equals("Circulo")) {
                        // Marcar como seleccionado el Círculo
                        btnCirculo.setSelected(true);
                        btnRectangulo.setSelected(false);
                        btnLinea.setSelected(false);
                } else if (e.getActionCommand().equals("Rectangulo")) {
                        // Marcar como seleccionado el Rectangulo
                } else if (e.getActionCommand().equals("Linea")) {
                        // Marcar como seleccionada la Linea
                }
        }
        ...
```

El contenido de cada "if" o "else" deberéis rellenarlo vosotros.

Volver

e) Cómo responder a eventos sobre el área de dibujo

Cuando pinchemos con el ratón sobre el área de dibujo, empezará a dibujarse la figura que tengamos seleccionada de las tres de abajo. Cuando arrastremos el ratón con el botón pulsado, se redibujará la figura, con las nuevas dimensiones que le estemos dando. Finalmente, cuando soltemos el botón del ratón, se terminará de dibujar la figura.

Para controlar todo esto, debemos definir dos tipos de eventos de ratón sobre la clase *MiAreaDibujo*: uno de tipo *MouseListener* (para cuando pulsemos el botón y lo soltemos), y otro de tipo *MouseMotionListener* (para cuando arrastremos el ratón). También sería interesante definir 4 campos, *xIni, yIni, xFin, yFin* que marcarán los límites superior izquierdo e inferior derecho de la figura que dibujamos:

```
...
public class JAplicGeom extends JFrame implements ActionListener
```

```
{
	...
	class MiAreaDibujo extends JPanel
	{
		int xIni, yIni, xFin, yFin;

		public MiAreaDibujo()
		{
			this.addMouseListener(new MouseAdapter()
			{
				public void mousePressed(MouseEvent e)
				{
					// Qué hacer al pulsar el boton
				}

				public void mouseReleased(MouseEvent e)
				{
					// Qué hacer al soltar el boton
				}
			});

			this.addMouseMotionListener(
			   new MouseMotionAdapter()
			{
				public void mouseDragged(MouseEvent e)
				{
				    // Qué hacer al arrastrar el ratón
				}
			});
		}
	}
}
```

Al **pulsar el botón del ratón**, crearemos una figura del tipo que tengamos seleccionado:

```
public void mousePressed(MouseEvent e)
{
	xIni = xFin = e.getX();
	yIni = yFin = e.getY();

	if (btnCirculo.isSelected())
	{
		figActual = new Circulo(xIni, yIni, 0);
	} else if (btnRectangulo.isSelected()) {
		figActual = new Rectangulo(xIni, yIni, xFin, yFin);
	} else if (btnLinea.isSelected()) {
		figActual = new Linea(xIni, yIni, xFin, yFin);
	}
}
```

Observa que el parámetro *MouseEvent* tiene métodos *getX()* y *getY()* que indican qué coordenadas estamos pinchando.

Al **arrastrar el ratón**, iremos modificando los parámetros de la figura que habíamos creado, modificando sus coordenadas, anchura, o lo que toque:

```
public void mouseDragged(MouseEvent e)
```

```
{
        xFin = e.getX();
        yFin = e.getY();

        if (btnCirculo.isSelected())
        {
         ((Circulo)figActual).setX((xIni + xFin) / 2);
         ((Circulo)figActual).setY((yIni + yFin) / 2);
         ((Circulo)figActual).setRadio(Math.min(Math.abs(xIni - xFin),
                                             Math.abs(yIni - yFin)));
        } else if (btnRectangulo.isSelected()) {
         ((Rectangulo)figActual).setX1(Math.min(xIni, xFin));
         ((Rectangulo)figActual).setY1(Math.min(yIni, yFin));
         ((Rectangulo)figActual).setX2(Math.max(xIni, xFin));
         ((Rectangulo)figActual).setY2(Math.max(yIni, yFin));
        } else if (btnLinea.isSelected()) {
         ((Linea)figActual).setX2(xFin);
         ((Linea)figActual).setY2(yFin);
        }
        repaint();
}
```

Observad que para el círculo reajustamos el centro en la mitad de los límites de la figura, y ponemos como radio el valor mínimo entre la anchura y la altura. En el rectángulo ponemos como x1 e y1 los valores menores de coordenadas, y como x2 e y2 los mayores. Al terminar hacemos un repaint para que actualice el dibujo en pantalla.

Al **soltar el botón del ratón**, añadimos la figura a la lista, y ponemos a *null* la figura actual:

```
public void mouseReleased(MouseEvent e)
{
        xFin = e.getX();
        yFin = e.getY();
        figuras.add(figActual);
        figActual = null;
        repaint();
}
```

Volver

f) Cómo dibujar las figuras

Ya hemos visto cómo se va actualizando la lista de figuras a medida que vamos poniendo nuevas con el ratón. Ahora falta en el método *paint* decir cómo se dibujan. Basta con recorrer la lista de figuras, y llamar al método *dibuja* de cada una, pasándole los gráficos del panel. Finalmente, dibujamos la figura actual (*figActual*), si no es *null*, puesto que será la figura que estamos modificando actualmente:

```
public void paint(Graphics g)
{
        g.clearRect(0, 0, getWidth(), getHeight());
```

```
        for (int i = 0; i < figuras.size(); i++)
        {
                Figura f = (Figura)(figuras.get(i));
                f.dibuja(g);
        }

        if (figActual != null)
                figActual.dibuja(g);
}
```

Volver

ELEMENTOS OPTATIVOS

Podéis añadir (si queréis) los elementos optativos que queráis. Aquí os proponemos algunos:

- Como CURIOSIDAD, comprobad qué le pasa al menú si la clase *MiAreaDibujo* hereda de *Canvas* en lugar de *JPanel*.
- Permitir borrar figuras del panel
- Permitir elegir el color con que dibujar cada figura.
- Permitir dibujar un color de fondo para círculos y rectángulos, distinto al del contorno

---

Ejercicio 2. En este ejercicio vamos a hacer un **juego Java** que consistirá en lo siguiente: tendremos un bosque de fondo, y en la parte inferior de la pantalla, al oso Yogi, que se podrá mover de izquierda a derecha de la pantalla. El juego consistirá en que, desde arriba caerán una serie de alimentos, que Yogi deberá coger. Por cada alimento que coja antes de que llegue al suelo, sumará un punto, y por cada uno que no pueda recoger, perderá una vida.

La apariencia general del juego sería parecida a la siguiente:

Figura 2. Apariencia del juego Java

En la plantilla, en la carpeta *Yogi* tenéis una parte del juego hecha, para seguir desde ahí. Veréis una clase principal **Yogi.java**, que tendrá la parte importante del juego. Si la compiláis y ejecutáis (deberéis elegir el modo gráfico), veréis que simplemente aparece Yogi, y lo podréis mover de izquierda a derecha. Explicaremos a continuación cómo está hecho todo eso.

Deberéis completar el juego para que:

- Aparezca el bosque de fondo (imagen *bosque.jpg* de la plantilla)
- Caiga comida desde arriba (imagen *food.png* de la plantilla)
- Se actualicen los puntos y las vidas (imagen *vida.png* de la plantilla) según corresponda, cuando vayan cayendo los alimentos

CONSIDERACIONES SOBRE LA IMPLEMENTACION

Tenéis libertad para hacer el juego como queráis, siempre que se cumplan los puntos anteriores. Aquí vamos a contaros cómo está hecha la parte que se os da, y cómo poder hacer lo que queda.

Conviene que leáis el primer apartado de los siguientes antes que nada, y entendáis con él el código que se os da en la plantilla, para poderlo modificar.

a) Clases hechas en la plantilla

En la plantilla se os dan 3 clases:

- La clase **DlgModos** es igual que la vista en la sesión 14, y es un diálogo que nos permite elegir el modo gráfico para poner el juego a pantalla completa.
- La clase **SpriteYogi** controla el dibujo de la figura de Yogi.
  - Veréis en la plantilla un fichero llamado ***yogi.png*** que contiene animaciones de Yogi moviéndose a izquierda y derecha:

Figura 3. Animaciones de Yogi

  - La clase tiene dos constantes **ALTO** y **ANCHO**, que indican la altura altura y anchura de cada uno de los "sprites" de animación que hay en esta imagen (cada figura de Yogi mide ANCHO x ALTO)
  - Tenemos además los siguientes campos:
    - **sprites**: es un objeto de tipo *Image* que contiene la imagen *yogi.png* entera
    - **x**: es un campo que indica la coordenada x en la imagen de la figura (sprite) que se va a mostrar
    - **width** y **height** guardan la anchura y altura total de la imagen (con todos los sprites juntos)
  - En el **constructor** se carga la imagen en *sprites*, y se asignan la anchura y altura de la misma (que se conocen con algún programa de tratamiento de imágenes, o también se pueden sacar desde Java). Inicialmente el campo *x* vale 0, lo que indica que apunta a la primera figura (la de más a la izquierda).
  - El método **setFrame** nos permite cambiar de una figura de Yogi a otra. Simplemente modifica el campo *x*, haciendo que se mueva de ANCHO en ANCHO píxeles, para pasar de una figura a otra:

Figura 4. Moverse por los sprites de Yogi

Así podremos elegir en cada paso de animación qué figura dibujar.

- Finalmente, la clase **Yogi** es la ventana principal.

CONSTANTES Y CAMPOS:

- Tiene las constantes **ANCHO** y **ALTO** que definen la anchura y altura de la ventana de dibujo (luego se redimensionará para ocupar toda la ventana), y además la constante **INC_MOV_YOGI** que indica cuántos píxeles se mueve Yogi a izquierda o derecha en cada pulsación de flecha.
- Tiene los campos **ancho** y **alto** que guardarán la resolución real que elijamos de pantalla, a fin de escalar luego el dibujo de ANCHO x ALTO a la pantalla completa de *ancho x alto*.
- El campo **backbuffer** se utiliza para dibujar la pantalla en doble buffer, y luego volcar el resultado
- El campo *sprn* será el dibujo de Yogi en pantalla. Es un objeto de tipo *SpriteYogi*, donde cada vez iremos eligiendo qué sprite o frame dibujar de Yogi.
- Los campos **secuenciaDer** y **secuenciaIzq** indican las secuencias de animación de Yogi a derecha e izquierda, respectivamente. Veréis que hay varios números repetidos consecutivamente. La idea es la siguiente: cuando pulsemos la flecha derecha la primera vez, el sprite a cargar de los de la figura 4 es el *secuenciaDer[0]*, es decir, el 0. La siguiente vez que pulsemos la flecha será *secuenciaDer[1]* (vuelve a ser el 0)... y a medida que vayamos pulsando la tecla derecha irá pasando al siguiente número de frame.

  La **pregunta** es... ¿por qué no se ha puesto {0, 1, 2} directamente, para pasar de un frame a otro? La respuesta es sencilla: si pasamos de un frame a otro tan instantáneamente, la animación resulta demasiado rápida, y no da la impresión de que Yogi camine, sino más bien parece que tenga epilepsia :-). Por eso se repiten frames durante un tiempo, para hacer la animación más lenta.

  La misma filosofía se aplica para animar a Yogi hacia la izquierda, con el array *secuenciaIzq*.
- El campo **numFrame** apuntará a la posición del array de movimientos (izquierdo o derecho) que se mostrará en cada paso de animación.
- El campo **xIni** indica la posición X actual de Yogi. Se irá actualizando cada vez que lo movamos.
- Por último, tenemos un hilo **t** que será el encargado de hacer las animaciones.

MÉTODOS:

- En el **constructor** hacemos algo parecido a lo de la sesión 14: obtenemos el dispositivo gráfico (**GraphicsDevice**) y el modo de pantalla seleccionado a través de la clase **DlgModos**. Por último ponemos el modo a **pantalla completa** seleccionado. Enmedio de todo eso, inicializamos el **sprite de Yogi** (campo *sprn*) y definimos los **eventos de teclado** (*KeyListener*) para que:
  - Al pulsar la **flecha IZQUIERDA** se actualice la xIni, se coja el frame correspondiente del array *secuenciaIzq*, y se pase a apuntar al frame siguiente, para la siguiente animación.

  - Al pulsar la **flecha DERECHA** se actualice la xIni, se coja el frame correspondiente del array *secuenciaDer*, y se pase a apuntar al frame siguiente, para la siguiente animación.
  - Al pulsar **ESCAPE** se salga del juego.
- Tenemos un método **update** que directamente llama al **paint** de la ventana. Dicho *paint* hace el doble buffer, y utiliza un método auxiliar llamado **dibuja** para dibujar en dicho doble buffer. Finalmente vuelca dicho buffer a la pantalla.
- El método **dibuja** será el que utilicemos para dibujar todos los componentes del juego. Ahora lo que hace es **limpiar el área** de dibujo (de ANCHO x ALTO) y **dibujar la imagen de Yogi** que toque. Pero **¿cómo se dibuja sólo la figura que nos interese de toda la tira?**. Lo que hacemos es definir un área de recorte (*setClip*) y poner la imagen sobre la pantalla de forma que en dicha área "caiga" la parte de la tira que nos interese:

Figura 5.Cómo animar a Yogi

En el código (método *dibuja*), esto queda reflejado como:

```
// Dibujar a Yogi
Shape clip = g.getClip();
```

```
g.setClip(xIni,ALTO-
SpriteYogi.ALTO,SpriteYogi.ANCHO,SpriteYogi.ALTO);
g.drawImage(sprn.sprites, xIni - sprn.x, ALTO - SpriteYogi.ALTO,
            sprn.width, sprn.height, this);
g.setClip(clip);
```

El área de recorte la definimos en el *xIni* que marca la posición X actual de Yogi. Marcamos un recorte de lo que mida cada sprite de animación (*SpriteYogi.ANCHO x SpriteYogi.ALTO*), y dibujamos la imagen de sprites entera, pero desplazada para que el sprite que toque "caiga" dentro del área de recorte. Antes de recortar, nos guardamos el area de recorte previa (para no machacarla), con un *getClip*, y después de dibujar en el área de recorte la volvemos a establecer con un *setClip*, para poder seguir dibujando en toda la ventana (si no lo restablecemos, cualquier cosa que dibujemos fuera del área de recorte no se verá).

- El método **run** lo ejecuta el hilo. Lo que hace es constantemente llamar a repaint() (que llamará a *update*, éste a *paint*, y éste a *dibuja*) para forzar un nuevo paso de animación, y actualizar la posición de los elementos en pantalla.

Volver

b) Cómo cargar el fondo

Cargar el fondo del juego (imagen *bosque.jpg* de la plantilla) es muy sencillo. Podéis definir un campo global de tipo *Image* para guardar la imagen:

```
...
public class Yogi extends JFrame implements Runnable
{
        Image fondo = null;
        ...
```

Luego en el constructor, ayudándonos de la clase *Toolkit* le cargamos la imagen JPG:

```
...
public Yogi()
{
        ...
        Toolkit tk = Toolkit.getDefaultToolkit();
        fondo = tk.createImage("bosque.jpg");
}
```

Tenéis que tener en cuenta que, si la imagen es muy grande (ocupa muchos KB), puede que tarde en cargarla, y puede que se cargue la ventana sin tener la imagen ya cargada. Para asegurarnos de que no va a abrir la ventana hasta

tenerla cargada, podemos utilizar la clase **MediaTracker**, y le pasamos la imagen, para que no siga hasta tenerla cargada:

```
...
public Yogi()
{
        ...
        Toolkit tk = Toolkit.getDefaultToolkit();
        fondo = tk.createImage("bosque.jpg");

        MediaTracker mt = new MediaTracker(this);
        mt.addImage(fondo, 1);
        try {
                mt.waitForAll();
        } catch (Exception e) { e.printStackTrace(); }
}
```

lo que hacemos es añadirle la imagen al *MediaTracker*, y luego hacer que espere hasta que todas las imágenes que tenga añadidas (en este caso, el fondo) estén listas.

Finalmente, en el método *dibuja* dibujamos la imagen ANTES de dibujar cualquier otra cosa (para dejarla detrás de todo). Podemos dibujarla justo después de limpiar el área de dibujo:

```
public void dibuja(Graphics g)
{
        // Limpiar area
        g.clearRect(0, 0, ANCHO, ALTO);

        // Dibujar fondo
        g.drawImage(fondo, 0, 0, ANCHO, ALTO, this);

        ...
```

Volver

c) Cómo dibujar los alimentos

Tenéis una imagen llamada *food.png* en la plantilla, con tres tipos diferentes de alimentos. Cada uno de ellos tiene un ancho y un alto de 20 píxeles.

Figura 6. Tipos de alimentos disponibles

De lo que se trata es de que construyáis una clase *SpriteComida*, similar a *SpriteYogi*, donde defináis los tamaños de cada alimento, y de la imagen entera, y podáis elegir con un *setFrame* qué comida mostrar. De hecho, el código será muy parecido a *SpriteYogi*, pero cambiando los tamaños de cada sprite y de la imagen.

Luego, en la clase principal, creáis un objeto de este tipo, y le ponéis la comida que queráis de la imagen (podéis generar qué sprite poner aleatoriamente, por ejemplo). De forma que dicho sprite se vaya moviendo en cada iteración hacia abajo, hasta llegar al suelo. En ese momento se creará otro sprite diferente y se repetirá el proceso.

Para mover la comida, conviene que actualicéis la posición Y del sprite dentro del método *run*, y luego en *dibuja* sólo tengáis que ver qué Y tiene, y dibujarlo como se hace con el sprite de Yogi. Notad que en este caso no hay que cambiar de frame en cada paso de animación: se mantiene siempre la misma comida hasta que termina de caer, y luego se crea otro sprite, con la misma comida u otra diferente. Así el *setFrame* para la comida sólo se ejecutaría al crearla, y no se variaría durante su caída.

Volver

d) Cómo dibujar las vidas

Para las vidas tenéis una imagen que se llama *vida.png* en la plantilla. Basta con que la asignéis a un campo *Image* en la clase principal, y lo dibujéis en la parte superior derecha, tantas veces como vidas se tengan.

Figura 7.Imagen para las vidas

Volver

e) Cómo dibujar los puntos

Como véis en la figura 2, los puntos son una cadena de texto en la parte superior izquierda. Para dibujar texto en un objeto *Graphics* tenéis el método *drawString*. Simplemente hay que pasarle la cadena a dibujar, y las coordenadas (X,Y) inferiores izquierdas.

Para aseguraros de que todo el texto os va a caer dentro de la pantalla, y no se os va a cortar nada por arriba, conviene que hagáis uso de la clase **FontMetrics**, que toma las medidas de la fuente establecida, y permite ver qué altura va a tener el texto.

```
public void dibuja(Graphics g)
{
        ...
        Font fuentePuntos=new Font("Arial",Font.BOLD|Font.ITALIC,15);
        FontMetrics fmPuntos = g.getFontMetrics(fuentePuntos);
        g.drawString("Puntos: ...los que sean... ", x, y);
}
```

Tomamos en este caso una fuente Arial de 15 puntos, negrita y cursiva. Luego obtenemos su *FontMetrics*, y dibujamos la cadena, donde *x* e *y* se deben

sustituir por la coordenada inferior izquierda desde donde dibujar la cadena. La *x* en nuestro caso sería 0 (dibujamos desde el borde izquierdo), y la *y* dependerá de la altura que vaya a tener el texto dibujado. Para obtener eso, el objeto *FontMetrics* tiene un método llamado *getAscent()* que nos indica qué altura va a tener el texto. Basta con dibujarlo con la Y en ese valor, para que al subir como mucho llegue a *y = 0*, y no se pase por arriba:

```
g.drawString("Puntos: ...los que sean... ", 0, fmPuntos.getAscent());
```

Volver

ALGUNAS CUESTIONES OPTATIVAS

Si tenéis tiempo (y ganas), podéis añadirle, de forma opcional, cualquier cosa que consideréis al juego. **Aunque no las vayáis a hacer, os recomendamos que las leáis**, para tener claro algunas mejoras que podrían ser importantes en cualquier juego.

Por ejemplo, os sugerimos las siguientes:

- Tratad de hacer un juego basado en estados o etapas: inicialmente el juego estará en un estado INICIAL, donde se le pedirá al usuario que pulse una tecla para empezar las animaciones. Después se pasará al estado JUGANDO, donde se desarrollará el juego normal, y cuando se acaben las vidas se pasará al estado FINALIZADO, donde no se seguirá animando más, y se pedirá al usuario que salga del juego. El paso de un estado a otro vendrá dado por pulsaciones de teclas, o porque se acaben las vidas. La animación del *run* se hará siempre y cuando el estado sea JUGANDO.
- En la plantilla tenéis también una imagen llamada *yogi_1.png*, que contiene un montón de sprites de Yogi. Podéis por ejemplo coger los que necesitéis para hacer que Yogi pueda **saltar**, y añadirlos al juego.
- También tenéis otra imagen llamada *food_1.png* con muchas más comidas que poder utilizar, si queréis.
- Aumentad la complejidad del juego haciendo que la comida caiga cada vez más deprisa, o de lugares más lejanos o aleatorios.
- Podéis aumentar también la complejidad haciendo que caigan varias comidas a la vez, o que caiga una comida sin esperar a que la anterior haya terminado.

PARA ENTREGAR

- Todos los ficheros que compongan la aplicación que hayáis elegido implementar (incluídos los que se os den en la plantilla)

ENTREGA FINAL DEL BLOQUE 3

- Como entrega **única** de TODOS los ejercicios del bloque 3, deberéis hacer un fichero ZIP (***bloque3.zip***), con una carpeta para cada sesión (*sesion11, sesion12, sesion13, sesion14* y *sesion15*), y dentro de cada carpeta copiar los ficheros que se os pidan en cada sesión.