



FORMACIÓN Y TECNOLOGÍAS JAVA
UNIVERSIDAD DE ALICANTE

Curso de Programación en Lenguaje Java

Julio 2005

Curso de Programación en Lenguaje Java

Sesión 1: Comenzando a programar en Java	1
Sesión 2: Programación Orientada a Objetos en Java I	27
Sesión 3: Programación Orientada a Objetos en Java II	43
Sesión 4: Excepciones	51
Sesión 5: Hilos	61
Sesión 6: Utilidades	69
Sesión 7: Colecciones	85
Sesión 8: Entrada/Salida	101
Sesión 9: Swing	117

Sesión 1. Comenzando a programar en Java

El lenguaje Java

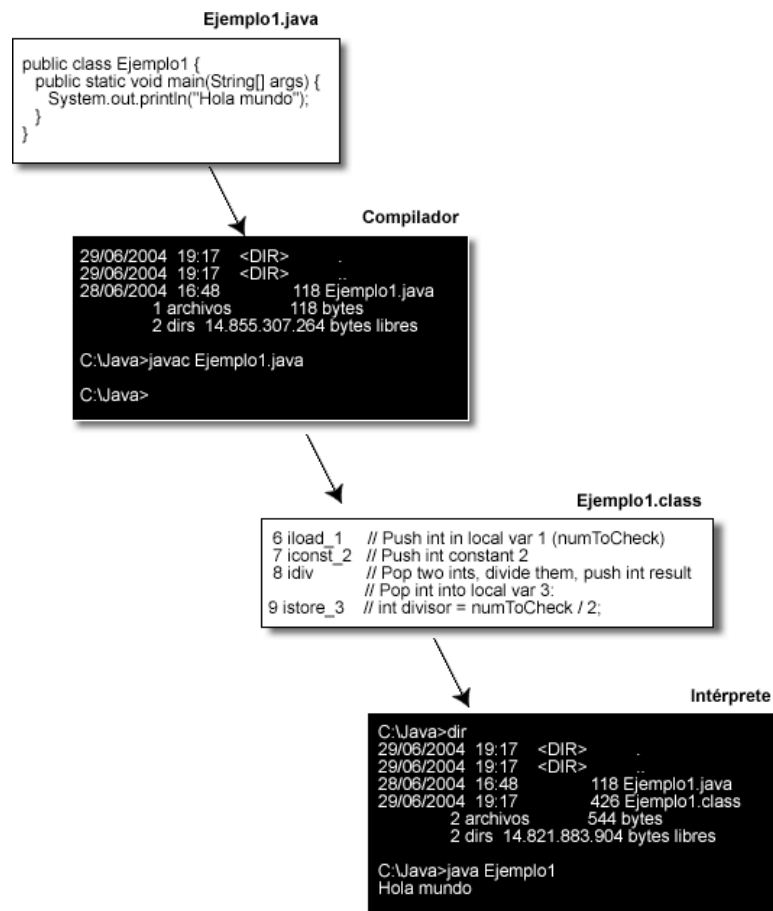
Java es un lenguaje de programación orientado a objetos que nace en 1995 en un grupo de trabajo de Sun Microsystems liderado por James Gosling. Con una sintaxis similar a C y una filosofía orientada a objetos como Smalltalk, muy pronto se convirtió en un lenguaje muy popular. Hoy es uno de los lenguajes más usados en todos los entornos de programación, desde móviles hasta servidores UNIX, pasando por dispositivos multimedia y ordenadores personales.

La plataforma Java consta de tres partes fundamentales:

- El lenguaje de programación mismo.
- La máquina virtual de Java, que permite la portabilidad en ejecución.
- Un conjunto de APIs, bibliotecas estándar para el lenguaje.

Una de las ideas fundamentales de Java es la portabilidad. Una de las frases más conocidas sobre Java es la máxima WORE (Write Once Run Everywhere, escribe una vez y ejecuta en cualquier lugar). Para conseguir esto es fundamental el carácter interpretado del lenguaje. A diferencia de otros lenguajes como C o C++, el compilador de Java no produce código máquina de un procesador dado, sino que produce un código intermedio o *bytecode* que es ejecutado por una máquina virtual java (JVM). Esta máquina virtual garantiza la portabilidad. Cualquier sistema operativo para el que se haya desarrollado una máquina virtual Java (como Windows, Linux, Mac OS X, Solaris, ...) puede ejecutar cualquier programa Java compilado a *bytecodes*. Y no sólo eso, sino que también existen máquinas virtuales para dispositivos como teléfonos móviles o decodificadores de televisión digital. En estas plataformas también se pueden ejecutar los programas Java.

La siguiente figura muestra el proceso de compilación y ejecución de un programa Java:



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

El tirón final en la popularidad de Java se debió a Internet y la World Wide Web. La JVM se incorporó a los navegadores web más populares de la época (Mosaic y Netscape) y permitió la distribución por Internet de programas Java y la ejecución de estos programas en los navegadores web. Estos programas se denominaron Applets. Hoy en día los applets han caído en desuso, pero Java sigue presente en Internet en forma de JavaScript, páginas JSP, servlets, Enterprise JavaBeans, servicios web y otras muchas tecnologías.

El lenguaje de programación Java ha ido sufriendo distintos cambios a lo largo de su historia. La última versión estable se denomina Java 5.0. El desarrollo del lenguaje y de las API (Interfaces de Programación) se realiza por comités en los que participan, a parte de Sun Microsystems, representantes de las más importantes empresas tecnológicas, así como usuarios destacados del lenguaje. El proceso en el que se propone un tema de innovación en el lenguaje, se forma un grupo de trabajo y se elabora y aprueba una propuesta final se denomina JCP (Java

Community Process).

Otra característica fundamental de Java es el extenso número de APIs (definidas por clases y paquetes) desarrolladas para el lenguaje. Estas APIs se definen y desarrollan bajo el modelo de JCP y, cuando están maduras, se incorporan en el lenguaje. Por ejemplo, en la edición 5.0 de Java se incorporan paquetes:

- javax.crypto: conjunto de clases e interfaces para operaciones de criptografía.
- java.sql: API para acceder y procesar datos almacenados en una fuente de datos (normalmente una base de datos relacional).
- java.awt y javax.swing: clases e interfaces para escribir programas de interfaz gráfica.

Se han definido varias distribuciones de Java orientadas a distintos tipos de aplicaciones. Todas ellas tienen como núcleo el lenguaje de programación, pero se diferencian en las APIs incluidas. Las más importantes son:

- J2ME: Java 2 Micro Edition. Distribución de Java orientada a aplicaciones para dispositivos móviles como teléfonos móviles o PDAs.
- J2SE: Java 2 Standard Edition. Edición de Java orientada a aplicaciones de escritorio en ordenadores personales.
- J2EE: Java 2 Enterprise Edition. Distribución de Java orientada a aplicaciones distribuidas que se ejecutan en red.

Ejercicio 1: Búsqueda en Web sobre Tecnologías Java

1. Busca en Google el nombre de 3 dispositivos (distintos de un ordenador) en donde se puedan ejecutar programas Java. Describe brevemente cada uno de ellos.
2. Busca en java.sun.com y encuentra 4 APIs que se incluyan en Java 5.0 (distintas de las que hemos comentado anteriormente). Indica el nombre del paquete (o paquetes) y descríbe cada uno en un par de frases.
3. Busca en internet el nombre de 2 JSR de Java que están en desarrollo en la actualidad o que se han cerrado recientemente. Puedes usar como palabra de búsqueda JSR y "public review". El estado de "Public Review" de una especificación se refiere a una especificación que se ha cerrado y que se ofrece al público para recibir comentarios.
4. Escribe las respuestas en un fichero de texto, con el nombre de respuestas.txt.

Compilación y ejecución de programas Java

Para programar en Java es necesario el compilador y la máquina virtual. Ambos programas están incluidos en el JDK (Java Development Kit, kit de desarrollo de Java) que distribuye Sun. Existen otros compiladores y máquinas virtuales desarrollados por otras empresas como IBM o BEA weblogic. Pero todos ellos son compatibles con las especificaciones de Java.

El nombre del compilador Java es javac, y el de la máquina virtual java es java. Se les invoca desde línea de comando de la siguiente forma:

```
> javac Ejemplo.java  
> java Ejemplo
```

Los programas javac y java se encuentran en el JDK. En el ejercicio siguiente vamos a instalarlos en el ordenador.

Ejercicio 2: Instalación del JDK de Sun

1. Instala el JDK (Java Developer Kit) 5.0 (se encuentra en la sección de recursos de la web del curso). Es el conjunto de herramientas que Sun proporciona para desarrollar en Java. Acepta las opciones por defecto que te propone la instalación.
2. Comprueba que la instalación ha funcionado correctamente. Para ello abre una ventana del sistema operativo y prueba a ejecutar el compilador de Java.

```
> C:\Archivos de programa\Java\jdk1.5.0_04\bin\javac
```

3. Para hacer más sencillas las llamadas a los comandos, actualiza la variable de entorno PATH, añadiendo el directorio C:\Archivos de programa\Java\jdk1.5.0_04\bin

```
> set PATH=%PATH%;\Archivos de programa\Java\jdk1.5.0_04\bin
```

4. Comprueba que todo funciona bien escribiendo:

```
> java -version
```

Los ficheros fuente de Java tienen la extensión .java. Cada fichero .java define una clase Java pública (y, posiblemente, más de una clase privada usada por la clase pública). Los ficheros de código intermedio generados por la compilación tienen la extensión .class. La compilación de un fichero .java puede generar más de un fichero .class, si en el fichero .java se define más de una clase.

Un detalle muy importante es que el nombre del fichero .java debe corresponder con el nombre de la clase pública definida en él. Si no es así, el compilador dará un error de compilación.

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. Ya explicaremos más adelante qué es un método estático.

Veamos el primer ejemplo, el típico programa *Hola mundo* que escribe una frase en la salida estándar:

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

El programa debe escribirse en un fichero que tenga el mismo nombre que la clase que se define. En este caso, debe llamarse *HolaMundo.java*. En la clase se define un método estático llamado main que es el que se ejecuta cuando se invoca la clase con la máquina virtual java usando el comando:

```
>java HolaMundo
```


Vamos a probarlo en el siguiente ejercicio.

Ejercicio 3: Primeros programas Java

1. Crea el directorio `sesion1`, 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 `HolaMundo.java` (es importante la "H" mayúscula).

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

Compila la clase, lista los ficheros del directorio y verás que se ha generado el fichero `HolaMundo.class` que contiene los *bytecodes*:

```
> javac HolaMundo.java
> dir

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

El fichero `HolaMundo.class` es el que se va a ejecutar en la máquina virtual java (JVM). Llama a la JVM pasándole como argumento el nombre de la clase (`HolaMundo`) y verás como aparece el mensaje "Hola mundo" por la salida estándar. No es demasiado espectacular, pero has conseguido compilar y ejecutar tu primer programa Java:

```
> java HolaMundo
Hola mundo
```

2. Vamos ahora a ver un ejemplo un poco más atractivo. Escribe ahora el siguiente programa, y sávalo 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);
    }
}
```

3. Compila el programa y dará un mensaje de error. ¿Cuál es el error? ¿Cómo corregirlo? (contestalo en el mismo fichero respuestas.txt del ejercicio anterior). 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.



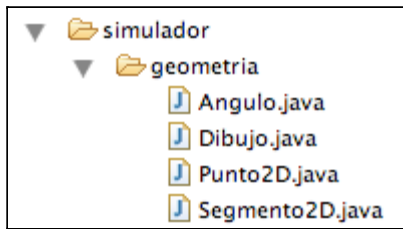
No intentes entender el código. En la última sesión del curso veremos una introducción a la programación de interfaces de usuario en Java.

Definición de paquetes

Las clases Java se organizan en paquetes. Un paquete contiene un conjunto de clases y también puede contener otros paquetes. Es una estructura similar a la de los directorios y ficheros. De hecho, ya que las clases y paquetes deben residir en la estructura de ficheros del sistema operativo, los paquetes se definen como directorios del sistema operativo.

Existe el convenio de que los nombres de los paquetes deben tener letras minúsculas, mientras que los nombres de las clases deben comenzar siempre con mayúscula.

En la figura siguiente vemos que las clases Punto2D, Segmento2D, Angulo y Dibujo residen en el directorio geometria que a su vez reside en el directorio simulador. Esto define el paquete simulador.geometria.



Hay que indicar en todos los ficheros java en qué paquete se encuentran usando la palabra `package`. Las rutas de paquetes se separan con puntos ".". Así, por ejemplo, para declarar que la clase `Punto2D` se encuentra en el directorio `geometria` del directorio `simulador` hay que comenzar el fichero `Punto2D.java` con la siguiente línea:

```
package simulador.geometria;
```

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

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

Para referirse a una clase hay que indicar su ruta completa, incluyendo el nombre del paquete en el que reside. Así, el nombre de la clase `Punto2D` será:

```
simulador.geometria.Punto2D
```

Ejercicio 4: Uso de paquetes

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

```
package sesion1;

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

Compila el programa. Verás que aunque el directorio `sesion1` no esté creado, la compilación no produce ningún error. Intenta ahora ejecutar el programa de la misma forma que lo hiciste en el ejercicio anterior. Verás que ahora sí que se produce un error, porque la clase `HolaMundo` no reside en el paquete declarado.

Para solucionar el error deberás crear la estructura de directorios del paquete. En este caso sólo hay que crear el directorio `sesion1` y colocar allí la clase compilada. Puedes hacerlo (en MS-DOS) con:

```
> javac HolaMundo.java
```

```
> mkdir sesion1
> copy HolaMundo.class sesion1
> del HolaMundo.class
```

Por último, para ejecutar el programa recién compilado hay que llamar a la clase indicando todo su camino (sesion1.HolaMundo).

```
> java sesion1.HolaMundo
```

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:

Borra los directorios que acabas de crear, incluyendo el fichero `HolaMundo.class` y vuelve a compilar el fichero `HolaMundo.java`

```
> deltree sesion1
> javac HolaMundo.java -d .
> java sesion1.HolaMundo
```

2. Un par de preguntas que debes contestar en el fichero `respuestas.txt`:

- Supongamos que quisieras crear la clase `HolaMundo` en el paquete `plj.sesion1`. ¿Qué tendrías que hacer?
- ¿Qué tipos de clases pondrías en el paquete `plj`? Escribe 2 nombres de clases que piensas que podrían ir bien en este paquete.

La variable de entorno CLASSPATH

¿Dónde colocar el directorio simulador para que el intérprete de la máquina virtual Java lo encuentre? El directorio se puede colocar en un directorio estándar de la distribución de Java en el que la máquina virtual busca las clases. O también se puede colocar en un directorio cualquiera e indicar a la máquina virtual que en ese directorio se encuentra un paquete Java. Para ello se utiliza la variable del entorno `CLASSPATH`. Si la variable no está definida, se buscan las clases en el directorio actual y en el directorio estándar de clases de Java (el cual depende del sistema operativo en el que estemos ejecutando Java).

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

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

Ejercicio 5: Definiendo la variable CLASSPATH

1. Supongamos que estamos en el directorio C:\cursojava, en el que se encuentra el paquete sesion1 y, dentro de él, la clase compilada HolaMundo.class. Probemos de nuevo que la JVM encuentra la clase:

```
C:\cursojava> java sesion1.HolaMundo
Hola mundo
```

La clase se encuentra porque, al no estar definida la variable de entorno CLASSPATH se toma el directorio actual como el directorio en el que buscar las clases. Supongamos que nos movemos al directorio padre y que intentamos ejecutar desde allí la clase sesion1.HolaMundo. Obtendremos un mensaje de error:

```
C:\cursojava> cd ..
C:\> java sesion1.HolaMundo
<MENSAJE DE ERROR>
```

Si actualizamos el CLASSPATH de la siguiente forma, java podrá encontrar de nuevo la clase HolaMundo

```
C:\> set CLASSPATH=C:\cursojava
C:\> java sesion1.HolaMundo
HolaMundo
```

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

- ¿Qué sucede si actualizas el CLASSPATH de la siguiente forma?:

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

¿Funciona la llamada a la clase HolaMundo desde el directorio C:\cursojava?

- ¿Se arregla la llamada cuando actualizas el CLASSPATH así?:

```
set CLASSPATH=.;C:\
```

¿Por qué?

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

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

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

```
C:\cursojava> set CLASSPATH=C:\cursojava\ejemplo.jar
```

```
C:\cursojava> deltree sesion1
C:\cursojava> java sesion1.HolaMundo
Hola mundo
```

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

- ¿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`?

Entornos integrados de desarrollo para Java

Java es un lenguaje enormemente complicado (cuando vas a desarrollar un gran proyecto) como para poder dominarlo usando el *Notepad*. Para programar en Java es conveniente utilizar un IDE (entorno integrado de desarrollo, en inglés).

Hace algunos años, era difícil encontrar IDEs de código abierto que compitieran con los IDEs comerciales que distribuían Borland y otras casas. Hoy es al revés. Existen dos IDEs de libre distribución que tienen una potencia difícilmente alcanzable por el resto. Se trata de *Eclipse* y *NetBeans*.

Eclipse está desarrollado por un consorcio en el que se incluyen IBM y otras importantes empresas relacionadas con el mundo de Java para entornos UNIX.

NetBeans está desarrollado por Sun y es el IDE *oficial* propuesto por Sun.

Nosotros vamos a utilizar en el curso Eclipse, por considerar que es algo más robusto y por que está más extendido que NetBeans. Pero cada vez más gente le está echando un ojo a NetBeans. Sobre todo los que hacen aplicaciones J2EE (Java Enterprise Edition) y desarrollan páginas JSP, servlets o componentes distribuidos.

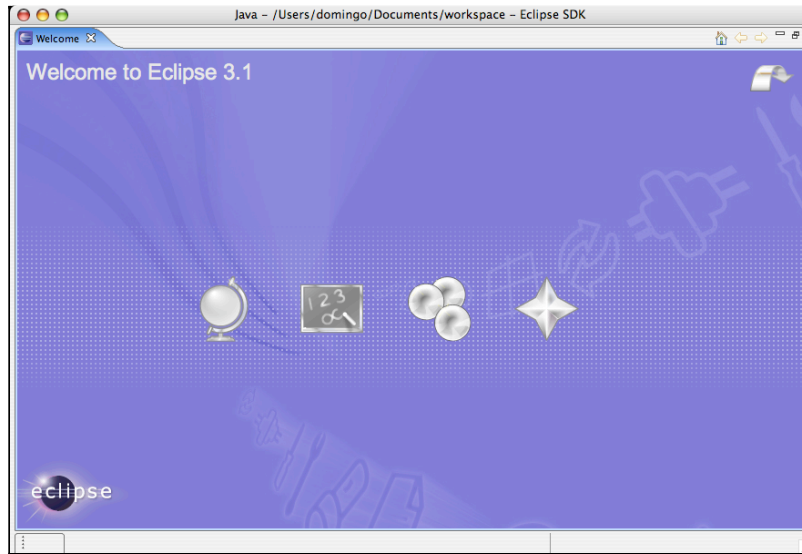
Ejercicio 6: Instalación de Eclipse

1. Descomprime el fichero Eclipse (se encuentra en la sección recursos de la web del curso) 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 o, si estás en Linux, actualizando el PATH para incluir el nuevo directorio recién creado `eclipse/bin` y ejecutando el comando `eclipse`.

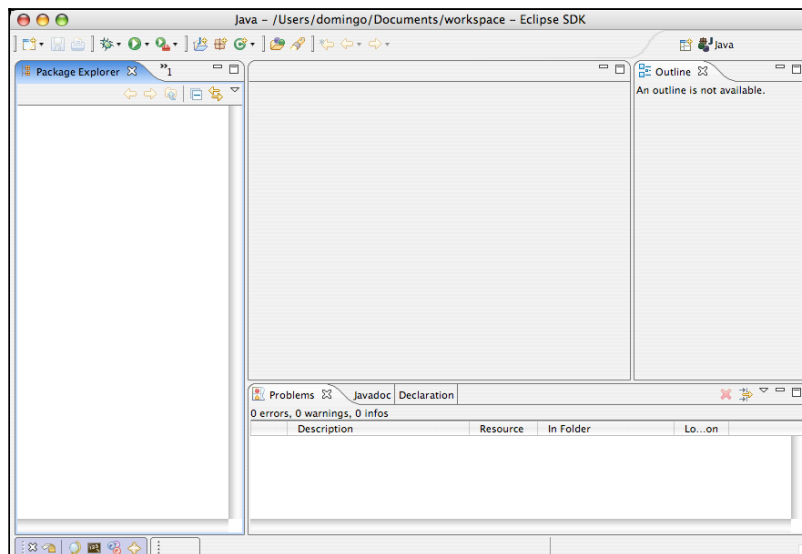
Eclipse es un programa escrito en Java y debes tener instalado el JDK, como has hecho en un ejercicio anterior, para que funcione. Cuando Eclipse arranca por primera vez pide el directorio de trabajo. Puedes aceptar el que te sugiere (`eclipse\workspace`), o indicarle alguno propio.

3. Cuando aceptes el directorio de trabajo, aparecerá la siguiente ventana de

presentación.



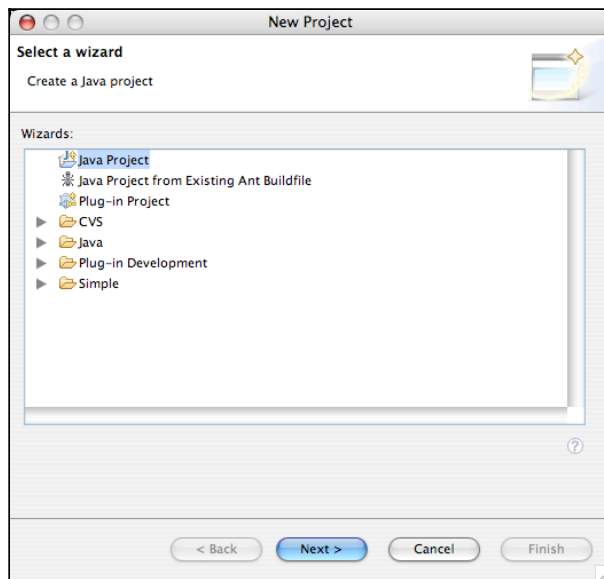
Desde esta ventana de presentación puedes ver tutoriales, ejemplos o entrar directamente en la zona de trabajo (esquina superior derecha). Haz esto último para comenzar a trabajar en Eclipse. Podrás volver a la ventana de presentación en cualquier momento con la opción *Help > Welcome*). Quédate por ahora en la zona de trabajo:



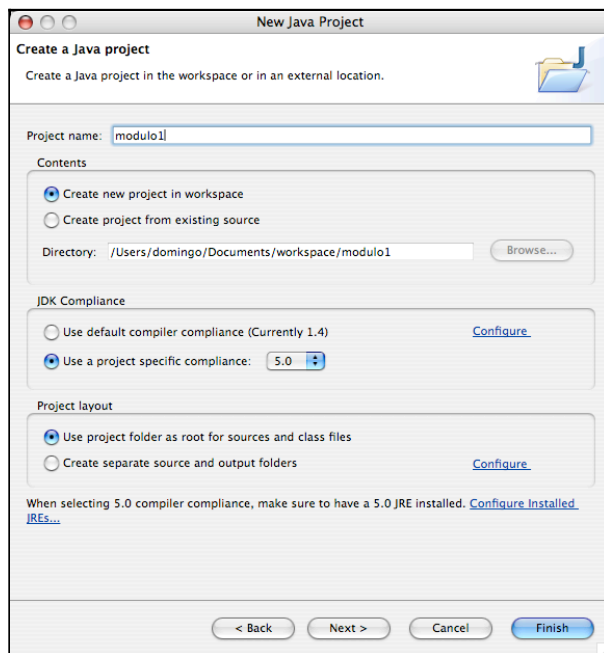
Una vez instalado Eclipse vamos a compilar en él las clases que hemos compilado antes en línea de comando.

Ejercicio 7: Compilación y ejecución en Eclipse

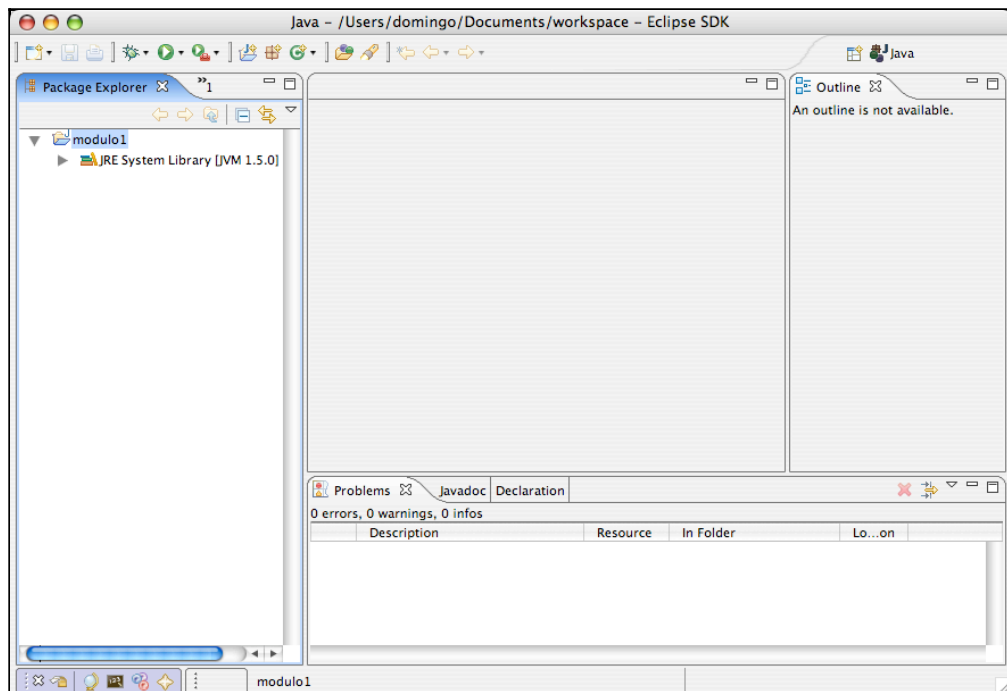
1. Crea un nuevo proyecto Java. Para ello situa el ratón en la *vista* Package Explorer, pulsa el botón derecho y escoge la opción New project (también puedes crearlo con la opción *File > New > Project*). Selecciona el *wizard* Java Project:



Llama al proyecto modulo1 y escoge como JDK compliance la opción 5.0. Esto significa que el editor Java de Eclipse va a aceptar código de la versión 5.0 de Java (también llamada versión 1.5). La mayoría del código que vamos a escribir en el curso es propiamente de Java 1.4.2, pero al usar esta opción también podemos escribir código que sea específico de la última versión de Java.



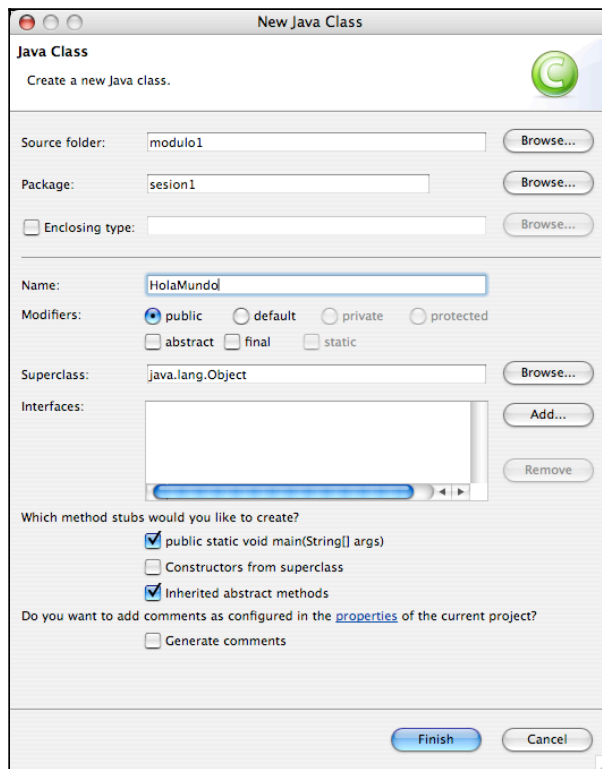
Pulsa Finish y se creará el proyecto.



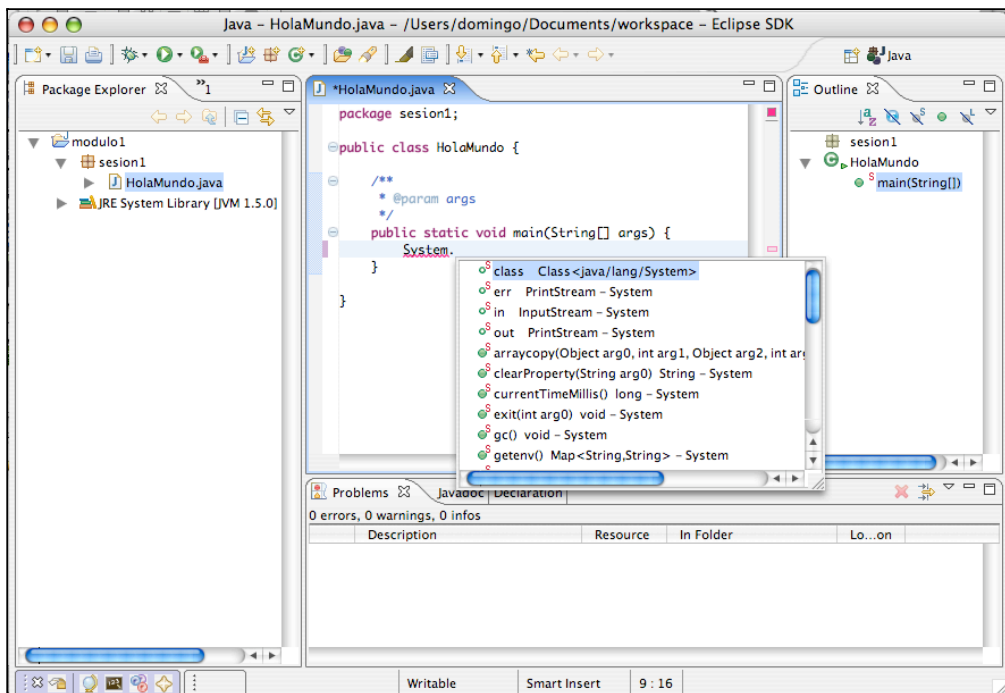
Abre otra ventana con el explorador del sistema de ficheros (o un terminal) y lista los ficheros del directorio de trabajo de Eclipse. Verás que se ha creado un nuevo directorio con el mismo nombre que el proyecto que acabas de crear.

En un proyecto se agrupan un conjunto de clases y paquetes. Vamos a usar este proyecto recién creado para todos los ejercicios de este primer módulo, separándolos en un paquete por sesión.

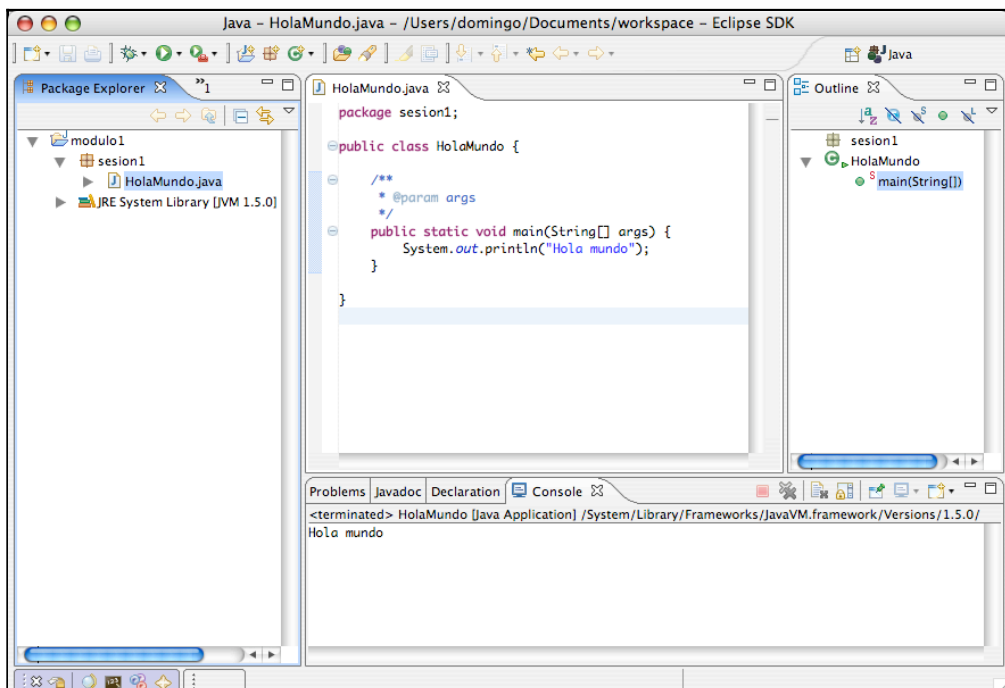
2. Crea el paquete sesion1. Para ello pincha el paquete modulo1, pulsa el botón derecho y escoge New > Package. Aparecerá una ventana de diálogo en la que deberás escribir el nombre del paquete. Escribe sesion1.
3. Vamos ahora a crear la clase HolaMundo que ya conocemos. Selecciona el paquete recién creado y, con el botón derecho, escoge la opción New > Class. Dale el nombre HolaMundo 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 un esqueleto de la clase. Al crear la clase, Eclipse introduce automáticamente el nombre del paquete en el que se encuentra y también añade 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 HolaMundo del ejercicio anterior. Durante la escritura podrás comprobar que el editor chequea la sintaxis, indicándote si hay un error, y también te ayuda a completar las llamadas a los métodos.

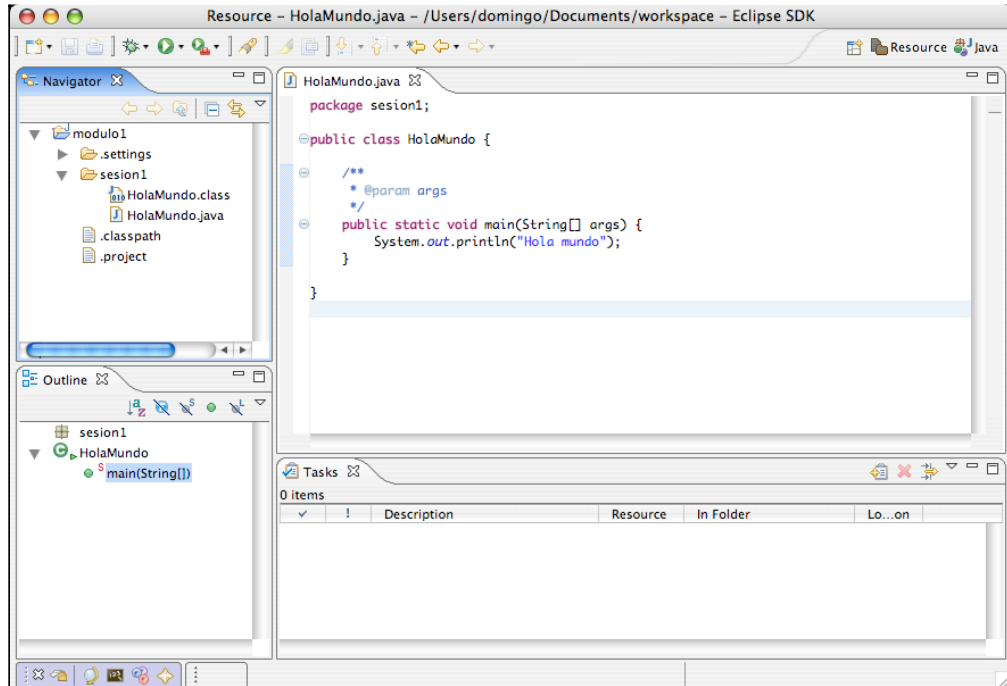


Una vez escrita la clase, vamos a ejecutarla. Para ello, pincha el nombre de la clase en la vista del explorador de paquetes y, con el botón derecho, escoge la opción Run As > Java Application (también puedes seleccionar en el menú la opción Run As > Java Application). Aparecerá la vista Console con la salida de la ejecución del programa.



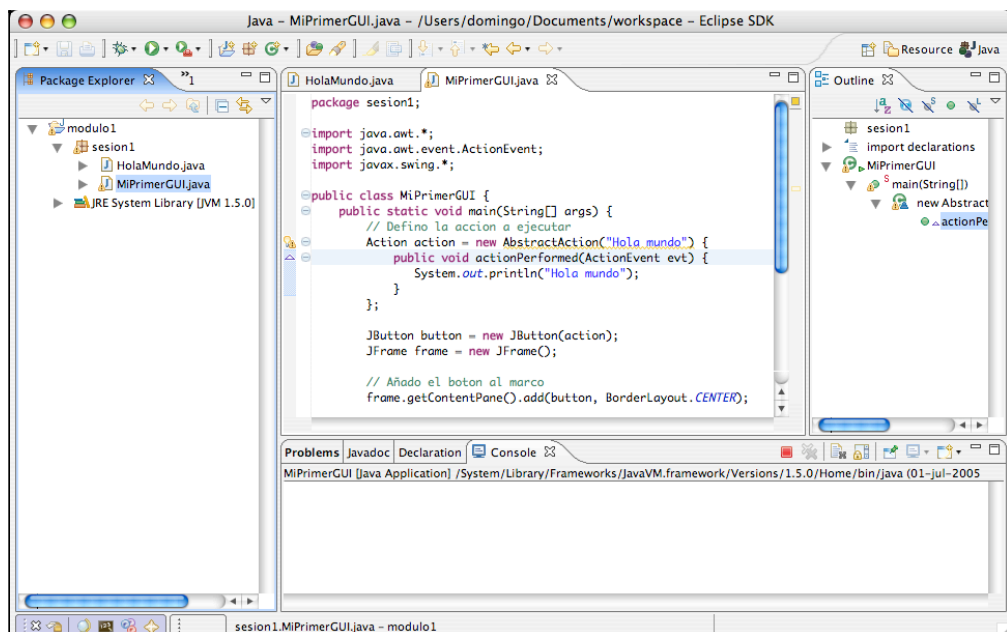
4. Si examinas en el explorador del sistema de ficheros o en una terminal el directorio modulo1 creado en el espacio de trabajo de Eclipse, verás que se encuentra el fichero HolaMundo.java y el fichero compilado HolaMundo.class. Eclipse ha compilado automáticamente el fichero java. Puedes examinar los

ficheros del sistema proyecto cambiando a la perspectiva Resource. Para ello debes pinchar en el icono junto al nombre Java que hay en la esquina superior derecha. Selecciona la opción Other... y escoge la perspectiva Resource. Podrás explorar en la vista de explorador de paquetes y aparecerá algo como esto:



Para volver a la perspectiva Java debes pulsar en la pestaña Java.

5. Escribe y ejecuta la clase MiPrimerGUI en el paquete sesion1. La perspectiva de Eclipse deberá tener el siguiente aspecto al terminar:



6. Ejecuta la clase de la misma forma que el programa HolaMundo. Puedes comprobar que no hay nada raro en tener dos clases *ejecutables* en el mismo proyecto y en el mismo paquete.

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

Ejercicio 8: El juego de los números

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). Hazlo todo en el paquete ya creado sesion1.

```

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 acertado?: " + p1IsRight);
                System.out.println("El jugador 2 ha acertado?: " + p2IsRight);
                System.out.println("El jugador 3 ha acertado?: " + p3IsRight);
                break;
            } else {
                System.out.println("Los jugadores lo intentan otra vez");
            }
        }
    }
}

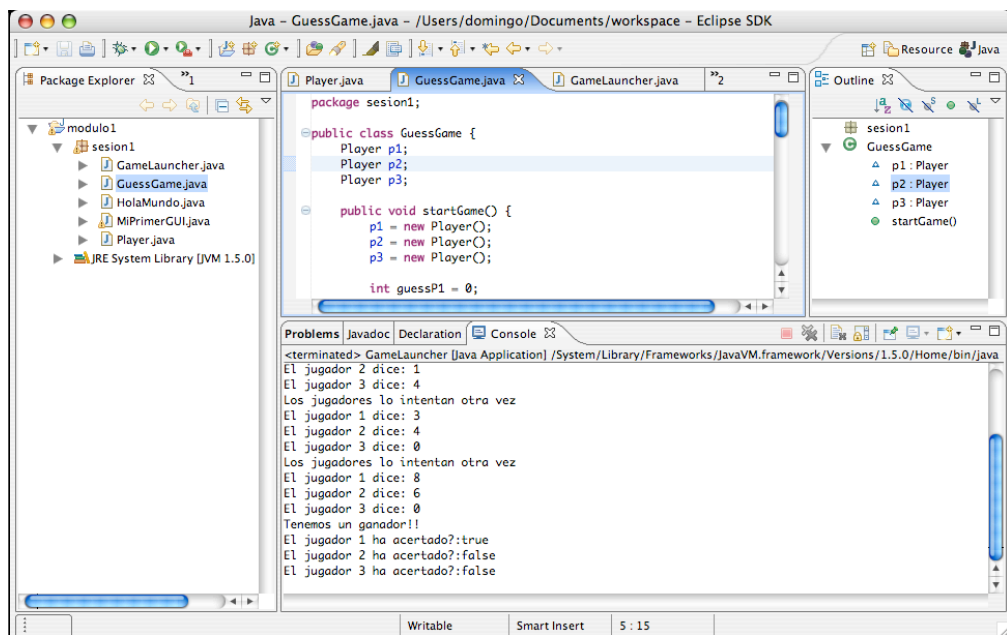
```

2. Escribe y compila la clase Player en el mismo paquete 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 que aparece en la salida estándar debe ser algo como:

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 acertado?: false
El jugador 2 ha acertado?: false
El jugador 3 ha acertado?: true
```

Y la apariencia de Eclipse después de crear todas las clases debe ser algo así:



El sistema CVS

Ya estamos a punto de terminar. El último ejercicio es una pequeña guía para publicar todo lo que has hecho durante esta sesión en el repositorio CVS. Pero antes de realizarlo, vamos a saber un poco más qué es eso de CVS.

CVS viene de las iniciales en inglés de *Sistema de Versiones Concurrente*. Se trata de una implementación de un sistema de control de versiones: mantiene todos los cambios de un conjunto de ficheros, como un proyecto de software y permite que varios desarrolladores colaboren en el mismo proyecto. CVS se ha hecho muy popular en el mundo de código abierto.

CVS utiliza una arquitectura cliente-servidor. Un servidor almacena el repositorio con las versiones actuales del proyecto y la historia de cambios. Los clientes se conectan para obtener (*check-out*) una copia completa del proyecto, trabajar en la copia y a continuación publicar (*check-in*) los cambios. Existen clientes de CVS en todos los sistemas operativos. Nosotros vamos a usar el cliente que lleva incorporado Eclipse. Es un cliente muy fácil de usar con una interfaz gráfica muy clara y está totalmente integrado en el entorno de programación.

Varios clientes pueden editar las copias del proyecto concurrentemente. Cuando se publican los cambios, el servidor intenta actualizar los ficheros del proyecto. Si se produce un fallo, por ejemplo debido a que dos clientes han modificado el mismo fichero, el servidor deniega la segunda operación de publicación e informa al cliente del conflicto. El usuario debe corregir el conflicto a mano y volver a publicar. Si la operación tiene éxito, los números de versión de todos los ficheros involucrados se incrementa automáticamente y el servidor CVS guarda una descripción del cambio proporcionada por el usuario, la fecha y el nombre del autor en los ficheros de log.

Los clientes pueden también comparar diferentes versiones de los ficheros, solicitar una historia completa de los cambios o también pueden obtener una instantánea histórica del proyecto tal y como estaba en una fecha dada o en un número de versión dado.

Los clientes pueden también usar el comando actualizar (*update*) para poner al día la copia local del proyecto y bajarse las últimas modificaciones publicadas por otros clientes.

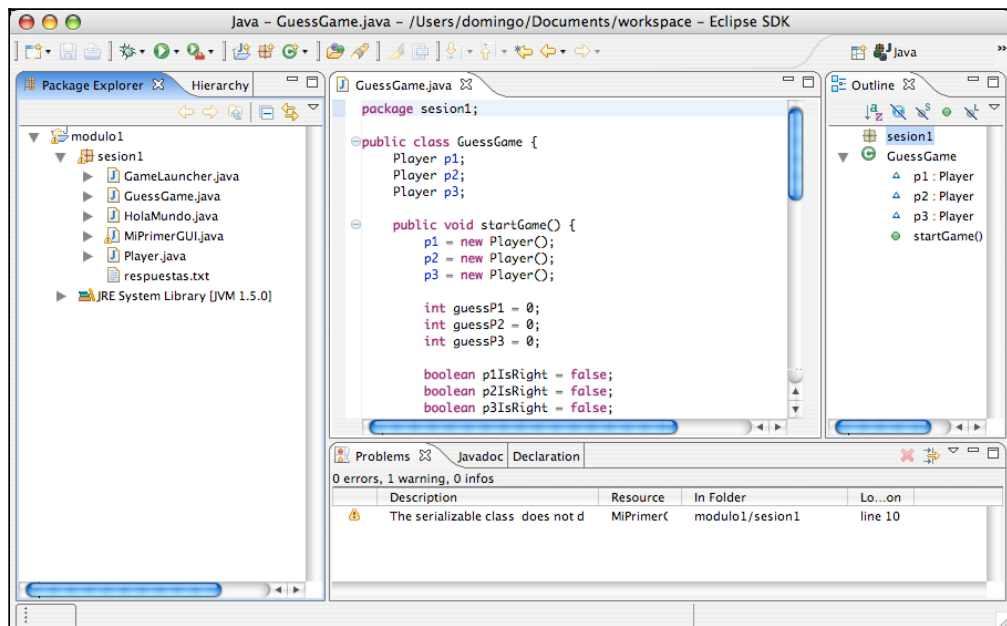
CVS puede también mantener distintas *ramas* de un proyecto. Por ejemplo, una versión lanzada de un proyecto software puede tener más de una rama y cada rama tener un control separado de cambios.

Nosotros vamos a usar sólo una mínima parte de las funcionalidades de CVS. Lo vamos a usar para publicar los ejercicios en un repositorio común. Cada alumno va a tener su directorio CVS independiente en el que publicar los ejercicios. Los profesores también tendremos acceso a esos directorios y podremos corregir fácilmente los ejercicios entregados. Lo vamos a ver paso a paso en el siguiente (último) ejercicio.

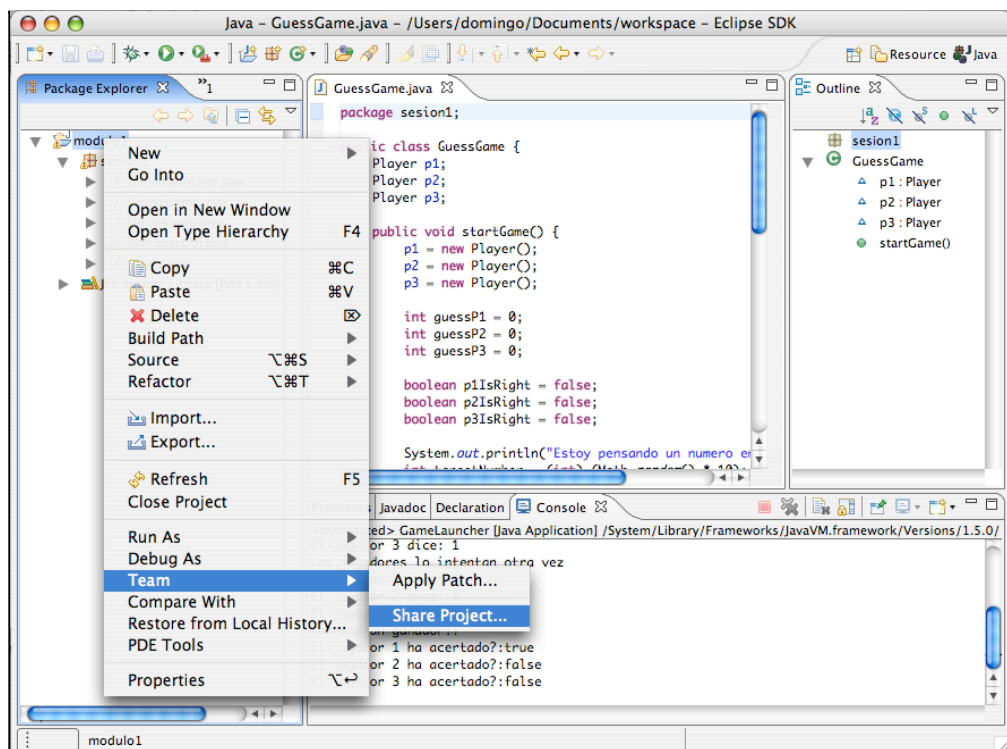
Ejercicio 9: Publicación en el repositorio CVS

Vamos a terminar la sesión de hoy publicando todos los ejercicios en el repositorio CVS.

1. Lo primero es incorporar el fichero respuestas.txt al proyecto de Eclipse. Abre un explorador del sistema de ficheros en el que puedas ver el fichero respuestas.txt. Arrástralo a la ventana de eclipse, sobre el explorador de paquetes y en el módulo sesion1. Debe quedar así:

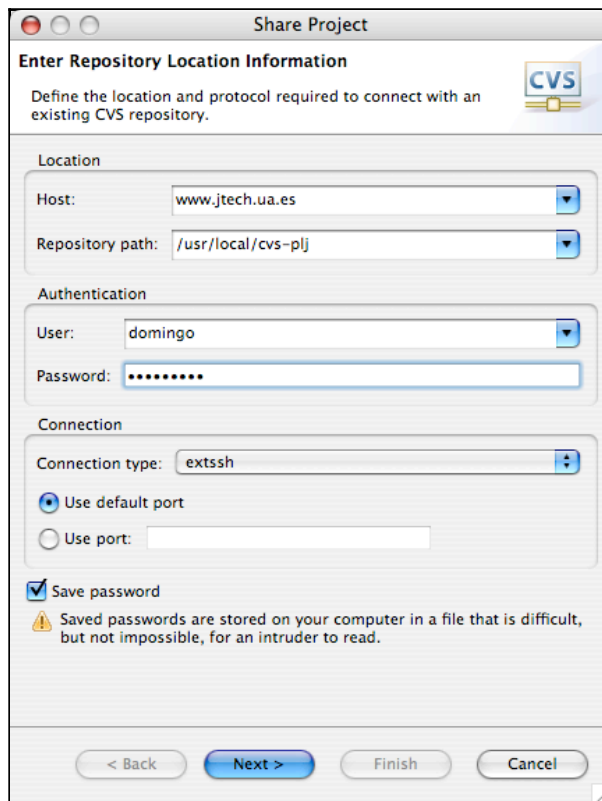


2. Pincha el nombre del proyecto y pulsa el botón derecho. Selecciona la opción Team > Share project...:

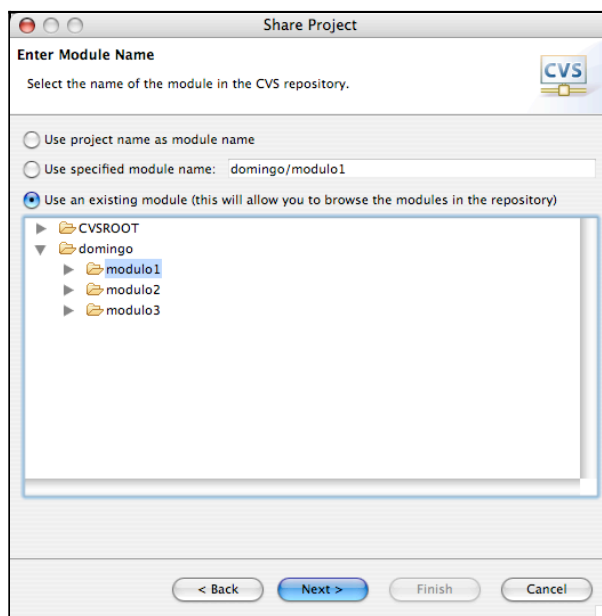


Introduce los datos del repositorio CVS:

- Servidor: www.jtech.ua.es
- Ruta del repositorio: /usr/local/cvs-plj
- Usuario: tu nombre de usuario en la web de PLJ
- Contraseña: tu contraseña de usuario en la web de PLJ
- Tipo de conexión: extssh

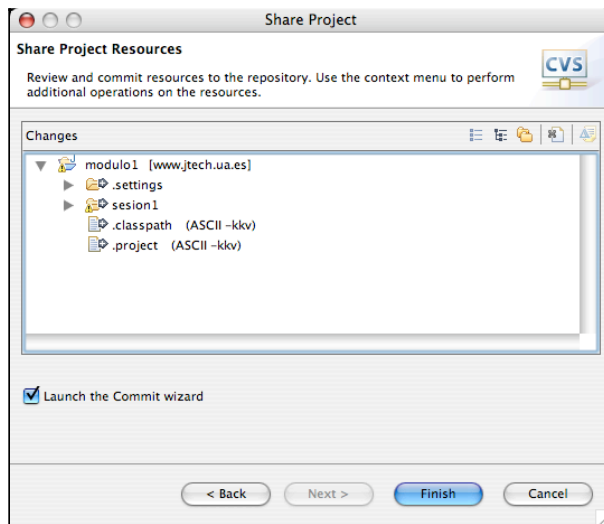


Ahora debes seleccionar el directorio del repositorio CVS en el que publicar los ficheros del proyecto. Debes pinchar la última opción (Use an existing module ...) para que puedas desplegar los distintos directorios del repositorio en la ventana del asistente. Habrá un directorio por alumno del curso. Abre la carpeta con tu nombre de usuario y verás que contiene a su vez otras tres carpetas. Selecciona la primera de ellas (modulo1).

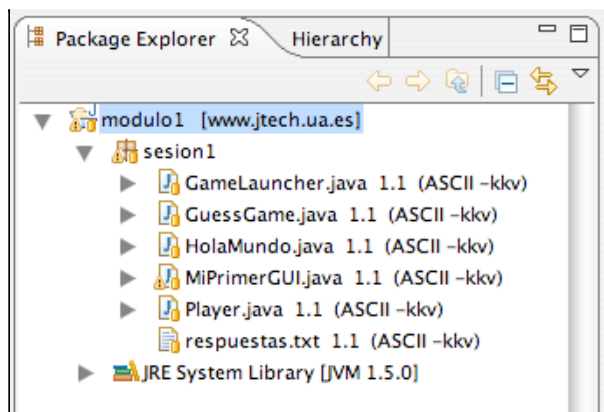


Acepta las ventanas que aparecen y llegarás a una ventana en la que Eclipse resume las acciones que van a realizarse en el repositorio CVS. En

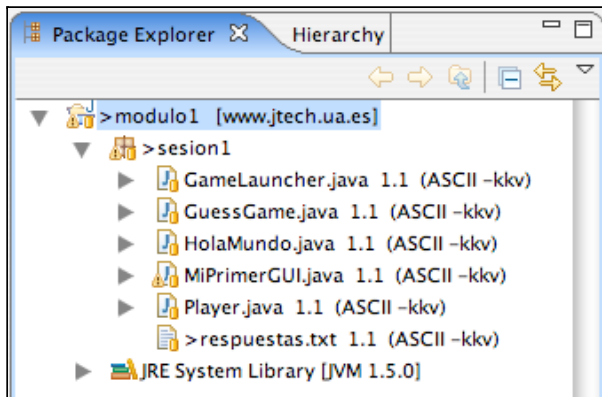
este caso se van a añadir (de ahí el signo "+" en la flecha) al repositorio todos los ficheros del proyecto.



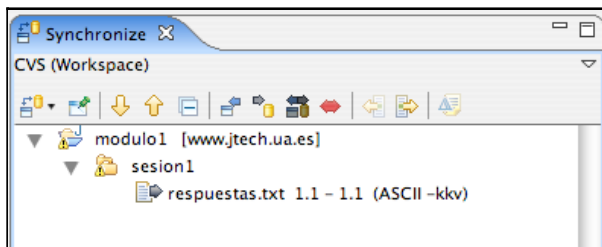
Por último, cuando pulses Finish Eclipse publicará los cambios en el repositorio. Te permite escribir un mensaje con la indicación de qué cambios has realizado. En este caso puedes escribir "versión inicial", o algo similar. Una vez publicados los ficheros, cambiarán los iconos que aparecen en el explorador de paquetes.



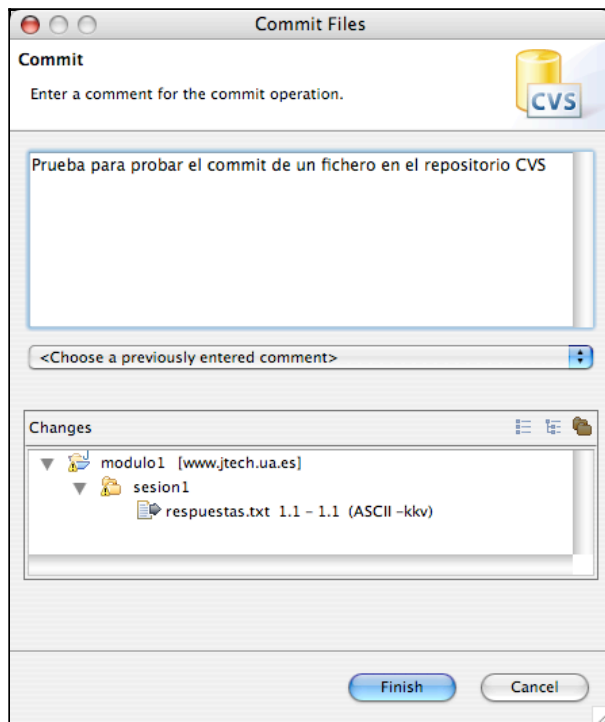
3. Por último, vamos a probar a realizar un cambio en algún fichero y a actualizar el repositorio con este cambio. Modifica uno de los ficheros, por ejemplo el fichero respuestas.txt. Verás que aparece un símbolo ">" junto a él, para indicar que está modificado con respecto a la versión del repositorio.



Pulsa el proyecto y selecciona con el botón derecho la opción Team > Synchronize with repository. Eclipse te consulta si quieres cambiar a la perspectiva Team Synchronizing. Dile que sí y aparecerá una vista como la siguiente.



En la ventana se muestran los cambios de la copia local respecto al repositorio. En este caso se indica que para realizar una sincronización hay que subir (flecha hacia la derecha) al repositorio los cambios del fichero respuestas.txt. Si hubiera algún cambio en el repositorio con respecto a la copia local también aparecería mostrado, en este caso con una flecha hacia la izquierda. Esto sucederá cuando el profesor corrija los ejercicios. En este caso, puedes publicar los cambios pulsando en el icono que aparece resaltado y que tiene la flecha amarilla hacia la derecha. Es la forma gráfica de hacer un *commit* en el repositorio CVS. Puedes escribir un comentario sobre los cambios que has realizado.



Pulsa Finish y se publicará el cambio. Termina cambiando a la perspectiva Java y cerrando Eclipse.

Sesión 2. Programación Orientada a Objetos

Introducción

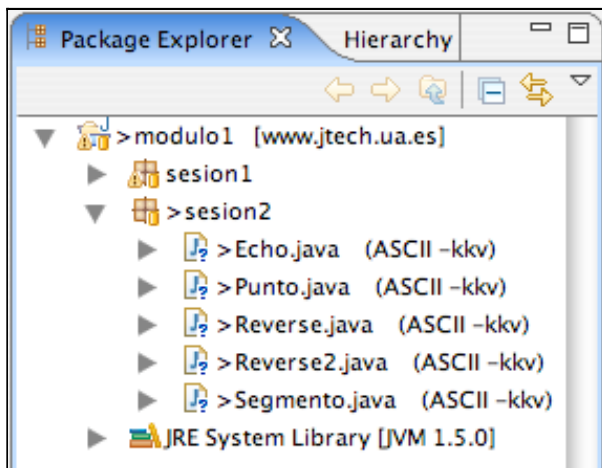
Antes de empezar a hablar de Programación Orientada a Objetos, vamos a continuar con algún ejercicio básico de Java. Nos va a servir para comprobar cómo usar en Java elementos muy útiles como los arrays o los parámetros de entrada de un programa. También vamos a aprovechar este primer ejercicio para incorporar al proyecto modulo1 algunas clases que hemos dejado en la web como material de la sesión.

Ejercicio 1: Algún ejercicio más de programación básica en Java

1. Lo primero es incorporar al proyecto actual las clases Java que te proporcionamos en esta sesión de ejercicios.

Primero descarga y descomprime el fichero sesion2.zip, que está en página de ejercicios de la web del curso. Verás que contiene una carpeta con el nombre sesion2 y, dentro de ella, algunas clases Java. Son las clases iniciales que vas a usar en esta sesión de ejercicios.

Vamos a arrastrar esas clases al proyecto actual de Eclipse. En el sistema operativo abre una ventana del explorador de ficheros y busca la carpeta sesion2 que acabas de descargar y descomprimir. Arrastra esa carpeta a la ventana de explorador de paquetes de Eclipse. Verás que Eclipse incorpora la carpeta en el proyecto, creando un nuevo paquete Java. La vista tendrá el siguiente aspecto (no te fijas demasiado en el nombre de las clases, puede que haya alguna más o menos de las que realmente tienes):



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

```
package sesion2;

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();
    }
}
```

Ejecuta este programa en Eclipse. Para lanzar la clase, deberás pasarle al intérprete los argumentos de entrada. Eso es muy sencillo cuando estás usando la línea de comandos:

```
>java Echo ho!a que ta!
ho!a:que:ta!
```

Pero, ¿cómo lanzar un programa con argumentos en la línea de comandos desde Eclipse? Para esto es necesario crear una *configuración de ejecución*, un elemento muy útil de Eclipse. Selecciona en el menú la opción Run > Run ... (o Run As > Run... si eres de los que te gusta ahorrar tiempo y usar el botón derecho). 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 Echo 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. 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 ":". Si lo ejecutáramos desde la línea de comando aparecería lo siguiente.

```
>java Reverse Ho!a que ta!
ta!:que:Ho!a
```

4. 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().

```
>java Reverse2 Ho!a que ta!
!at:euq:a!oH:
```

Programación Orientada a Objetos

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

Supongamos la siguiente clase Persona, definida en un paquete llamado base

```
package base;

public class Persona {
    String nombre;
    public int edad;

    public Persona() {
        nombre = "Pepe";
        edad = 18;
    }

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

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}
```

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

Supongamos ahora la siguiente clase TestPersona que contiene un método main. Supongamos que se encuentra definida en un paquete distinto del paquete base, por ejemplo el paquete test. En este caso habrá que importar la clase anterior (por estar en un paquete distinto):

```
package test;
import base.Persona;

public class TestPersona {
    public static void main(String[] args) {
        Persona p = new Persona();
        System.out.println("Nombre de persona: " + p.getNombre());
        p.setNombre("Maria");
        System.out.println("Nombre de persona: " + p.getNombre());
        System.out.println("Edad de persona: " + p.getEdad());
        p.edad = 9;
        System.out.println("Edad de persona: " + p.getEdad());
    }
}
```

El test debería dar los siguientes resultados.

```
Nombre de persona: Pepe
Nombre de persona: Maria
Edad de persona: 18
```

Edad de persona: 9

En POO debemos pensar que los objetos encapsulan (contienen) tanto los datos como los métodos que modifican y acceden a estos datos. Así, una instancia de la clase `Persona` contiene los datos nombre y edad y los métodos `getNombre`, `getEdad` y `setNombre`. Es posible acceder a un método o una variable de instancia de un objeto usando el operador `"."`. Por ejemplo:

```
p.setNombre("Maria");  
p.edad = 9;
```

Podemos usar *modificadores de acceso* para configurar los permisos de acceso a los elementos. En el ejemplo anterior todos los métodos y la variable de instancia `edad` tienen el modificador `public`. La variable de instancia `nombre` tiene el nivel de acceso por defecto. En este nivel, se puede acceder al elemento desde clases definidas en el mismo paquete, pero no en otros paquetes.

Lo normal es restringir el acceso a las variables de instancia de un objeto y no hacerlas públicas, sino que todos los accesos externos a los datos de un objeto sean a través de métodos. Es el caso de la variable `nombre` del ejemplo. La variable `edad` la hemos hecho pública para mostrar que también es posible. Hablaremos más sobre los selectores de acceso en la siguiente sesión.

Veamos ahora el siguiente código. Crea un objeto de la clase `Persona` y lo guarda en la variable `unaPersona`. Después se llama al método `setNombre` y se modifica la edad del objeto recién creado.

```
Persona unaPersona = new Persona();  
unaPersona.setNombre("Juan Pérez");  
unaPersona.edad = 12;
```

Después de ejecutar el código, el objeto `unaPersona` tendrá como nombre el *String* "Juan Pérez" (otro objeto, por cierto) y como edad el entero (*int*) 12. En Java existen datos primitivos que no son objetos, como por ejemplo *double*, *int*, *char*, etc. Se pueden reconocer porque el nombre del tipo no comienza por mayúscula.

Otra cosa a resaltar del ejemplo anterior. Fijémonos en la primera instrucción del ejemplo anterior:

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

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

```
Persona otraPersona = unaPersona;
```

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

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

Una última consideración. ¿Qué sucede cuando pasamos un objeto como parámetro de una función?. Supongamos que añadimos el siguiente método en la clase Persona:

```
public void copiarEn(Persona p) {  
    p.edad = this.edad;  
    p.nombre = this.nombre;  
}
```

Y supongamos que escribimos el siguiente método main:

```
public static void main(String[] args) {  
    Persona p1, p2;  
  
    p1 = new Persona();  
    p1.setNombre("Pepe");  
    p1.edad = 40;  
    p2 = new Persona();  
    p2.setNombre("Juan");  
    p2.edad = 10;  
    p1.copiarEn(p2);  
    System.out.println(p2.getNombre());  
    System.out.println(p2.edad);  
}
```

Cuando llamamos al método copiarEn del objeto p1, le pasamos como parámetro el objeto p2. Tenemos que entender que le estamos pasando al método una referencia (¡no una copia!) al objeto p2. Así, cuando se dentro del método copiarEn modificamos el objeto que se pasa como parámetro estamos modificando el objeto p2. Por esto la salida del ejemplo sería:

```
Pepe  
40
```

Repasamos a continuación de forma concisa algunos de estos conceptos fundamentales de POO, incluyendo ejemplos adicionales:

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

Supongamos el siguiente programa Java

```
import java.util.*;  
  
public class MiClase
```

```
{
    public int a;
    ArrayList v;

    public MiClase()
    {
        a = 0;
        v = new ArrayList();
    }

    public void imprimirA()
    {
        System.out.println (a);
    }

    public void insertar(String cadena)
    {
        v.add(cadena);
    }
}
```

La forma de especificar los elementos en el programa es la siguiente:

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

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

- **Clases:**

```
public class MiClase {
    ...
}
```

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

```
public int a;
ArrayList v;
```

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

```
public void imprimirA() {
    // implementación del método
    System.out.println(a);
}

public void insertar(String cadena) {
    // implementación del método
    v.add(cadena);
}
```

- **Constructores:** Un tipo de método que siempre tiene el mismo nombre que la clase. Se pueden definir uno o varios. No devuelve nada; se llama automáticamente cada vez que se crea un objeto de la clase, para inicializar sus variables de instancia.

```
public MiClase() {  
    // inicializo las variables de instancia del objeto MiClase  
    a = 0;  
    v = new ArrayList();  
}
```

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

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

No tenemos que preocuparnos de liberar la memoria del objeto al dejar de utilizarlo. Esto lo hace automáticamente el **garbage collector**.

Ejercicio 2: Clases, objetos y variables

1. Fíjate ahora en las clases Punto y Segmento que estaban entre los materiales de la sesión. Escribe (en el paquete sesion2) una clase llamada TestGeom que contenga un método main y que pruebe estas clases.
2. Añade en la clase Punto un metodo traslada(incX, incY) que añade los incrementos en x e y a las coordenadas correspondientes del punto. Prueba a crear un segmento entro dos puntos dados p1 y p2 y a trasladar alguno de ellos. ¿Qué pasa con el segmento? ¿Cambia sus coordenadas? ¿Cambia su longitud? (incluye la prueba en la clase TestGeom y contesta en un fichero respuestas.txt que debe estar en el paquete sesion2).
3. Añade las clases Circulo y Rectangulo. Defínelas e impleméntalas como te parezca más adecuado. La única restricción es que debes definir en la clase Rectangulo un método interseca(Rectangulo otroRect) y otro método rectInterseccion(Rectangulo otroRect). El primero devuelve un booleano y comprueba si intersectan el rectángulo que ejecuta el método con otro que se le pasa como parámetro. El segundo método devuelve el rectángulo de intersección entre los dos rectángulos y null si ambos no intersectan.

Modifica el programa TestGeom para comprobar estas operaciones.

4. Escribe un programa llamado MuchosRectangulos que haga lo siguiente:
 - Pide por la entrada estandar un número al usuario.
 - Genera ese número de rectángulos aleatorios, los guarda en un array y muestra sus coordenadas por la salida estándar. Las coordenadas de los puntos deben estar entre 0 y 800, por ejemplo.
 - Comprueba qué rectángulos intersectan y muestra por la salida estándar el número de rectángulos que han intersectados y toda la información sobre las intersecciones: coordenadas de los rectángulos que intersectan y coordenadas del rectángulo resultante de la intersección.

Como ejemplo puedes mirar el siguiente código, que pide un número al usuario, guarda en un array esa cantidad de números aleatorios y escribe por la salida estándar los números generados que son pares.

```
package sesion2;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Numeros {
    static int[] numeros;

    public static void main(String[] args) {
        int i, j;
        String line;

        // Leo el número escrito por el usuario
        do {
            System.out.print("Cuántos números?: ");
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
            try {
                line = in.readLine();
                i = Integer.parseInt(line);
            } catch (IOException e) {
                i = 0;
                System.out.println("IOException: " + e);
            }
        } while (i > 100);

        numeros = new int[i];

        for (j = 0; j < i; j++) {
            double d = Math.random() * 800;
            System.out.println(d);
            numeros[j] = (int) d;
        }
        int nPares = 0;
        for (j = 0; j < numeros.length; j++) {
            int n = numeros[j];
            if (n % 2 == 0) {
                System.out.println(n);
                nPares++;
            }
        }
        System.out.println("Total pares: " + nPares);
    }
}
```

Variables y métodos de clase: el modificador static

Los campos y los métodos de una clases pueden tener el modificador static, como ya has visto en todas las funciones main que hemos escrito hasta ahora. ¿Qué significa esa palabra?.

Sencillamente, es la forma en Java de indicar que un elemento (campo o método) corresponde a la clase y no a las instancias de esa clase. Se usa para definir variables y métodos de clase. El elemento declarado como static corresponde a la clase en la que se define y todas los objetos de esa clase lo comparten. Es más, no es necesario crear ningún objeto de la clase para acceder a ese elemento, sino que

se accede mediante el propio nombre de la clase.

Por ejemplo, si declaramos lo siguiente:

```
public class Trabajador {  
    public static double sueldoBase;  
}
```

estamos declarando una variable de clase de la clase Trabajador. Al ser public podemos acceder a ella desde cualquier clase y paquete. Para acceder a la variable habría que referenciarla a través del propio nombre de la clase:

```
...  
double sueldo = Trabajador.sueldoBase;  
...
```

Con los métodos de clase sucede igual. Por ejemplo, la clase Math contiene una gran cantidad de métodos estáticos. Todos son del estilo siguiente:

```
public class Math {  
    public static double cos(double arg0) {  
        ...  
    }  
    ...  
}
```

Para llamar a un método de clase hay que usar el nombre de la clase. No es necesario crear ningún objeto de esa clase.

```
...  
double coseno = Math.cos(alfa);  
...
```

Por último, si añadimos a una variable de clase el modificador final estamos indicando que esa variable no se va a poder modificar. Es la forma de declarar constantes en Java. En el siguiente ejemplo se está definiendo la constante Trabajador.sueldoBase:

```
public class Trabajador {  
    public final static double sueldoBase = 550.40;  
}
```

Como curiosidad, el orden en el que se declaran los modificadores es indiferente. Podríamos haber escrito:

```
public class Trabajador {  
    double static final public sueldoBase = 550.40;  
}
```

Vamos con un ejercicio en el que comprobar estos conceptos.

Ejercicio 3: Contadores

Todas las clases las vamos a definir en el paquete sesion2. Se encuentran en la plantilla de ejercicios de la sesión 2.

Supongamos las siguientes clases Contador y ContadorTest

```
package sesion2;
public class Contador {
    static int acumulador = 0;
    int valor;

    static public int acumulador() {
        return acumulador;
    }

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

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

    public int getValor(){
        return valor;
    }
}

package sesion2;
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);
    }
}
```

1. 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, sin que cambie el funcionamiento de la clase? ¿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 sesion2;

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.

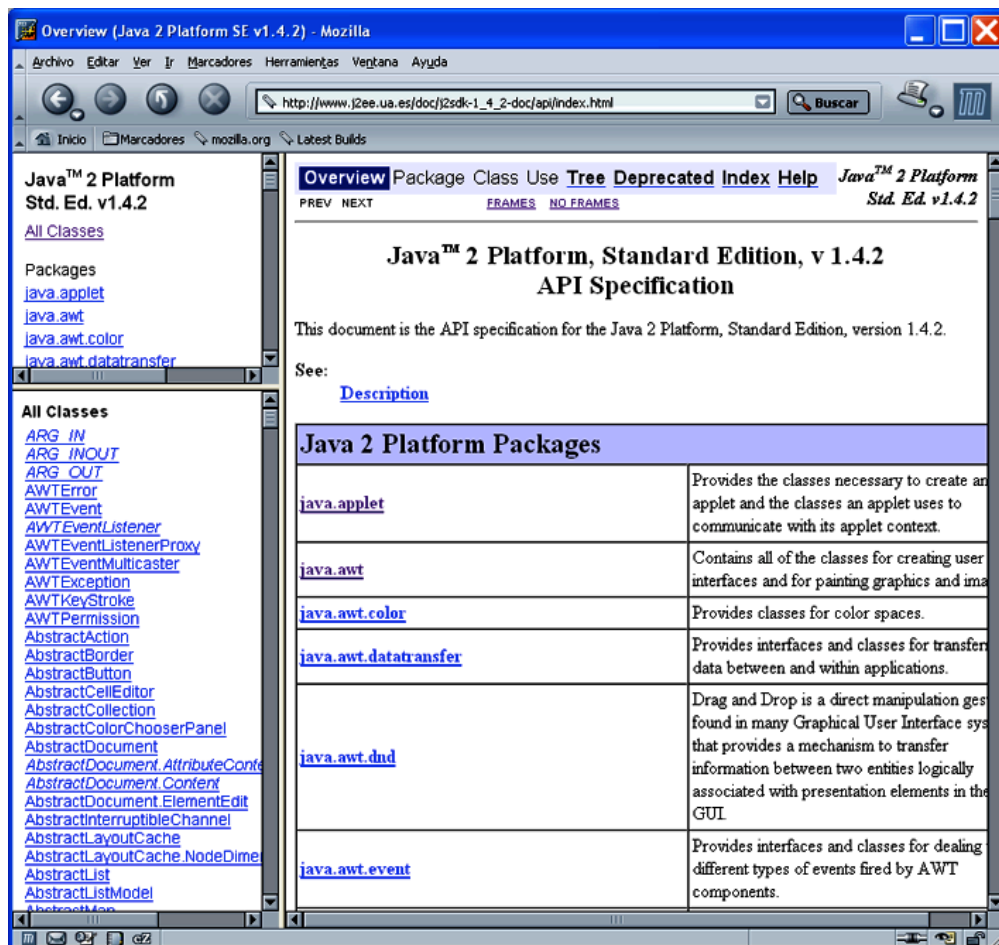
- Añade un metodo getValores que devuelva un array con los valores de los contadores creados.

La documentación del API de Java

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

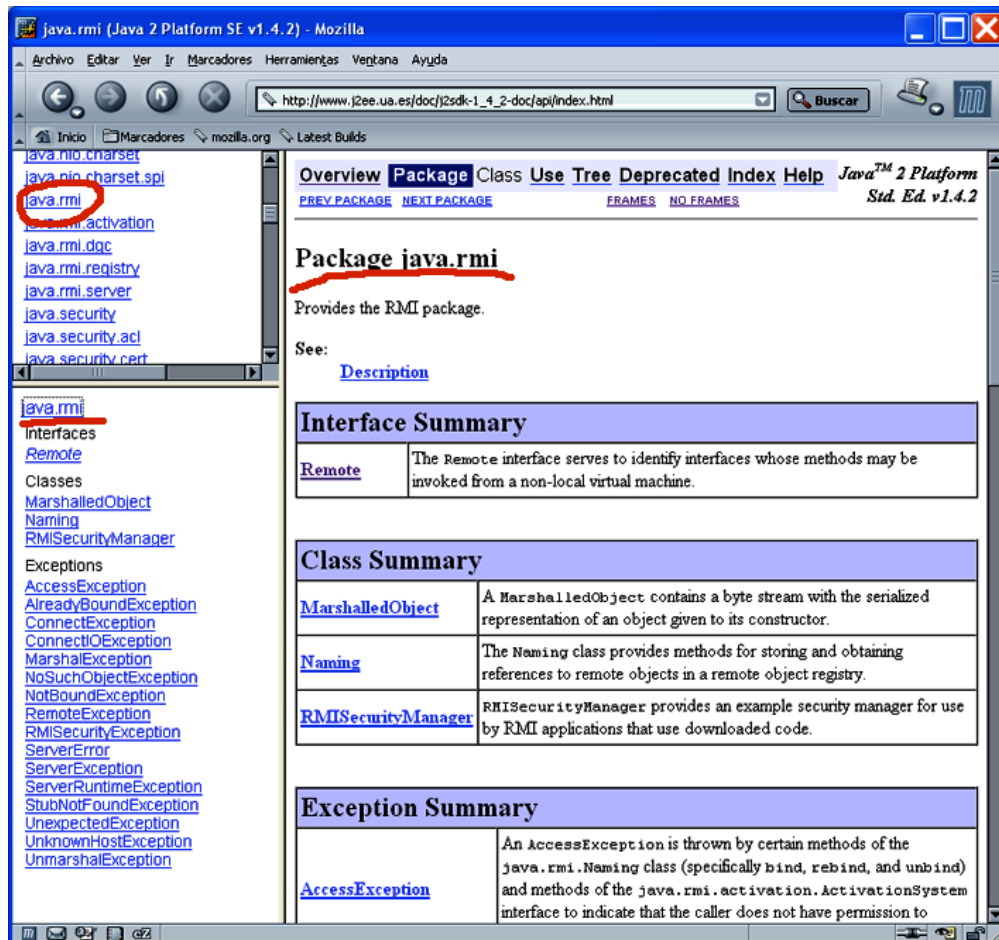
Una herramienta muy útil son las páginas HTML con la documentación del API de Java. Puedes encontrar estas páginas en los recursos del curso. Vamos a usar en concreto la versión 1.4.2 de Java.

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

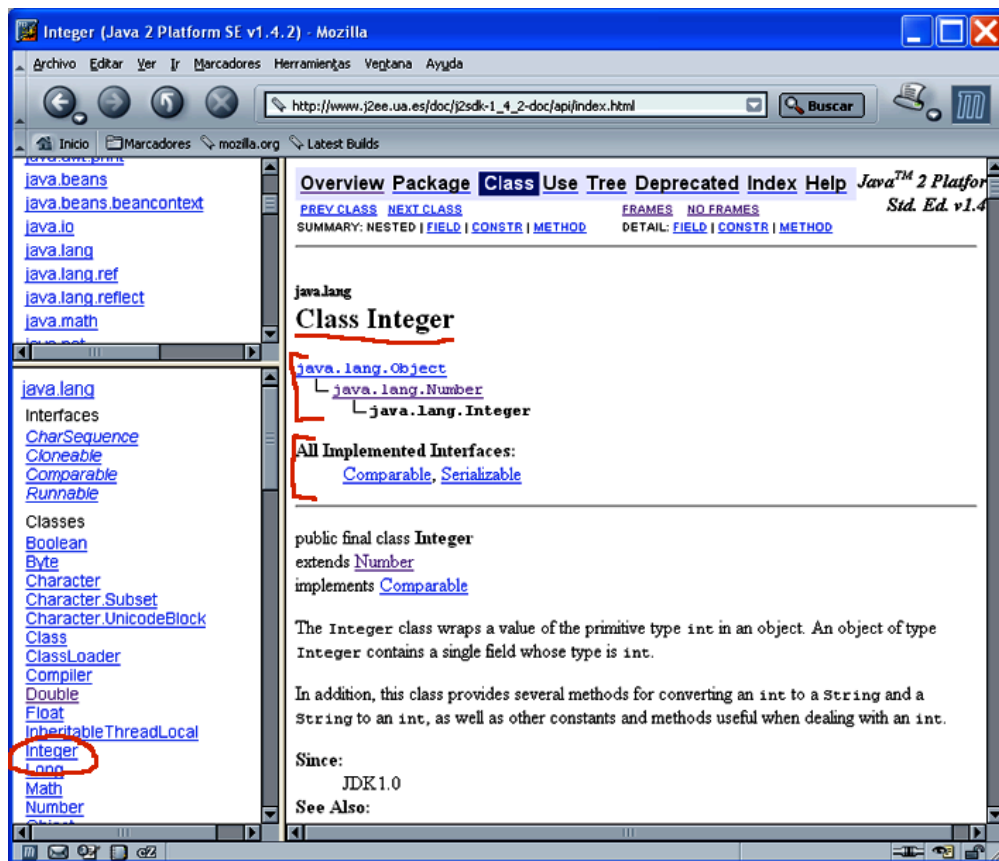


Si seleccionamos un paquete, por ejemplo java.rmi, aparece la siguiente página HTML. En el frame inferior izquierdo aparecen los elementos que constituyen el

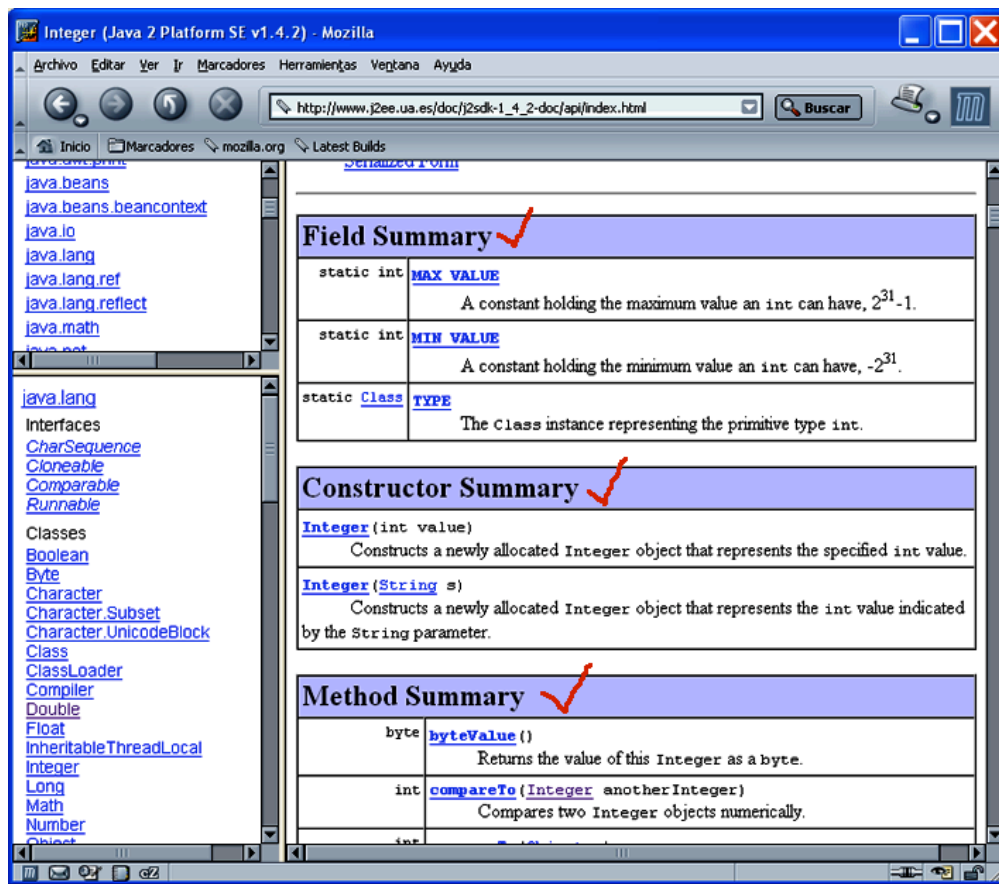
paquete: las clases, interfaces y excepciones definidas en el mismo. En el frame principal se describen con más detalle estos elementos. Todos los elementos están enlazados a la página en la que se detalla la clase, el interface o la excepción.



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



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



Ejercicio 4: El API de Java

- 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?
- 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 en Java 1.4.2, ¿por qué? (consulta el apartado 1.7.3 de los apuntes) ¿Cuál sería el código correcto?

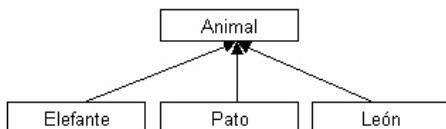
```
int i = 10;
Stack pila = new Stack();
pila.add(i);
```

- ¿Ese código sería correcto en Java 5.0?

Sesión 3. Programación Orientada a Objetos II

Herencia y polimorfismo

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



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

Las flechas hacia arriba indican una relación **ES-UN**:

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

```
class Pato extends Animal
```

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

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

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

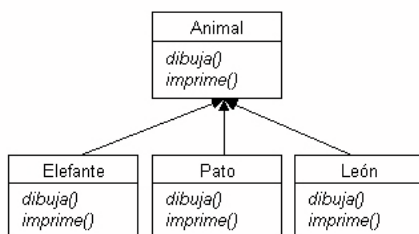
```
public class MiNuevaClase extends MiClase {
    public void Suma_a_i (int j) {
        i = i + (j / 2);
        super.Suma_a_i (j);
    }
}
```

```
}
}
```

Clases abstractas e interfaces

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

Usa una clase abstracta cuando quieras definir una **plantilla** 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.



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

- En una **clase**, al definir Animal tendríamos que implementar los métodos `dibujar()` e `imprimir()`. Las clases hijas no tendrían por qué implementar los métodos, a no ser que quieran adaptarlos a sus propias necesidades.
- En una **clase abstracta** podríamos implementar los métodos que nos interese, dejando sin implementar los demás (dejándolos como métodos abstractos). Dichos métodos tendrían que implementarse en las clases hijas.
- En un **interfaz** no podemos implementar ningún método en la clase padre, y cada clase hija tiene que hacer sus propias implementaciones de los métodos. Además, las clases hija podrían implementar otros interfaces.

La especificación en Java es como sigue.

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

```

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

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

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


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

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

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

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

Ejercicio 1. Un ejemplo de herencia

Considera la siguiente clase ObjetoGeometrico

```
package sesion3;

public class ObjetoGeometrico {
    double xMin, yMin;

    public ObjetoGeometrico(double xMin, double yMin){
        this.xMin = xMin;
        this.yMin = yMin;
    }

    public Punto puntoInicial() {
        return new Punto(xMin, yMin);
    }
}
```

Estamos definiendo un ObjetoGeometrico como algo que contiene una coordenada x (la coordenada x más pequeña del objeto geométrico) y una coordenada y (la coordenada y más pequeña del objeto geométrico). Esta es una característica común de todos los objetos geométricos.

La clase tiene el método puntoInicial() que devuelve un objeto Punto creado a partir de las coordenadas mínimas x e y del objeto geométrico.

Vamos a comenzar con el ejercicio.

1. Copia todas las clases geométricas (Punto, Segmento, Circulo y Rectangulo) del paquete sesion2 al sesion3. Crea en este paquete la clase ObjetoGeometrico anterior. Haz que todas las clases geométricas sean subclases de ella. Verás que aparecen errores en los constructores de las clases geométricas. Corrígelos.

Un truco de Eclipse muy útil: cuando Eclipse detecta un error aparece un indicador rojo en el editor a la izquierda de la línea donde se ha producido el error. Si posicionas el ratón sobre el indicador rojo aparece un mensaje indicando cuál es el error. Más aún: si en el indicador de error hay un pequeño icono con una bombilla es porque Eclipse cree que puede corregir el error. Haz un click (¡sólo uno!) en la bombilla y Eclipse te ofrecerá más de una opción para solucionar el error. Haz un doble click en la que creas más oportuno (normalmente la primera es la correcta) y *Eclipse corregirá el error*. Así de sencillo.

Copia la clase TestGeom del paquete sesion2 al sesion3. Añade en esta clase algunas sentencias para probar los cambios que acabas de hacer y responde las siguientes preguntas en el fichero respuestas.txt (nuevo fichero en el paquete correspondiente a la sesión actual), después de haber hecho las pruebas oportunas:

- ¿Se pueden crear objetos de la clase ObjetoGeometrico?
- ¿Un objeto de cualquiera de las subclases geométricas es también un objeto de la clase ObjetoGeometrico? ¿Cómo lo has comprobado?

2. Añade a la clase ObjetoGeometrico el siguiente método abstracto

```
public abstract double area();
```

Responde en respuestas.txt:

- ¿Qué cambio adicional hay que hacer para que la clase ObjetoGeometrico no tenga errores?
- ¿Se pueden ahora crear objetos de esa clase?
- ¿Qué error aparece en las subclases de ObjetoGeometrico?

Modifica todas las subclases de ObjetoGeometrico para corregir el error.

3. Añade a la clase ObjetoGeometrico el método abstracto abstract Rectangulo limites(); que devuelve el rectángulo que limita (de forma estricta) al objeto.

Prueba el nuevo método con alguna prueba en la clase TestGeom.

4. Añade e implementa en la clase ObjetoGeometrico el método siguiente

```
public boolean posibleInterseccion(ObjetoGeometrico objGeom2) {  
    // añadir aquí la implementación  
}
```

Este método debe devolver true cuando intersectan los rectángulos límites de los objetos geométricos y false en otro caso.

Prueba el método en la clase de pruebas.

5. Por último, escribe en la clase TestGeom un método que haga la siguiente prueba: crear algunos objetos geométricos, guardarlos todos en un array (el mismo array para todos) y recorrer el array imprimiendo por la salida estándar el tipo de objeto geométrico y su área:

```
El objeto 1 es un círculo de área 3.453245
El objeto 2 es un segmento de área 0.0
El objeto 3 es un punto de área 0.0
...
```

Haz que el método principal de TestGeom termine llamando a esta prueba.

Ejercicio 2. Crea una jerarquía de clases

1. Define e implementa un ejemplo de jerarquía de clases. Define una clase ejecutable Test que realice unas pruebas para comprobar que el ejemplo funciona correctamente.

Ejercicio 3. Un ejemplo de interfaces

En este ejercicio vamos a continuar con el ejemplo de las figuras geométricas, definiendo las interfaces Dibujable y Medible.

1. Vamos a empezar por crear una interfaz en Eclipse. Pincha en el paquete sesion3 y escoge la opción New > Interface. Escribe Dibujable como nombre de la interfaz.

Escribe el siguiente código:

```
package sesion3;

public interface Dibujable {
    public void draw();
}
```

Fíjate que una interfaz define un conjunto de métodos pero no proporciona la implementación. La implementación debe estar en la clase que implementa la interfaz. De esta forma, cualquier objeto de una clase que implementa la interfaz Dibujable va a poder responder al método draw().

Ahora modifica las clases Segmento, Círculo y Rectángulo para que implementen la interfaz Dibujable. Puedes poner como implementación de los métodos draw() que se escriba un mensaje en la salida estándar.

Añade a la clase TestGeom el método testInterfazDraw() en el que se pruebe la interfaz. Llama a ese método en el último paso de la ejecución de TestGeom.

2. Vamos a complicar un poco el ejemplo. Supongamos la interfaz Medible definida de la siguiente forma:

```
package sesion3;

public interface Medible {
```

```
static final double A_CENTIMETROS = 1.0;
static final double A_PUNTOS = 2.0;
static final double A_PULGADAS = 3.0;

public double tamaño();
}
```

Uno de los objetivos de este ejemplo es comprobar que es posible definir constantes en las interfaces. En el caso anterior, estamos definiendo tres constantes que representan factores de conversión para convertir las unidades en las que están definidas las figuras geométricas (las que definen sus coordenadas x e y y su área) en distintas unidades métricas.

Crea la interfaz Medible en el paquete actual. Y ahora declara que la clase ObjetoGeometrico implementa la interfaz.

¿En qué clases aparecen errores? ¿Por qué piensas que aparecen errores en esas clases? (contesta en el ya famoso fichero respuestas.txt).

Y ahora, para terminar, un reto: ¿cómo puedes arreglar *todos* los errores modificando *una única clase*?. Implementa la función tamaño() de forma que se devuelva el área de la figura pasada a centímetros. Comprueba que todo funciona bien haciendo un test en el fichero TestGeom y llamándolo desde el método principal.

Por último, explica en el fichero respuestas.txt cómo piensas que está funcionando lo que acabas de implementar (explícalo como si se lo contaras a alguien que no sabe nada de interfaces ni de subclases).

Modificadores de acceso

Un elemento (método, variable de clase o variable de instancia) de una clase tiene asociado unas condiciones de acceso según el modificador de acceso que definamos en el mismo. El modificador de acceso define desde qué clases se va a poder acceder al elemento. Así, por ejemplo, un método con modificador de acceso public permite que desde *cualquier* otra clase se realice una llamada al mismo.

En Java existen cuatro posibles niveles de acceso: private, *vacío* (cuando 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 las subclases. **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.
- **vacío**: es el nivel de acceso que tiene un elemento si no declaramos nada. **Se puede acceder al elemento desde cualquier clase del mismo paquete.** Si, por ejemplo, el campo A de la clase Clase1 que está en el paquete sesion2 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.

Vamos a con un pequeño ejercicio para comprobar los modificadores de acceso de Java

Ejercicio 4: Modificadores de acceso

1. Supongamos la siguiente clase en el paquete sesion3

```
package sesion3;
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 sesion3;
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 sesion3;
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 pruebaAcceso (créalo antes), modificando la instrucción package y añadiendo el import de la clase sesion3.Acceso:

```
package pruebaAcceso;  
import sesion3.Acceso;
```

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

Publica todo el proyecto en el repositorio CVS. Enhorabuena, ya has terminado el primer módulo del curso.

Sesión 4. Excepciones

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

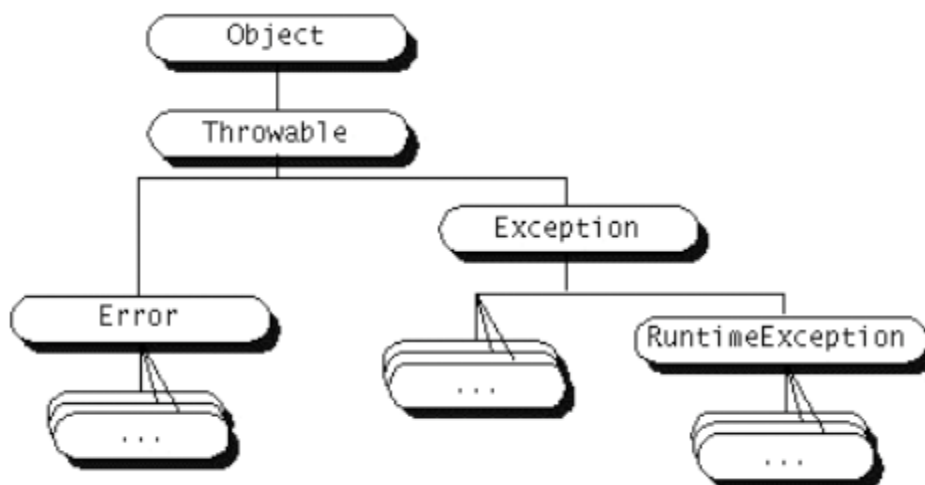
Tipos de excepciones

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

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

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

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



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

Captura de excepciones

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

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

Podemos poner un bloque *try* sin *catch* pero con *finally*, ya que cuando la excepción no es de tipo *checked* no es obligatorio capturarla en *catch*, pero puede haber cosas que queramos hacer tanto si se produce como si no. También podemos poner un bloque *try* con *catch* y sin *finally*, cuando queremos capturar la excepción, y no hay ningún código que necesitemos que se ejecute inexorablemente.

Para el bloque *catch* además deberemos especificar el tipo o grupo de excepciones que tratamos en dicho bloque, pudiendo incluir varios bloques *catch*, cada uno de ellos para un tipo/grupo de excepciones distinto.

La sintaxis genérica de *try - catch - finally* será la siguiente (luego podemos omitir las partes que no necesitemos):

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



```

    } finally {
        // Código de finalización
    }

```

Si como tipo de excepción especificamos un grupo de excepciones este bloque se encargará de la captura de todos los subtipos de excepciones de este grupo. Por lo tanto, si especificamos **Exception** capturaremos cualquier excepción de tipo *Exception* y sus subtipos, es decir, todas las excepciones existentes. Y si ponemos **Throwable** capturaremos cualquier tipo de error o excepción, ya que es el elemento padre de todos los errores y excepciones.

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

```

String getMessage()
void printStackTrace()

```

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

Unos ejemplos de uso:

```

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

try
{
    ... // Aquí va un código que puede lanzar una excepción que no es obligatorio capturar
} finally {
    ... // Aquí pondremos lo que queramos que se haga salte o no salte la excepción
}

```

Ejercicio 1: Captura de excepciones

Echa un vistazo a la clase *sesion04.Ej1* 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.

Compila y prueba el funcionamiento de la clase, ejecutando una división sencilla, como por ejemplo 20 / 4, de la siguiente forma:

```
java sesion04.Ej1 20 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.

- Primero comprobaremos que al programa se le han pasado 2 parámetros. Para ello, captura una excepción de tipo **ArrayIndexOutOfBoundsException**, al acceder al parámetro *args*:

```
public static void main(String[] args)
{
    try
    {
        param1 = args[0];
        param2 = args[1];
    } catch (ArrayIndexOutOfBoundsException e) {
        ...
    }
    ...
}
```

En el bloque *catch* pon el mensaje de error que consideres oportuno (por ejemplo, "Faltan parámetros"), y sal del programa (con *System.exit(-1)*). Prueba después la excepción capturada, ejecutando el programa con algo como:

```
java sesion04.Ej1 20
```

Debería capturar la excepción y mostrar por pantalla el mensaje "*Faltan parametros*" (o el que hayas elegido).

- Ahora vamos a comprobar que los dos parámetros son numéricos. Para esto captura 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(...)*):

Pon el mensaje de error que quieras en el bloque *catch* (por ejemplo, "Formato incorrecto del parámetro"), y prueba después la excepción capturada, ejecutando el programa con algo como:

```
java sesion04.Ej1 20 hola
```

Debería capturar la excepción y mostrar por pantalla el mensaje "*Formato incorrecto del parametro*", o el que hayas elegido.

- 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: " + divide(dividendo, divisor));
}
```

```
}
```

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.

Lanzamiento de excepciones

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

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

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

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

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

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

- Juntando estos dos pasos:

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

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

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

```

    ...
}

...

public void otroMetodo()
{
    try
    {
        leeFich();
    } catch (ParseException e) {
        System.out.println ("Se ha producido un error al leer el fichero");
        System.out.println ("El mensaje es: " + e.getMessage());
    }
}

```

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

Ejercicio 2: Lanzamiento de excepciones

Sigamos con la clase *Ej1.java* vista antes. 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*.

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 ("Divisor incorrecto");

    ...
}

```

Compila y ejecuta la clase. Prueba con algo como:

```
java sesion04.Ej1 20 0
```

¿Qué mensaje aparece? ¿Qué significa?

Captura la excepción en el *main* y muestra la traza del error con *printStackTrace*. Observa que capturar la excepción y hacer *printStackTrace* produce casi el mismo resultado que no capturarla. Esta técnica se utiliza en la fase de depurado para corregir errores del programa. Una vez depurado, ya se muestran mensajes de error más "amigables".

Creación de nuevas excepciones

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

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

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

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

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

...

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

Ejercicio 3: Creación de excepciones

Vamos a terminar con la clase *Ej1.java* que estamos completando. 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**, en el paquete **sesion04**, como la siguiente:

```
public class NumeroNaturalException extends Exception
{
    public NumeroNaturalException(String mensaje)
    {
```

```

        super(mensaje);
    }
}

```

¿Para qué sirve la llamada a *super* en el constructor en este caso?

- Ahora añadiremos el lanzamiento de esta excepción en el método *divide*, comprobando antes de dividir que tanto dividendo como divisor son positivos:

```

public static int divide(int dividendo, int divisor)
{
    ...
    if (dividendo < 0 || divisor < 0)
        throw new NumeroNaturalException("La division no es natural")
    ...
}

```

Prueba a compilar la clase. ¿Qué error te da? ¿A qué puede deberse?

- 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 int divide(int dividendo, int divisor)
    throws 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. Por eso antes cuando utilizamos *IllegalArgumentException* no hemos tenido que añadirla, porque pertenece a *RuntimeException*.

Prueba a compilar la clase. ¿Qué error te da ahora? ¿Por qué?

- Para terminar de corregir la clase, captura la excepción que se lanza en el método *divide*, cuando utilizamos dicho método en el *main*.
- ¿Para qué sirve la llamada a "getMessage()" (qué texto obtenemos con esa llamada)?
- Crea una nueva excepción **DivideParException** en el paquete **sesion04**, 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.
- Compila y ejecuta el programa de las siguientes formas, observando que da la salida adecuada:

```

java sesion04.Ej1 20 4           // 5
java sesion04.Ej1 20           // "Faltan parametros"
java sesion04.Ej1 20 ho!a       // "Formato incorrecto del parametro"
java sesion04.Ej1 20 0         // Excepción de tipo IllegalArgumentException

```

```
java sesion04.Ej1 20 -1    // "Error: La division no es natural"  
java sesion04.Ej1 5 2     // "Error: La división entre 2 no es exacta"
```

PARA ENTREGAR

Guarda en la carpeta **modulo2** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**Ej1**, **NumeroNaturalException** y **DivideParException**), dentro del paquete **sesion04**, cada uno con las modificaciones que se han ido solicitando.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.

Sesión 5. Hilos

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

Creación de hilos

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

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

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

Para crear nuestro hilo mediante herencia haremos lo siguiente:

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

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

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

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

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

```
public class EjemploHilo implements Runnable
{
```

```

        public void run() {
            // Código del hilo
        }
    }

```

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

```

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

```

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

Estado y propiedades de los hilos

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

```

Thread t = new Thread(this);

```

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

```

t.start();

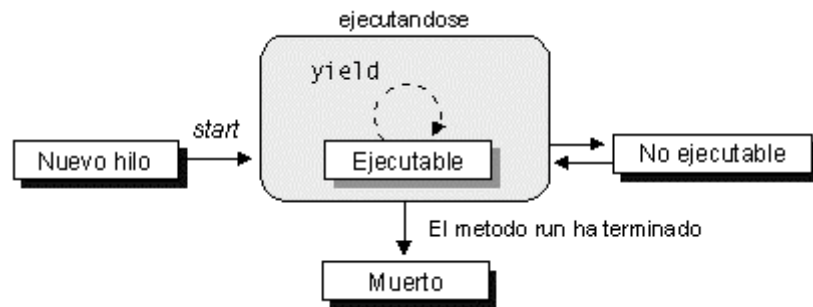
```

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

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

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

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



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

Prioridades de los hilos

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

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

Hilo actual

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

```

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

La llamada a *currentThread* dentro del constructor de la clase nos devolverá el hilo que corresponde con el programa principal (puesto que no hemos creado ningún otro hilo, y si lo creáramos, no ejecutaría nada que no estuviese dentro de un método *run*).

Sin embargo, en este otro caso:

```

public class EjemploHilo implements Runnable
{
    public EjemploHilo()
    {
    }
}
    
```

```

        Thread t1 = new Thread(this);
        Thread t2 = new Thread(this);
        t1.start();
        t2.start();
    }

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

```

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

Dormir hilos

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

Ejercicio 1: Hilos y prioridades

Implementaremos ahora un ejercicio para practicar diferentes aspectos sobre hilos y multiprogramación. Echa un vistazo a la clase *sesion05.Ej2* 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, que se encarga de liberar memoria no usada. 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.

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

¿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). ¿Qué pasaría si no estuviese el bucle *do...while* en el constructor del programa principal?

- Añade y lanza dos hilos más de tipo *MiHilo* en la clase *Ej2*. Para ello puedes copiar y pegar el código que crea y lanza el primer hilo, para hacer otros dos nuevo. Modifica también el bucle **do...while** para que muestre cuánto vale el contador de cada hilo cada vez.

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?

- Vamos a modificar la prioridad de los hilos, para hacer que terminen en orden inverso al que se lanzan. Para ello utiliza el método *setPriority* y da al hilo 3 la máxima prioridad (MAX_PRIORITY), al hilo 2 una prioridad normal (NORM_PRIORITY), y al hilo 1 una prioridad mínima (MIN_PRIORITY). Esto deberás hacerlo después de crear cada uno, justo antes de llamar a su método *start*.

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

- 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?
- Crea una clase **Ej2b** similar a la anterior en el paquete **sesion05**, pero en la que no sea el programa principal quien active los 3 hilos, sino que cada uno de los 3, al 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.

Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por ejemplo para acceder a una misma variable, o a un mismo fichero, por lo que es necesario un mecanismo de sincronización entre ellos.

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

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

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

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

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

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

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

Para desbloquear a los hilos que haya bloqueados se utilizará ***notifyAll()***, o bien ***notify()*** para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor. Notar que con *notifyAll* se despertarán todos, y "lucharán" por ver quién entra primero en el cerrojo. Notar también que no existen métodos para despertar un hilo concreto, sino que se despertarán todos (*notifyAll*), o bien uno al azar (*notify*).

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método ***join()*** de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado. Por ejemplo, si queremos esperar a que el hilo *t* termine para seguir con nuestro código, haremos algo como:

```
t.join();
```

Ejercicio 2: Sincronización de hilos

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.

Echa un vistazo a la clase *sesion05.Ej3* 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*.

A continuación sincronizaremos productores y consumidores para que funcionen adecuadamente.

- Compila y prueba varias veces (3 o 4) el funcionamiento de la clase. ¿Funciona bien el programa? ¿Qué anomalía(s) detectas?
- 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.

- Añade el código necesario en los métodos *produce* y *consume* para sincronizar el acceso a ellos. El comportamiento debería ser el siguiente:
 - 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 a que se consuman */

    ... // Comprueba aquí si hay datos disponibles, y si los hay llama

    /* Producir */
    this.valor = valor;

    /* Ya hay datos disponibles */
    /* Notificar de la llegada de datos a consumidores a la espera */
    ... // Haz aquí las modificaciones necesarias para indicar que ya l
    // y para notificar a los demás hilos que lo necesiten de que ya p
}
```

¿Qué papel puede tener el método *wait* en este código?

¿Para qué se utiliza el flag *disponible*?

¿Qué efecto puede tener la llamada a *notifyAll*? ¿Qué pasaría si no se llamase a este método?

- 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 esperar a que se produzcan */

    ... // Comprueba aquí si hay datos disponibles, y si no llama al m

    /* Ya no hay datos disponibles */
    /* Notificar de que el recipiente esta libre a productores en espe

    ... // Haz aquí las modificaciones necesarias para indicar que ya l
    // y para notificar a los demás hilos que lo necesiten de que ya p

    return valor;
}
```

- Compilad y ejecutad el programa. ¿Qué excepción o error da? ¿A qué puede deberse?

- Observad que 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. Añade la marca *synchronized* a los métodos *produce* y *consume* para solucionarlo.
- 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)

Grupos de hilos

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

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

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

PARA ENTREGAR

Guarda en la carpeta **modulo2** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**Ej2** y **Ej3**), dentro del paquete **sesion05**, cada uno con las modificaciones que se han ido solicitando.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.

Sesión 6. Utilidades

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

Wrappers

Un *wrapper* es un objeto que se utiliza para envolver a otro. De esta forma, podemos "encapsular" un objeto dentro de otro, y acceder al objeto encapsulado a través del objeto que lo encapsula. Veremos durante el curso que en Java se utilizan Wrappers en diferentes ámbitos.

Wrappers de tipos básicos

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

Si queremos agrupar diferentes elementos de estos tipos en un solo objeto, la única forma aparente de hacerlo es crear un array de ese tipo (por ejemplo, un array de *int[]*). Pero tenemos el inconveniente de que los arrays deben tener un tamaño estático. ¿Qué pasa si queremos tener una lista o colección de estos elementos, que vaya variando su tamaño? Cuando trabajamos con colecciones o listas de datos los elementos que contienen éstas son siempre objetos, por lo que en un principio no podríamos insertar elementos de estos tipos básicos. Para hacer esto posible tenemos una serie de objetos que se encargarán de envolver a estos tipos básicos, permitiéndonos tratarlos como objetos y por lo tanto insertarlos como elementos de listas. Estos objetos son los llamados wrappers, y las clases en las que se definen tienen nombre similares al del tipo básico que encapsulan, con la diferencia de que comienzan con mayúscula: **Boolean**, **Integer**, **Long**, **Float**, **Double**, **Byte**, **Short**, **Character**.

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

Ejemplos:

```
String unNumero = "123";
String otroNumero = "145.5";
int elNumero = Integer.parseInt(unNumero);
double elOtroNumero = Double.parseDouble(otroNumero);
```

esto sería útil por ejemplo para tomar valores numéricos de parámetros del programa, y pasarlos a un valor numérico verdadero para poder operar con él.

También podemos encapsular un valor simple dentro de su wrapper, y luego añadirlo a una lista (veremos las listas y colecciones más adelante):

```
int miNumero = 20;
Vector v = new Vector();

v.add(miNumero); // No está permitido!!
Integer intobj = new Integer(miNumero);
v.add(intobj); // Esto sí está permitido

// ¿Cómo recuperamos luego el entero?
Integer elemento = (Integer)(v.elementAt(0)); // Sacamos el wrapper del vector
int miValor = elemento.intValue(); // Y luego sacamos el valor
```

esto lo utilizaremos cuando veamos las colecciones, para añadir a ellas tipos simples y luego recuperarlos.

Ejercicio 1: Trabajando con wrappers

Utiliza la clase *sesion06.Ej4* que se proporciona en la plantilla para construir un programa que tome como parámetro (en el *main*) dos valores reales (*doubles*), y Muestre por pantalla el producto del primero y el segundo.

Captura la(s) excepcion(es) que consideres adecuadas para asegurarte de que el dato que se pasa como parámetro es numérico.

AYUDAS: utiliza la clase **Double**, y mira entre sus métodos cuáles pueden servirte para convertir los parámetros a números reales. Echa un vistazo a la clase *Ej1* que hiciste en la sesión de excepciones. Puede que te dé alguna pista de cómo hacer este otro ejercicio.

La clase Object

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

En Java es importante distinguir claramente entre lo que es una variable, y lo que es un objeto. Las variables simplemente son referencias a objetos, mientras que los objetos son las entidades instanciadas en memoria que podrán ser manipulados mediante las referencias que tenemos a ellos (mediante variable que apunten a ellos) dentro de nuestro programa. Cuando hacemos lo siguiente:

```
new MiClase()
```

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

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

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

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

Podremos hacer referencia a ella de diferentes formas:

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

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

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

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

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

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

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

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

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

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

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

hemos visto en el ejemplo.

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

```
mc1 == mc2
```

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

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

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

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

```
public class Punto2D {  
    public int x, y;  
  
    ...  
  
    public boolean equals(Object o) {  
        Punto2D p = (Punto2D)o;  
        // Compara objeto this con objeto p  
        return (x == p.x && y == p.y);  
    }  
  
    public Object clone() {  
        Punto2D p = new Punto2D();  
        // Construye nuevo objeto p  
        // copiando los atributos de this  
        p.x = x;  
        p.y = y;  
        return p;  
    }  
}
```

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

```
public class Punto2D {  
    public int x,y;
```

```

...

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

```

La cadena que devuelve este método puede utilizarse después en la salida que queramos: sacarla por pantalla (con `System.out.println`), volcarla a un cuadro de texto... etc.

Ejercicio 2: Aprovechando los elementos de Object

Echa un vistazo a la clase `sesion06.Ej5` de la plantilla. Verás que hay una clase interna llamada **MiObject** que contiene un valor entero y una cadena. Tiene un constructor para poder asignar valor a cada campo. Después, desde la clase principal `Ej5` se crean varios objetos de este tipo, cada uno con un valor entero y una cadena. Después se utiliza el método `equals` para compararlos entre sí, e indicar si son iguales o no. Finalmente, se imprime por pantalla cada uno de los objetos creados.

Asumimos que dos objetos de tipo `MiObject` son iguales si sus campos enteros son iguales, y si sus cadenas también son iguales. Teniendo en cuenta todo esto:

- Ejecuta el programa. ¿Funcionan bien las comparaciones? ¿Qué resultados obtienes con ellas?
- ¿Qué texto aparece cuando imprimes los objetos `m1`, `m2` y `m3`? ¿Te resultan entendibles?

Probablemente habrás descubierto que las comparaciones no funcionan del todo bien: `m1` y `m2` son aparentemente iguales, y sin embargo dice que son diferentes. También habrás visto que a la hora de imprimir el objeto saca una ristra de caracteres que no se entienden. Vamos a corregir todo esto por separado.

- Para poder comparar bien los objetos `MiObject` entre sí necesitamos definir en esta clase su propio método `equals`, que redefina el que viene heredado de `Object`. Así que coloca un método `equals` en esta clase:

```

public boolean equals(Object o)
{
    ...
}

```

y rellénalo de forma adecuada para que devuelva `true` si el objeto `o` es igual al actual (`this`), es decir, si tiene su campo entero y su campo cadena igual al actual.

AYUDA: observa que tendrás que convertir el objeto `o` a tipo `MiObject` para poder comparar. El método `equals` exige que el objeto que se le pasa como parámetro sea SIEMPRE de tipo `Object`, para poder hacer bien la herencia y sobrescritura del método `equals` original. Después, en el código, deberemos convertir este objeto al tipo concreto con que trabajemos:

```
MiObject mo = (MiObject) o;
```

- Para poder imprimir algo entendible al indicar que se imprima el objeto, debemos redefinir su método *toString*. Añade un método *toString* al código de *MiObject*:

```
public String toString()
{
    ...
}
```

y haz que devuelva una cadena conteniendo el número entero, una coma, y la cadena del objeto *MiObject*.

- Ejecuta de nuevo el programa, y comprueba que funciona correctamente.

La clase System

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

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

Otros métodos útiles que encontramos son:

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

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

```
void gc()
```

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

```
long currentTimeMillis()
```

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

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

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

```
String getProperty(String propiedad)
```

El sistema tiene una serie de propiedades, identificadas por un nombre. A

continuación mostramos una lista con diferentes nombres de propiedades del sistema, y qué significan:

Clave	Contenido
file.separator	Separador entre directorios en la ruta de los ficheros. Por ejemplo "/" en UNIX.
java.class.path	Classpath de Java
java.class.version	Versión de las clases de Java
java.home	Directorio donde está instalado Java
java.vendor	Empresa desarrolladora de la implementación de la plataforma Java instalada
java.vendor.url	URL de la empresa
java.version	Versión de Java
line.separator	Separador de fin de líneas utilizado
os.arch	Arquitectura del sistema operativo
os.name	Nombre del sistema operativo
os.version	Versión del sistema operativo
path.separator	Separador entre los distintos elementos de una variable de entorno tipo PATH. Por ejemplo ":"
user.dir	Directorio actual
user.home	Directorio de inicio del usuario actual
user.name	Nombre de la cuenta del usuario actual

Por ejemplo, si hacemos:

```
String barra = System.getProperty("file.separator");
```

estaremos obteniendo el símbolo con el que el sistema operativo separa los nombres de carpetas, subcarpetas y ficheros (será "/" en Linux y "\" en Windows).

Otro ejemplo:

```
System.out.println(System.getProperty("line.separator"));
```

en este caso, estamos sacando por pantalla el salto de línea propio del sistema operativo. Esto puede sernos útil si, como veremos más adelante, imprimimos este símbolo en un fichero, para así hacer los saltos de línea de acuerdo con el sistema

operativo en el que estemos. Si abris un fichero de texto Windows en Linux, veréis que los finales de línea contienen caracteres extraños (una M mayúscula con otro color). Esto es porque la secuencia de caracteres que emplea Windows para indicar fin de línea es diferente a la empleada en Linux, y los caracteres que no reconoce los muestra.

Podemos obtener todas las propiedades del sistema en un objeto *Properties*:

```
Properties getProperties()
```

Este objeto, que veremos al hablar de entrada/salida, contiene una lista de elementos, en este caso las propiedades del sistema, que podemos fácilmente guardar y leer. En el tema de entrada/salida se explicará cómo trabajar con este tipo de datos.

La clase Runtime

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

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

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

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

donde *comando* puede ser una cadena con el comando entero (incluyendo parámetros) que queremos ejecutar, o un array donde se separen el comando y cada parámetro, u otras posibilidades (consultad el API para ver todas las variantes). De esta forma podremos invocar programas externos desde nuestra aplicación Java.

Ejercicio 3: Ejecutando otros programas

En la clase *sesion06.Ej6* vamos a ejecutar una aplicación externa a Java. Por ejemplo, vamos a abrir el navegador Internet Explorer, y el bloc de notas.

- Para ello, en el *main* (podría ser en cualquier método, pero aprovecharemos este, ya que lo tenemos), deberemos crearnos un objeto de tipo *Runtime*, de la forma que hemos explicado antes en los apuntes.
- Después, deberemos localizar dónde está el ejecutable que lanza el navegador, y el del bloc de notas. Normalmente se encuentran, respectivamente, en:

```
C:\Archivos de programa\Internet Explorer\iexplore.exe  
C:\Windows\notepad.exe
```

(la carpeta *C:/Windows* puede ser *C:/Winnt* si estás trabajando con Windows 2000 o NT). Guadaremos cada ruta en una cadena:

```
String ruta = "C:\\Archivos de programa\\Internet Explorer\\iexplore.exe";
```



```
String ruta2 = "C:\\Windows\\notepad.exe";
```

Observa que las barras invertidas `\` se deben poner dobles en las variables `String` (es una secuencia de escape para guardar la barra, igual que `\n` guarda un salto de línea). También funcionará si en lugar de barras invertidas ponemos barras simples `"`, y en este caso no hace falta duplicar:

```
String ruta = "C:/Archivos de programa/Internet Explorer/iexplore.exe";  
String ruta2 = "C:/Windows/notepad.exe";
```

- Después, utilizaremos el método `exec` del *Runtime* para que ejecute primero un comando y después el otro (pondremos dos líneas `exec`, una para cada comando, y una a continuación de la otra). Probablemente necesites capturar alguna excepción cuando ejecutes los `exec`.
- Cuando lo tengas todo, ejecuta el programa, y comprueba si se abren las aplicaciones. ¿Se abren las dos a la vez o primero una, y cuando la cierras, la otra?
- Observa que la aplicación Java TERMINA después de abrir las ventanas de los programas que le hemos dicho. Esto es porque se crean hilos independientes para lanzar estas aplicaciones, y como el hilo principal ya no tiene nada más que hacer, termina.
- Si queremos secuenciar la ejecución de programas externos (es decir, que no empiece uno hasta que no haya terminado el anterior), debemos valernos de lo siguiente:

Supongamos que tenemos un objeto *Runtime* creado, y listo para ejecutar dos comandos guardados en las cadenas `com1` y `com2`:

```
Runtime rt = Runtime.getRuntime();  
String com1 = "C:/comando1.exe";  
String com2 = "C:/comando2.exe";
```

Si hacemos un `exec` con el primero y luego un `exec` con el segundo, se ejecutarán los dos a la vez, como habrás podido comprobar:

```
rt.exec(com1);  
rt.exec(com2);
```

Observa que la llamada a `exec` realmente devuelve un objeto de tipo *Process* (consulta el API). Podemos quedarnos con este objeto, y hacer que el programa actual (el hilo principal de Java) espere a que termine ese proceso *Process* antes de pasar a la siguiente línea de código:

```
Process p = rt.exec(com1);  
p.waitFor();  
rt.exec(com2);
```

Aplica este esquema sobre el ejercicio, para que primero se abra el navegador, y cuando lo cierras, se abra el bloc de notas. Captura, como antes, las excepciones que se te pidan al compilar.

La clase Math

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

Fechas y horas

La clase **java.util.Date** nos servirá para almacenar una determinada fecha y hora, que podrán ser las actuales, o las de un momento determinado (por ejemplo, el 3 de junio de 2002, a las 13:32 horas).

Si queremos obtener la fecha actual, construimos un objeto *Date* sin parámetros y ya está:

```
Date d = new Date();
```

Si queremos construir una fecha y hora concretas, indicando día, mes, año, hora, minuto o segundo, veréis en la API que la clase *Date* tiene constructores y métodos para establecer todos estos valores por separado:

```
public Date(int año, int mes, int día);
public Date(int año, int mes, int día, int hora, int minuto, int segundo);
public void setYear(int año);
public void setHours(int hora);
...
```

y construir así una fecha diferente. Sin embargo, consultando la API, veréis que todos estos métodos están **deprecated** (desaconsejados), y la documentación indica que *Date* sólo debe utilizarse para obtener la fecha actual (de la forma que hemos visto antes), y se debe utilizar la clase **java.util.Calendar** para especificar otras fechas concretas. Además, la clase **Calendar** nos servirá para realizar operaciones con fechas, comparar fechas, u obtener distintas representaciones para mostrar la fecha en nuestra aplicación.

Para ello, primero deberemos obtener una instancia de un objeto de tipo *Calendar*, con:

```
Calendar c = Calendar.getInstance();
```

Una vez obtenido, el objeto *Calendar* almacenará la fecha y hora actuales, pero luego podremos construir una fecha propia utilizando métodos como:

```
void set(int anyo, int mes, int dia, int hora, int minuto, int segundo)
```

y otros similares. Veréis que hay varias modalidades del método *set*, y otros alternativos, como *setTime*, *setTimeInMillis*, etc.

También podremos obtener por partes los elementos de la fecha, llamando al método *get*, e indicándole como parámetro qué parte queremos obtener. Para ello, la clase *Calendar* dispone de varias constantes para indicar cada una de las partes por separado.

```
int anyo = c.get(Calendar.YEAR);
```

```
int mes = c.get(Calendar.MONTH) + 1;
int dia = c.get(Calendar.DAY_OF_MONTH);
int hora = c.get(Calendar.HOUR_OF_DAY);
int minuto = c.get(Calendar.MINUTE);
int segundo = c.get(Calendar.SECOND);
```

```
System.out.println ("La fecha es " + dia + "/" + mes + "/" + anyo + ", " + hora +
```

Observa que luego podemos tomar las partes que necesitemos y representarlas como queramos. Observa también que cuando obtenemos el mes a través de la constante *Calendar.MONTH*, el mes se nos devuelve con valores entre 0 y 11, por lo que le deberemos sumar 1 para poner el número de mes correcto.

Dando formato

Muchas veces queremos sacar por pantalla o en un cuadro de texto un valor numérico, o una fecha, y no tenemos forma cómoda de especificar con qué formato queremos mostrarlos: si con 2 decimales, o si poniendo el año con 4 dígitos... etc.

Para esto Java pone a nuestra disposición clases como **java.text.NumberFormat** (para formatear números) o **java.text.DateFormat** (para formatear fechas).

Formateando números

Para poder formatear números, primero debemos obtener una instancia del objeto *NumberFormat*. Normalmente suele hacerse con:

```
NumberFormat nf = NumberFormat.getInstance();
```

Una vez obtenido, la clase *NumberFormat* tiene diversos métodos para indicar cómo queremos que salga el número. Por ejemplo:

```
setMinimumIntegerDigits(int n);
```

Establece que el mínimo número de cifras enteras que hay que mostrar sea de *n* cifras. Si por ejemplo *n* fuese 4, y queremos sacar un número como 23, lo que se mostraría finalmente sería "0023".

```
setMaximumIntegerDigits(int n);
```

Parecido al anterior, pero a la inversa: establece que el máximo número de cifras enteras que hay que mostrar sea de *n* cifras. Si queremos mostrar 4567 y *n* es 3, en este caso mostraría "567".

```
setMinimumFractionDigits(int n);
setMaximumFractionDigits(int n);
```

Similares a los dos métodos anteriores, pero para establecer cuántas cifras decimales queremos que tenga el número como mínimo y como máximo.

```
setGroupingUsed(boolean grouping);
```

El *grouping used* indica si queremos que se agrupen los dígitos para indicar los miles, millones, etc, o si queremos simplemente mostrar la secuencia de dígitos que componen el número. Si le pasamos *true* a este método, un número como 4567 se

representará como "4.567", y si le pasamos *false* lo representará tal cual está, como "4567". Podemos ver si está activado o no este flag booleano en cada momento, con:

```
boolean isGroupingUsed(boolean grouping);
```

La idea es llamar a estos métodos después de haber instanciado nuestro objeto *NumberFormat*, y una vez lo tengamos todo configurado, formatear el número o números que queramos llamando al método *format* de *NumberFormat* (veréis que hay varias formas de utilizarlo en el API). Por ejemplo:

```
double num1 = 1623.5;

NumberFormat nf = NumberFormat.getInstance();

nf.setMinimumFractionDigits(3);
nf.setMinimumIntegerDigits(5);
nf.setGroupingUsed(true);

String valorFormateado vf = nf.format(num1);
System.out.println(vf);
```

mostraría por pantalla el número "01.623,500".

Formateando fechas

Al igual que con *NumberFormat*, para poder formatear fechas también deberemos obtener previamente una instancia de objeto *DateFormat*. Podríamos utilizar el método *getInstance()* de *DateFormat*, al igual que hacemos con *NumberFormat*. Sin embargo, nos vamos a valer de la subclase **java.text.SimpleDateFormat**, que nos permitirá dar un formato más cómodo a las fechas:

```
DateFormat df = new SimpleDateFormat(String patron);
```

El *patron* es una cadena de texto que indicará cómo se debe presentar la fecha. Algunos ejemplos:

```
"dd / MM / yyyy"
```

muestra el día con dos dígitos, una barra, el mes con dos dígitos, una barra y el año con 4 dígitos.

```
"dd - MMMM - yyyy, hh:mm:ss"
```

muestra el día con dos dígitos, un guión, el mes con letras, un guión, el año con 4 dígitos, coma, y luego la hora actual, dos puntos, minutos actuales, dos puntos y segundos actuales.

A la luz de los anteriores ejemplos, podemos extraer que, dentro de la cadena del patrón:

- **d** la utilizaremos para colocar el día de la fecha. Pondremos tantas *d* como dígitos queramos que tenga (como mínimo) el día.
- **M** la utilizaremos para colocar el mes de la fecha. Pondremos tantas *M* como dígitos queramos que tenga (como mínimo) el mes. Si ponemos 3 o más *M*, el mes no se representará mediante dígitos, sino mediante su nombre (enero,

febrero, en lugar de 01, 02...)

- **y** la utilizamos para colocar el año de la fecha. Pondremos tantas como dígitos queramos que tenga (como mínimo) el año.
- **h** la utilizamos para colocar la hora actual.
- **m** la utilizamos para colocar los minutos actuales (no confundir con M mayúscula, utilizada para el mes)
- **s** la utilizamos para colocar los segundos.

Podéis consultar la API de *SimpleDateFormat* para más información sobre otros posibles caracteres. Aquí ponemos un ejemplo de uso, combinando lo anterior:

```
DateFormat df = new SimpleDateFormat("dd - MMMM - yyyy, hh:mm:ss");

// Fecha actual
Date d = new Date();

// Sacará la fecha actual con el formato indicado, por ejemplo: "05 - julio - 200
System.out.println (df.format(d));
```

Ejercicio 4: Practicar con Math y formatear números

Utiliza el método *main* de la clase *sesion06.Ej7* de la plantilla para calcular el logaritmo neperiano del número real (*double*) que se le pase como parámetro. Guarda ese número en una variable de tipo *double* y utiliza después un *NumberFormat* para mostrarla por pantalla, con un mínimo de 3 cifras decimales y un máximo de 5, y sin punto para separar los miles (es decir, sin *grouping used*).

AYUDA: para calcular el logaritmo deberás utilizar el método *log* de la clase *Math*.

Ejercicio 5: Practicar con fechas

Haz que la clase *sesion06.Ej8* lance un hilo que cada 200 ms esté calculando la fecha actual, y guardándola en la variable global *Calendar* que tiene *Ej8*. El hilo principal (*Ej8*) estará en un bucle infinito donde dormirá 1 segundo, y luego mostrará por pantalla la fecha que tenga su campo *Calendar*, con el siguiente formato:

<día> del <mes_en_cifra> de <año>, a las <hora>:<minuto>:<segundo>

por ejemplo:

23 del 03 de 2005, a las 21:15:23

AYUDA: para colocar texto dentro del patrón de la fecha (por ejemplo, 'del', o 'a las'), colócalo con comillas simples, insertándolo dentro del patrón con comillas dobles. Por ejemplo:

```
"dd 'del' MM 'de' yyyy"
```

El funcionamiento del hilo principal, una vez lanzado el segundo hilo, deberá ser el siguiente:

```
while (true)
{
    dormir 1000 milisegundos
```

```
        imprimir por pantalla la fecha del campo c (el otro hilo se encargará de actualizarlo)
    }
```

Este código deberá estar en el constructor de *Ej8*, justo después de crear y lanzar un hilo (*Thread*), que en su método *run* hará lo siguiente:

```
while (true)
{
    dormir 200 milisegundos
    actualizar el valor del campo c
}
```

recuerda que para poder trabajar con el hilo, deberás hacer que la clase principal implemente *Runnable*, y luego definir un método *run* con el código del segundo hilo, tal y como se ha indicado antes.

NOTAS:

- Para actualizar el valor del campo *c* en el hilo secundario, simplemente obtendremos una nueva instancia de *Calendar*:

```
c = Calendar.getInstance();
```

- Para imprimir la fecha formateada por pantalla, haremos algo como lo siguiente:

```
DateFormat df = ... // configurar el formato del DateFormat
...
System.out.print(df.format(c.getTime()) + "\r");
```

El colocar el "\r" al final hará que al volver a imprimir machaque la misma línea una y otra vez, con lo que quedará más elegante que ver cada vez una línea más abajo en la pantalla. Esta característica no funcionará bien si la salida la veis en la consola de Eclipse, donde probablemente sacará una línea para cada nueva fecha. No os preocupéis por eso, simplemente probad que el programa funciona.

Otras clases

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

Por ejemplo, podemos encontrar la clase **Currency** con información monetaria. La clase **Locale** almacena información sobre una determinada región del mundo, por lo que podremos utilizar esta clase junto a las anteriores para obtener la moneda de una determinada zona, o las diferencias horarias y de representación de fechas.

Optimización de código

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

- No instanciar (crear) más objetos de los necesarios. Es una buena práctica

para la eficiencia temporal del código reutilizar los objetos que tenemos ya instanciados siempre que sea posible, ya que consume tiempo tanto instanciar nuevos objetos, como después limpiar de la memoria los objetos que ya no se necesitan por parte del colector de basura.

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

PARA ENTREGAR

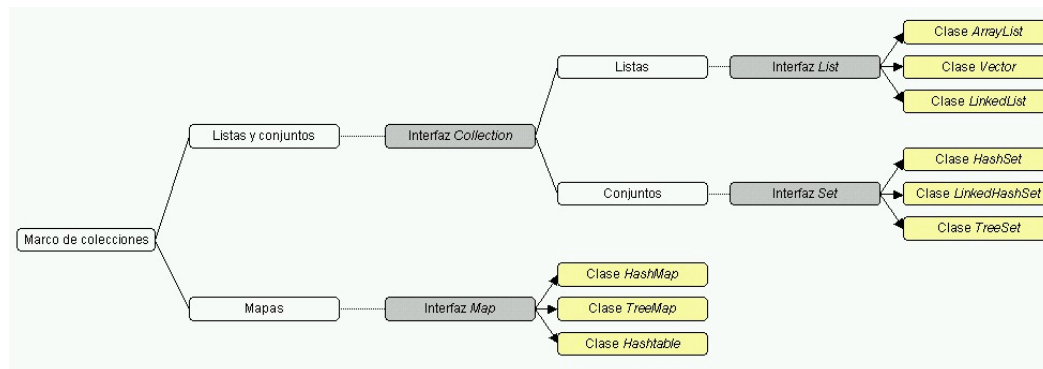
Guarda en la carpeta **modulo2** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**Ej4**, **Ej5**, **Ej6**, **Ej7** y **Ej8**), dentro del paquete **sesion06**, cada uno con las modificaciones que se han ido solicitando.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.

Sesión 7. Colecciones

Esta sesión la vamos a dedicar a estudiar las diferentes formas que tenemos en Java de almacenar conjuntos de objetos, para luego recorrerlos, modificar los elementos del conjunto, reordenarlos, etc. En definitiva, vamos a hablar de los distintos tipos de **colecciones**.

Todo el marco de colecciones está formado por clases e interfaces que se encuentran dentro del paquete **java.util**. Distinguimos las siguientes subcategorías:



Veremos que hay dos tipos de colecciones principales:

- Las **colecciones** propiamente dichas, que son grupos de elementos, y que parten de una interfaz padre que es **java.util.Collection**. Dentro de este grupo, distinguimos dos subtipos:
 - Las **listas**, que son secuencias de elementos. Todas parten de una interfaz padre, **java.util.List**, y dependiendo de la implementación de dicha secuencia, tendremos un tipo concreto de lista, como los *ArrayLists*, los *Vectores*, etc.
 - Los **conjuntos**, que son agrupaciones de elementos donde no puede haber repetidos. También parten de una interfaz padre, **java.util.Set**, y según el tipo de implementación, también tendremos tipos concretos de conjuntos, que veremos a continuación.
- Los **mapas**, son asociaciones de valores a claves, de forma que se tiene una secuencia de elementos *clave=valor*. Se tiene una interfaz padre, que es **java.util.Map**, y dependiendo de la implementación también tendremos subtipos concretos, como por ejemplo las tablas hash.

Las distintas interfaces de las que parten los tipos concretos proporcionan métodos para acceder a la colección de elementos, o al mapa, y que podremos utilizar para cualquier tipo de datos que implemente la correspondiente interfaz. Esto provoca un *polimorfismo* en la estructura, puesto que se proporciona una serie de métodos que pueden ser utilizados para acceder a distintos tipos de datos. Por ejemplo, un operador *add* utilizado para añadir un elemento, podrá ser empleado tanto si estamos trabajando con una lista enlazada, con un array, o con un conjunto.

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

- Interfaces para distintos tipos de datos: Definirán las operaciones que se pueden realizar con dichos tipos de datos. Podemos encontrar aquí la

interfaz para cualquier colección de datos, y de manera más concreta para listas (secuencias) de datos, conjuntos, etc.

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

Enumeraciones e iteradores

Antes de ver los distintos tipos de elementos que ofrece el marco de colecciones, comenzaremos viendo dos elementos utilizados comúnmente en Java para acceder a colecciones de datos. El primero de ellos es la **enumeración**, definida mediante la interfaz **Enumeration**, nos permite consultar los elementos que contiene una colección de datos. Muchos métodos de clases Java que deben devolver múltiples valores, lo que hacen es devolvernos una enumeración que podremos consultar mediante los métodos que ofrece dicha interfaz.

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

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

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

```
enum.hasMoreElements()
```

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

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

Vemos como en este bucle se van leyendo y procesando elementos de la enumeración uno a uno mientras queden elementos por leer en ella. Notar también que el método *nextElement* devuelve un objeto *Object* genérico, que nosotros deberemos convertir (mediante cast) al tipo de datos que estemos manejando:

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

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

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

Que nos devuelve el siguiente elemento a leer por el iterador. Al igual que en el caso de la enumeración, deberemos convertir lo que devuelve al tipo de datos que nos interese. Para saber si quedan más elementos que leer tenemos el método:

```
iter.hasNext()
```

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

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

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

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

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

Colecciones genéricas

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

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

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

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

```
void clear()
```

Elimina todos los elementos de la colección.

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

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

`boolean isEmpty()`

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

`Iterator iterator()`

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

`boolean remove(Object o)`

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

`int size()`

Nos devuelve el número de elementos que contiene la colección.

`Object [] toArray()`

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

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

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

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

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

Listas de elementos

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

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

`void add(int indice, Object obj)`

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

Object **get(int indice)**

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

int **indexOf(Object obj)**

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

Object **remove(int indice)**

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

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

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

int **size()**

Obtiene el número de elementos que hay en la lista.

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

ArrayList

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

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

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

Vector

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

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

Sin embargo, si trabajamos con versiones previas de JDK, hemos de tener en

cuenta que dicha interfaz no existía, y por lo tanto esta versión previa del vector no contará con los métodos definidos en ella. Además de los nuevos métodos que incorporó de *List*, los métodos propios del vector para acceder a su contenido, que han existido desde las primeras versiones, son los siguientes:

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

Añade un elemento al final del vector.

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

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

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

Inserta un elemento en la posición indicada.

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

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

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

Elimina el elemento de la posición indicada en el índice.

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

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

```
int size()
```

Devuelve el número de elementos del vector. Este método lo incorpora también la interfaz *List*, pero ya existía previamente en *Vector*.

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

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

Ejercicio 1: Recorrido de vectores y listas

La clase *sesion07.Ej1* de la plantilla tiene un método *main*, donde hemos creado un objeto de tipo *Vector*, y le hemos añadido 10 cadenas: *Hola0*, *Hola1*, *Hola2*...*Hola9*. Con este vector deberás hacer lo siguiente:

- Primero, recorrerlo mediante el objeto *Enumeration* que puedes obtener del

propio vector. Si observas la API de *Vector*, verás que tiene un método:

```
Enumeration elements();
```

que devuelve un *Enumeration* para poder recorrer los elementos del vector. Se trata de que obtengas esa enumeración, y formes un bucle como se ha explicado en los apuntes, para recorrerla de principio a fin. Para cada elemento, saca su valor por pantalla (imprime la cadena).

- A continuación de lo anterior, haz otro recorrido del vector, pero esta vez utilizando su *Iterator*. Verás también en la API que el objeto *Vector* tiene un método:

```
Iterator iterator();
```

que devuelve un *Iterator* para poder recorrer los elementos del vector. Haz ahora otro bucle como el que se explica en los apuntes, para recorrer los elementos del vector, esta vez con el *Iterator*. Para cada elemento, vuelve a imprimir su valor por pantalla.

- Finalmente, tras los dos bucles anteriores, añade un tercer bucle, donde a mano vayas recorriendo todo el vector, accediendo a sus elementos, y sacándolos por pantalla. En este caso, ya no podrás utilizar los métodos *nextElement*, *hasNext*, ni similares que has utilizado en los bucles anteriores. Deberás ir posición por posición, accediendo al valor de esa posición del vector (puedes utilizar el método que prefieras: *get* o *elementAt*), y sacando el valor obtenido por pantalla.

Una vez tengas los tres bucles hechos, ejecuta el programa, y observa lo que saca cada uno de los bucles por pantalla. ¿Encuentras alguna diferencia en el comportamiento de cada uno? ¿Qué forma de recorrer el vector te resulta más cómoda de programar y por qué?

NOTA: algunas de las técnicas que has utilizado para recorrer el vector se pueden utilizar de la misma forma para recorrer otros tipos de listas. Por ejemplo, puedes obtener el *Iterator* de un *ArrayList* y recorrerlo, o ir elemento por elemento...

LinkedList

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

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

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

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

Object **getFirst()** / Object **getLast()**

Obtiene el primer / último objeto de la lista respectivamente.

Object **removeFirst()** / Object **removeLast()**

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

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

Ejercicio 2: Pruebas de eficiencia

La clase *sesion07.Ej2* contiene un método *main* que a su vez llama a dos métodos de la propia clase:

- El método *creaYBorraEnmedio* crea un *Vector* de 10.000 cadenas, y una *LinkedList* con otras 10.000 cadenas. Después, hace N inserciones y borrados en la parte **media** del *Vector* y del *LinkedList*, y compara los tiempos que se ha tardado en uno y otro tipo de datos en hacer todas las operaciones.
- El método *creaYBorraFinal* crea un *Vector* de 10.000 cadenas, y una *LinkedList* con otras 10.000 cadenas. Después, hace N inserciones y borrados en la parte **final** del *Vector* y del *LinkedList*, y compara los tiempos que se ha tardado en uno y otro tipo de datos en hacer todas las operaciones.

El método *main* prueba el primer método con N = 10.000 operaciones, y el segundo con N = 1.000.000 operaciones. Se pide:

- Ejecuta el programa y observa los tiempos de ejecución de *creaYBorraEnmedio*. ¿Qué conclusiones sacas?
- Observa también los tiempos de ejecución de *creaYBorraFinal*. ¿Qué conclusiones sacas en este otro caso?
- A la vista de los resultados... ¿en qué casos crees que es mejor utilizar *LinkedList*, y en qué otros no es aconsejable hacerlo?
- Finalmente, añade un tercer método *creaYBorraInicio* que haga lo mismo que los anteriores, pero haciendo las N inserciones y borrados por el **inicio** del *Vector* y de la *LinkedList* (para ésta, utiliza los métodos *addFirst* y *removeFirst*). Para este caso, haz que N sea de 1.000.000. ¿Qué conclusiones obtienes al ejecutar este tercer método?

Conjuntos

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

Los conjuntos se definen en la interfaz **Set**, a partir de la cual se construyen diferentes implementaciones. Todas tienen métodos genéricos basados en la interfaz padre:

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

Añade el objeto al conjunto y devuelve si lo ha podido añadir (*true*) o no (*false*), porque ya existía, o por otra razón.

```
boolean addAll(Collection c)
```

Añade todos los elementos de la colección que se le pasa como parámetro al conjunto. Devuelve si los ha podido añadir (*true*) o no (*false*). Notar que así podemos pasar cualquier tipo de colección (*Vector*, *ArrayList*, incluso otros tipos de conjuntos) a otros conjuntos.

```
void clear()
```

Borra todos los elementos del conjunto.

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

Devuelve *true* si el objeto *o* está en el conjunto, y *false* si no.

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

Elimina el objeto del conjunto y devuelve si lo ha podido borrar (*true*) o no (*false*).

Existen otros métodos que también se tenían en *Collection*, como *size*, *iterator*, *etc.* Consultad el API para una visión más detallada. A continuación explicamos por encima algunos subtipos de conjuntos.

HashSet

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

LinkedHashSet

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

TreeSet

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

Ejercicio 3: Trabajar con conjuntos

Aunque tienen su utilidad, normalmente los tipos de conjuntos no se suelen emplear demasiado a la hora de programar. Su programación es muy similar a la que pueda tener un *ArrayList* o un *Vector*, y siempre se tiende a utilizar estos, porque sus clases son más conocidas. Sin embargo, los conjuntos tienen su verdadera utilidad cuando queremos tener listas de elementos no repetidos. Muchos programadores tienden a hacer ellos "a mano" la comprobación de si está o no repetido, y con estas clases se facilitaría bastante la tarea.

La clase *sesion07.Ej3* recibe una cadena (sin espacios) como parámetro, y la añade dentro de sus campos *Vector* y *HashSet*. Dichos campos ya tienen insertadas las cadenas "a1", "a2" y "a3".

- Rellena los métodos *anyadeVector* y *anyadeConjunto* para que añadan al vector o al conjunto, respectivamente, el elemento que se les pasa como parámetro, SIEMPRE QUE NO EXISTA YA. Tras insertarlo, deberán imprimir por pantalla su contenido actualizado (para recorrer el conjunto *HashSet* deberás acceder a su *Iterator*, probablemente).
 - Prueba a ejecutar el programa, pasándole como parámetro tanto una cadena que no exista ("b1", por ejemplo), como otra que sí ("a2", por ejemplo). Comprueba que en los dos casos se hace la inserción cuando toca, y se sacan bien los datos por pantalla. ¿Observas alguna diferencia en el orden en que se muestran los datos por pantalla cuando se hacen nuevas inserciones? ¿A qué crees que puede deberse?
 - Comenta las ventajas e inconvenientes que encuentres a la hora de programar con tipos conjunto como *HashSet* frente a tipos lista como *Vector*.
-

Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no implementan la interfaz **Collection**, tal y como puede verse en el esquema planteado al inicio de esta sesión.

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

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

Object **get(Object clave)**

Nos devuelve el valor asociado a la clave indicada, o *null* si no existe dicha clave.

Object **put(Object clave, Object valor)**

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

Object **remove(Object clave)**

Elimina una clave, devolviendonos el valor que tenía dicha clave, o *null* si no existía.

Set **keySet()**

Nos devuelve el conjunto de claves registradas

int **size()**

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

Encontramos distintas implementaciones de los mapas:

HashMap

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

TreeMap

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

Hashtable

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

Enumeration **keys()**

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

Ejercicio 4: Ventajas de los mapas

Trabajar con mapas es la forma más eficiente y cómoda de almacenar pares *clave=valor*. La clase *sesion07.Ej4* contiene una subclase llamada *Parametro*, que utilizamos para guardar ciertos parámetros de configuración, y sus valores. Verás que esta clase tiene un campo *nombre* donde pondremos el nombre del parámetro, y otro *valor* con su valor.

La clase principal *Ej4* crea muchos parámetros de este tipo, y los almacena en un *ArrayList*. Finalmente, busca en dicho *ArrayList* el valor del parámetro cuya clave se le pasa en el *main*. Sacar un mensaje indicando en qué posición lo encontró, y luego imprime todos los *Parametros* por pantalla, sacando su nombre y su valor.

- Haz una clase *sesion07.Ej4Hash* que haga lo mismo, pero utilizando una

Hashtable en lugar de un *ArrayList*. Debes tener en cuenta lo siguiente:

- En esta clase no tendrás que usar la subclase *Parametro*, ya que podrás almacenar el nombre por un lado y el valor por el otro dentro de la tabla hash. Es decir, donde antes hacías:

```
ArrayList al = new ArrayList();
...
al.add(new Parametro("Clave1", "Valor1"));
```

ahora harás:

```
Hashtable ht = new Hashtable();
...
ht.put("Clave1", "Valor1");
```

- A la hora de buscar el elemento en la hash ya no necesitas ningún bucle. El método *get* de la *Hashtable* te permite obtener un valor si le das el nombre con que lo guardaste. Te devolverá el objeto asociado a ese nombre (como un *Object* que deberás convertir al tipo adecuado), o *null* si no lo encontró.

```
String valor = (String)(ht.get(nombre));
if (valor == null) ... else ...
```

En este caso no hace falta que indiques en qué posición encontraste al elemento, puesto que, como verás después, las tablas hash no mantienen las posiciones como te esperas.

- A la hora de imprimir todos los elementos por pantalla, una opción es obtener todo el listado de claves, y luego para cada una ir sacando su valor, e imprimir ambos. Para obtener un listado de todas las claves, tienes el siguiente método en *Hashtable*:

```
Enumeration keys();
```

te devuelve una enumeración de las claves. Luego útilízala para recorrerlas, y con cada una sacar su valor e imprimirlo:

```
Enumeration en = ht.keys();
while (en.hasMoreElements())
{
    String clave = (String)(en.nextElement());
    String valor = (String)(ht.get(clave));
    ... // Imprimir clave y valor por pantalla
}
```

- Comenta las conclusiones que obtienes tras haber hecho este ejercicio, y qué ventajas e inconvenientes encuentras a la hora de añadir elementos en la hash, y de recuperarlos.
- Observa cómo se imprimen los valores de la hash al sacarlos por pantalla. ¿Conservan el orden en que se introdujeron? ¿A qué crees que puede deberse?
- Si en lugar de trabajar con listas o tablas hash de 10 elementos fuesen de 1.000.000 de elementos, ¿quién se comportaría más eficientemente (*ArrayList* o *Hashtable*) y por qué? No hace falta que lo pruebes, haz una estimación basándote en lo que has visto hasta ahora.

Algoritmos con colecciones

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

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

Ordenar colecciones

Supongamos que queremos ordenar una lista *List*, que ya tengamos previamente creada y llena de elementos del tipo *String*. Haríamos algo como lo siguiente:

```
List miLista = new List();
miLista.add("Hola");
miLista.add("Adios");
miLista.add("Hasta luego");
Collections.sort(miLista);
```

Y obtendríamos el orden siguiente: "Adios", "Hasta luego", "Hola".

Podremos ordenar una lista de cualquier tipo de datos, incluso de objetos definidos por nosotros. Lo único que se necesita para que Java la sepa ordenar automáticamente es que el tipo de datos de la lista tenga métodos que permitan comparar elementos entre sí. ¿Cómo hacer eso? Basta con hacer que el tipo de datos implemente la interfaz **Comparable**, y que definamos en su clase un método **compareTo**, que indique, dados dos elementos, cuál es mayor, menor, o igual.

Por ejemplo, supongamos que tenemos objetos de tipo *MiClase*:

```
class MiClase
{
    String valor;

    public MiClase(String v)
    {
        valor = v;
    }
}
```

Luego hacemos una lista con objetos de este tipo, y queremos que se ordene alfabéticamente, según el campo *valor*:

```
List miLista = new List();
miLista.add(new MiClase("AAA"));
miLista.add(new MiClase("FFF"));
miLista.add(new MiClase("DDD"));
Collections.sort(miLista);
```

Si probamos un código como este, la ordenación no funcionará, porque Java no sabe cómo queremos ordenar los objetos de tipo *MiClase*. Para solucionarlo,

hacemos que *MiClase* implemente la interfaz *Comparable*, y definimos un método *compareTo* (*Object o*) que:

- devolverá -1 si el objeto actual (*this*) es menor que el objeto *o* (según el criterio que nosotros elijamos, por ejemplo, comparar por el campo *valor* alfabéticamente).
- devolverá 0 si el objeto actual es igual que el objeto *o* (según el mismo criterio).
- devolverá 1 si el objeto actual es mayor que el objeto *o* (según el mismo criterio).

En nuestra clase, quedaría de la siguiente forma:

```
class MiClase implements Comparable
{
    String valor;

    public MiClase(String v)
    {
        valor = v;
    }

    public int compareTo(Object o)
    {
        MiClase mcAux = (MiClase)o;
        return this.valor.compareTo(mcAux.valor);
    }
}
```

Notar que la clase *String* ya implementa la interfaz *Comparable*, y por tanto podemos comparar cadenas entre sí llamando a su *compareTo*. Por tanto, podemos aprovechar lo que devuelva esta llamada entre los dos campos *valor*, para devolver lo mismo en nuestra *MiClase*. A fin de cuentas, el orden que tendrán estos objetos *MiClase* será el mismo que internamente tengan los objetos *valor*.

Una vez hayamos hecho estas modificaciones, el método *sort* que utilizábamos antes funcionará automáticamente, puesto que ya sabrá qué criterio seguir para comparar y ordenar objetos de tipo *MiClase*.

Ejercicio 5: Ordenar tus propios datos

La clase *datos.Persona* de la plantilla almacena los datos generales de una persona, como son su nombre, primer apellido, segundo apellido, dirección y teléfono. Tiene un constructor que se encarga de asignar todos esos campos, y métodos *get* y *set* para obtener sus valores o cambiarlos, respectivamente.

Además, al final tiene un método *main* que crea varios objetos de tipo *Persona*, los coloca en un *ArrayList*, y luego intenta ordenarlos llamando al método *Collections.sort*. Sin embargo, de momento el método no funciona (probablemente salte una excepción, porque no sabe cómo comparar los elementos de la lista).

Haz las modificaciones necesarias en la clase para que el método ordene correctamente. Queremos que se siga el siguiente criterio de ordenación:

- Ordenar de menor a mayor, según el primer apellido
- Si el primer apellido coincide, ordenar de menor a mayor según el segundo apellido
- Si también coincide, ordenar de menor a mayor por el nombre. Si también

coinciden, se considerará que los nombres son iguales en orden.

Comprueba, una vez lo tengas hecho, que la secuencia que saca el programa tras ordenar es la correcta:

```
Elemento 1: "Bravo Murillo, Manuel, C/La Huerta - 22, 965123456"
Elemento 2: "García Hernández, Marta, C/Aloma - 22, 634253456"
Elemento 3: "García Hernández, Rafael, C/Aloma - 1, 601123546"
Elemento 4: "García Rodríguez, Carolina, Avda. Doctor Rico - 25, 661228844"
Elemento 5: "Simón Mas, Eva, Camino del Prado - 30, 966124627"
```

Imagina que queremos cambiar el criterio de ordenación, y ahora queremos ordenar de mayor a menor por el nombre. ¿Qué cambios tendríamos que hacer? No los hagas, simplemente déjalos indicados en la respuesta a esta pregunta.

Wrappers

Ya hemos visto con anterioridad que ciertos objetos de Java llevan asociado un *wrapper*, es decir, otra clase u objeto que los encapsula, y que permite trabajar con ellos en otros ámbitos.

Aparte de los algoritmos comentados en el apartado anterior, la clase **Collections** aporta otros métodos para cambiar ciertas propiedades de las listas. Estos métodos nos los proporcionan los denominados *wrappers* de los distintos tipos de colecciones. Estos *wrappers* son objetos que 'envuelven' al objeto de nuestra colección, pudiendo de esta forma hacer que la colección esté sincronizada, o que la colección pase a ser de solo lectura.

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

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

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

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

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

```
Collection synchronizedCollection(Collection c)
List synchronizedList(List l)
Set synchronizedSet(Set s)
Map synchronizedMap(Map m)
SortedSet synchronizedSortedSet(SortedSet ss)
SortedMap synchronizedSortedMap(SortedMap sm)
```

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

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

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

PARA ENTREGAR

Guarda en la carpeta **modulo3** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**sesion07.Ej1**, **sesion07.Ej2**, **sesion07.Ej3**, **sesion07.Ej4** y **datos.Persona**), dentro de los paquetes correspondientes, cada uno con las modificaciones que se han ido solicitando.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.

Sesión 8. Entrada/salida

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

Flujos de datos de entrada/salida

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

- Flujos de caracteres
- Flujos de bytes

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

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

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

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

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

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

de los datos que viajan a través de ellos (con prefijo Filter), conversores datos (con prefijo Data), buffers de datos (con prefijo Buffered), preparados para la impresión de elementos (con prefijo Print), etc.

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

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

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

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

Entrada, salida y salida de error estándar

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

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

	Tipo	Objeto
Entrada estándar	InputStream	System.in
Salida estándar	PrintStream	System.out
Salida de error estándar	PrintStream	System.err

Para la entrada estándar vemos que se utiliza un objeto **InputStream** básico, sin embargo para la salida se utilizan objetos **PrintWriter** que facilitan la impresión de texto ofreciendo a parte del método común de bajo nivel *write(int b)* para escribir bytes, dos métodos más: *print(s)* y *println(s)*. Estas funciones nos permitirán escribir

cualquier cadena, tipo básico, o bien cualquier objeto que defina el método `toString()` que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

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

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

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

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

Acceso a ficheros

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

	Lectura	Escritura
Caracteres	FileReader	FileWriter
Binarios	FileInputStream	FileOutputStream

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

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

```
public void copia_fichero()
{
    int c;
    try
    {
        FileReader in = new FileReader("fuente.txt");
        FileWriter out = new FileWriter("destino.txt");
        while( (c = in.read()) != -1)
        {
            out.write(c);
        }
        in.close();
        out.close();
    } catch(FileNotFoundException e1) {
        System.err.println("Error: No se encuentra el fichero");
    } catch(IOException e2) {
        System.err.println("Error leyendo/escribiendo fichero");
    }
}
```

```
    }
}
```

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

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

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

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

Ejercicio 1: Lectura y escritura básicas

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

Echa un vistazo a la clase *sesion08.Ej6* 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 *Ej6* 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.

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).

- 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:

```
FileInputStream in = new FileInputStream("entrada.dat");
```

NOTA IMPORTANTE: vigilad dónde ponéis el fichero *entrada.dat*, porque puede que no lo encuentre. Una buena idea para que esta

línea de código os funcione es ponerlo en la carpeta raíz del proyecto, y no dentro del paquete *sesion08*. De todas formas, podéis ponerlo en cualquier parte, siempre que después sepáis cómo encontrarlo desde Java.

- 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:

```
FileOutputStream out = new FileOutputStream("salidaStream.dat");
```

Aquí tendremos que tener las mismas consideraciones que con el fichero de entrada, en cuanto a cómo localizarlo. En este caso no es tan importante, porque el fichero lo creará de todas formas, en un lugar u otro, pero debemos saber dónde lo va a crear. Con una línea como ésta, lo creará en la carpeta raíz del proyecto también.

- 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:

```
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?

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

```
in.close();
out.close();
```

- 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). Arréglalo y prueba el resultado.
- 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:

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

- 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
```

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).

- 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.

```
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 *PrintWriter* 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. Deberemos tener las mismas consideraciones que con el método anterior sobre dónde poner los ficheros para que el programa los encuentre.

- 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:

```
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 *readLine*. ¿Por qué se compara el dato que se lee con *null*?

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

```
br.close();
pw.close();
```

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

prueba el resultado.

- 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:

```
pw.print(cabecera);
```

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

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 *OutputStreams* (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.

Un caso particular: ficheros de propiedades

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

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

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

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

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

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

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

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

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

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

Propiedades del sistema

Como se ha visto en la sesión de utilidades, la clase **System** tiene métodos como **getProperty**, **setProperty** a los que se les pasa un nombre de propiedad del sistema, y permiten obtener o cambiar su valor, respectivamente. Además, tenemos un método

```
Properties getProperties();
```

que devuelve un objeto *Properties* con todas las propiedades del sistema. Podremos recorrerlas, ver cuáles son, y aprovechar sus valores en nuestros programas.

Ejercicio 2: Trabajar con propiedades

En este segundo ejercicio practicaremos el uso de ficheros de propiedades, y el uso de la entrada y salida estándares. Echa un vistazo a la clase *sesion08.Ej7* 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.

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.

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*:

```
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*.

Ahora ya tenemos en el objeto *p* todas las propiedades del fichero. Vamos a ir las 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:

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

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:

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

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:

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

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

```
in.close();
p.store(new FileOutputStream("prop.txt"), "Cabecera del fichero");
```

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

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.

Lectura de tokens

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

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

puede leer:

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

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

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

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

Un bucle de procesamiento básico será el siguiente:

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

Podemos distinguir tres tipos de caracteres:

Ordinarios (ordinaryChars)	Caracteres que forman parte de los <i>tokens</i> .
De palabra (wordChars)	Una secuencia formada enteramente por este tipo de caracteres se considerará una palabra.

De espacio en blanco (whitespaceChars)	Estos caracteres no son interpretados como <i>tokens</i> , simplemente se utilizan para separar <i>tokens</i> . Normalmente estos caracteres son el espacio, tabulador, y salto de línea.
---	---

Para establecer qué caracteres pertenecerán a cada uno de estos tipos utilizaremos los métodos *ordinaryChars*, *wordChars* y *whitespaceChars* del objeto

StreamTokenizer respectivamente. A cada uno de estos métodos le pasamos un rango de caracteres (según su código ASCII), que serán establecidos al tipo correspondiente al método que hayamos llamado. Por ejemplo, si queremos que una palabra sea una secuencia de cualquier carácter imprimible (con códigos ASCII desde 32 a 127) haremos lo siguiente:

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

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

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

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

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

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

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

Ejercicio 3: Leyendo matrices

Vamos a practicar la lectura de tokens de un fichero, y su almacenamiento para realizar alguna operación. Echa un vistazo a la clase *sesion08.Ej8* 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.

Lo que vamos a hacer es que el constructor acceda a un fichero (fichero *matriz.txt* de la plantilla) que tiene una matriz $m \times n$. 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.

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

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

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

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

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:

```
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?

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

```
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);
        }
    }
}

```

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. Utiliza un objeto *PrintWriter* para facilitar el volcado de la matriz al fichero.

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.

Acceso a ficheros o recursos dentro de un JAR

Los ficheros JAR son una forma de empaquetar clases Java y otros recursos en un solo fichero, con un formato similar a los ficheros TAR que se suelen utilizar en Linux. De hecho los comandos *jar* (que viene con Java y se utiliza para crear ficheros JAR) y *tar* (de Linux) tienen los mismos parámetros.

De esta forma podremos hacer un programa más portable, al usar un sólo fichero. También hay formas de hacer que el JAR auto-ejecute las clases que lleva dentro, con lo que aumenta la facilidad de uso.

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

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

```

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

```

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

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

actual.

Codificación de datos

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

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

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

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

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

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

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

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

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

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

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

Serialización de objetos

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

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject()` y `writeObject(Object obj)` respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de ser *serializables*.

Serán *serializables* aquellos objetos que implementan la interfaz `Serializable`. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase

pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

Ejercicio 4: Guardar datos personales

En este 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.

Vamos a retomar la clase *datos.Persona* que hicimos en la sesión 7. Por otro lado, tenemos en la plantilla de esta sesión la clase *io.LeeGuardaPersona*. Tiene dos métodos *leePersonas* y *guardaPersonas* que deberemos implementar:

- El método *leePersonas* deberá crear un *ObjectInputStream* y leer de un fichero que se le pase como parámetro una serie de objetos de tipo *datos.Persona*, y guardarlos en un *ArrayList*, que devolverá cuando termine de leer

```
ArrayList al = new ArrayList();
try
{
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fichero));
    while (true)
    {
        Persona p = (Persona)(ois.readObject());
        al.add(p);
    }
} catch (Exception e) {
}
return al;
```

Te parecerá raro utilizar un bucle infinito para leer el fichero. El motivo es sencillo. Cuando trabajamos con *ObjectInputStreams* no hay forma de saber cuándo acaba el fichero, porque leemos objetos complejos enteros, hasta que ya no hay más. En ese momento, se provoca una excepción, y el bucle terminará cuando dicha excepción salte, lo que indicará el fin de fichero. Después de eso, devolvemos la lista con los objetos que haya guardados y listo.

- El método *guardaPersonas* recibirá como parámetro un *ArrayList* con objetos de tipo *datos.Persona*, y un nombre de fichero, y deberá guardar los objetos *Persona* del *ArrayList* en el fichero que se le dice mediante un *ObjectOutputStream*.
- Para que todo esto te funcione bien, la clase *datos.Persona* debe ser serializable, es decir, debe implementar la interfaz *java.io.Serializable*. Haz que la implemente.
- Una vez lo tenga todo listo, utiliza el método *main* que viene con la clase para probar su funcionamiento. Dicho método crea un *ArrayList*, lo llena con algunas *Personas*, y luego llama a *guardaPersonas*. Aquí vuestro método deberá guardar los datos adecuadamente en fichero. Después, el *main* llama

al método *leePersonas* para leer las personas del fichero que se indique, y mostrar sus datos por pantalla. Ejecuta la clase cuando la tengas terminada, para ver si tus métodos funcionan como deben.

- Opcionalmente, haz que el método *main* de *LeeGuardaPersona* muestre ordenadamente las personas, según los criterios de ordenación vistos en la sesión 7 cuando hiciste la clase *Persona*.

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

PARA ENTREGAR

Guarda en la carpeta **modulo3** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**sesion08.Ej6**, **sesion08.Ej7**, **sesion08.Ej8** e **io.LeeGuardaPersona**), dentro de los paquetes correspondientes, cada uno con las modificaciones que se han ido solicitando.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.

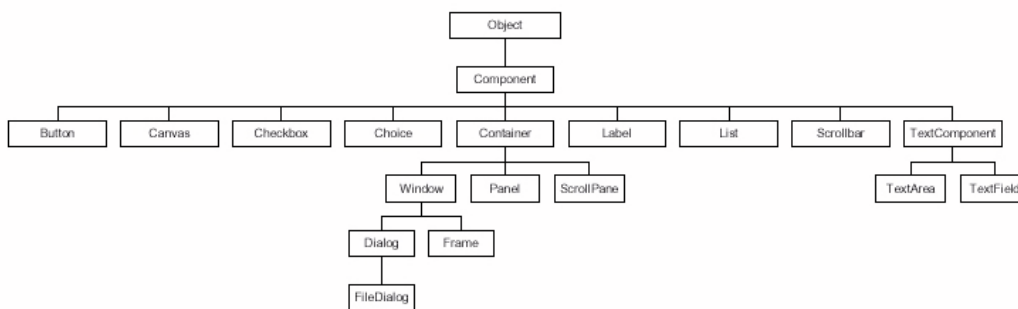
Sesión 9. Introducción a Swing

AWT

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

Resumen de controles AWT

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



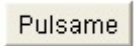
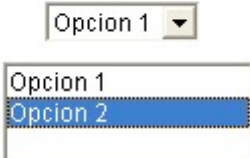
Los controles sólo se verán si los añadimos sobre un contenedor (un elemento de tipo *Container*, o cualquiera de sus subtipos). Para ello utilizamos el método **add(...)** del contenedor para añadir el control. Por ejemplo, si queremos añadir un botón a un *Panel*:

```

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

```

A continuación vamos a ver una descripción de cada uno de los controles que ofrece AWT.

<p>Component</p>	<p>La clase padre <i>Component</i> no se puede utilizar directamente. Es una clase abstracta, que proporciona algunos métodos útiles para sus subclases.</p>
<p>Botones</p> 	<p>Para emplear la clase Button, en el constructor simplemente indicamos el texto que queremos que tenga :</p> <pre>Button boton = new Button("Pulsame");</pre>
<p>Etiquetas</p> <p>Etiqueta</p>	<p>Para utilizar Label, el uso es muy similar al botón: se crea el objeto con el texto que queremos darle:</p> <pre>Label etiq = new Label("Etiqueta");</pre>
<p>Areas de dibujo</p>	<p>La clase Canvas se emplea para heredar de ella y crear componentes personalizados. Accediendo al objeto <i>Graphics</i> de los elementos podremos darle la apariencia que queramos: dibujar líneas, pegar imágenes, etc:</p> <pre>Panel p = new Panel(); p.getGraphics().drawLine(0, 0, 100, 100); p.getGraphics().drawImage(...);</pre>
<p>Casillas de verificación</p> <p><input type="checkbox"/> Mostrar subdirectorios</p>	<p>Checkbox se emplea para marcar o desmarcar opciones. Podremos tener controles aislados, o grupos de <i>Checkboxes</i> en un objeto CheckboxGroup, de forma que sólo una de las casillas del grupo pueda marcarse cada vez.</p> <pre>Checkbox cb = new Checkbox ("Mostrar subdirectorios", false); System.out.println ("Esta marcada: " + cb.getState());</pre>
<p>Listas</p> 	<p>Para utilizar una lista desplegable (objeto Choice), se crea el objeto y se añaden, con el método addItem(...), los elementos que queramos a la lista:</p> <pre>Choice ch = new Choice(); ch.addItem("Opcion 1"); ch.addItem("Opcion 2"); ...</pre>

Swing



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

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

- Los componentes *Swing* están escritos sin emplear código nativo, con lo que ofrecen más versatilidad multiplataforma (podemos dar a nuestra aplicación un aspecto que no dependa de la plataforma en que la estemos ejecutando).
- Los componentes *Swing* ofrecen más capacidades que los correspondientes *AWT*: los botones pueden mostrar imágenes, hay más facilidades para modificar la apariencia de los componentes, etc.
- Al mezclar componentes *Swing* y componentes *AWT* en una aplicación, se debe tener cuidado de emplear contenedores *AWT* con elementos *Swing*, puesto que los contenedores pueden solapar a los elementos (se colocan encima y no dejan ver el componente).

Resumen de controles Swing

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

JComponent	La clase padre para los componentes Swing es <i>JComponent</i> , paralela al <i>Component</i> de AWT.
Botones 	Se tienen botones normales (JButton), de verificación (JCheckBox), de radio (JRadioButton), etc, similares a los <i>Button</i> , <i>Checkbox</i> de AWT, pero con más posibilidades (se pueden añadir imágenes, etc).
Etiquetas 	Las etiquetas son JLabel , paralelas a las <i>Label</i> de AWT pero con más características propias (iconos, etc).
Cuadros de texto	Las clases JTextField y JTextArea representan los cuadros de texto en Swing, de forma parecida a los

	<p><i>TextField</i> y <i>TextArea</i> de AWT.</p>
<p>Listas</p> 	<p>Las clases JComboBox y JList se emplean para lo mismo que <i>Choice</i> y <i>List</i> en AWT.</p>
<p>Diálogos y ventanas</p> 	<p>Las clases JDialog (y sus derivadas) y JFrame se emplean para definir diálogos y ventanas. Se tienen algunos cuadros de diálogo específicos, para elegir ficheros (<i>JFileChooser</i>), para elegir colores (<i>JColorChooser</i>), etc.</p>
<p>Menús</p> 	<p>Con JMenu, JMenuBar, JMenuItem, se construyen los menús que se construian en AWT con <i>Menu</i>, <i>MenuBar</i> y <i>MenuItem</i>.</p>

Gestores de disposición

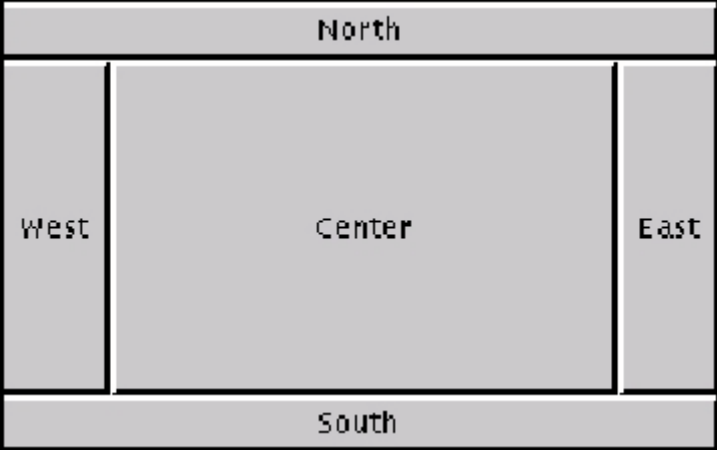


Para colocar los controles Java en los contenedores se hace uso de un determinado **gestor de disposición**. Dicho gestor indica cómo se colocarán los controles en el contenedor, siguiendo una determinada distribución. Para establecer qué gestor queremos, se emplea el método *setLayout(...)* del contenedor. Los gestores de disposición se encuentran dentro del paquete **java.awt**, pero nos servirán indistintamente para controles AWT y Swing. Por ejemplo:

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

JPanel p2 = new JPanel();
```

```
p2.setLayout(new BorderLayout());
```

Veremos ahora los gestores más importantes:

<p style="text-align: center;">BorderLayout</p>  <p style="text-align: center;">(gestor por defecto para contenedores tipo <i>Window</i>)</p>	<p>Divide el área del contenedor en cinco zonas: Norte (<i>NORTH</i>), Sur (<i>SOUTH</i>), Este (<i>EAST</i>), Oeste (<i>WEST</i>) y Centro (<i>CENTER</i>), de forma que al añadir componentes deberemos indicar en qué zona se colocará.</p> <pre>panel.setLayout(new BorderLayout()); Button btn = new Button("Pulsar"); panel.add(btn, BorderLayout.CENTER);</pre> <p>Al colocar un componente en una zona, se colocará sobre el que existiera anteriormente en dicha zona.</p>
<p style="text-align: center;">FlowLayout</p>  <p style="text-align: center;">(gestor por defecto para contenedores de tipo <i>Panel</i>)</p>	<p>Con este gestor, se colocan los componentes en fila, uno detrás de otro, con el tamaño preferido (por defecto) que se les haya dado. Si no cabe en una fila, se utilizan varias.</p> <pre>panel.setLayout(new FlowLayout()); panel.add(new Button("Pulsar"));</pre>
<p style="text-align: center;">GridLayout</p> 	<p>Este gestor sitúa los componentes en una cuadrícula, dividiendo el contenedor en celdas del mismo tamaño, de forma que el componente ocupe todo el tamaño de la celda.</p> <p>Se indica en el constructor el número de filas y de columnas. Luego, se añaden los componentes por orden (rellenando fila por fila, de izquierda a derecha).</p> <pre>panel.setLayout(new GridLayout(3, 2)); panel.add(new Button("Pulsar")); panel.add(new Label("Etiqueta"));</pre>
<p style="text-align: center;">Sin gestor</p>	<p>Si especificamos un gestor de layout, no necesitamos colocar a mano los componentes en el contenedor, con métodos como setBounds(...) , o setLocation(...).</p>

```
panel.setLayout(null);
Button btn = new Button ('
btn.setBounds(0, 0, 100, 30);
panel.add(btn);
```

Diferencias entre AWT y Swing en la disposición de elementos

Hay una diferencia en los gestores de disposición en Swing: para ciertos contenedores (*JFrames* y *JDialogs*), se debe acceder al *getContentPane()* del contenedor, antes de llamar a su método *setLayout* para establecer el gestor, y también antes de llamar a su método *add* para añadirle elementos. No ocurre así para los *JPanel*. Veamos algunos ejemplos:

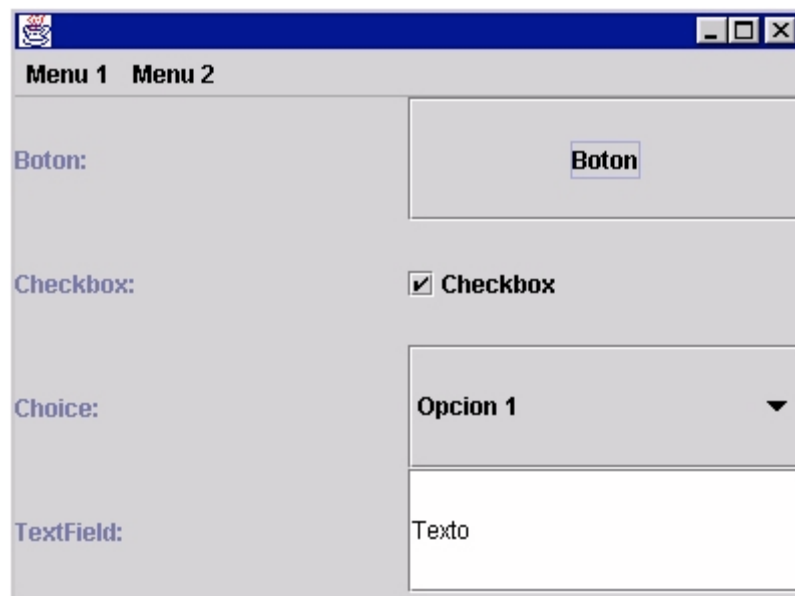
```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(new JButton("Hola"));

JFrame f = new JFrame();
f.getContentPane().setLayout(new BorderLayout());
f.getContentPane().add(new JButton("Hola"));

JDialog d = new JDialog();
d.getContentPane().setLayout(new FlowLayout());
d.getContentPane().add(new JButton("Hola"));
```

Esta diferencia desaparece con la versión 1.5 de Java, pero deberemos tenerla presente en versiones anteriores.

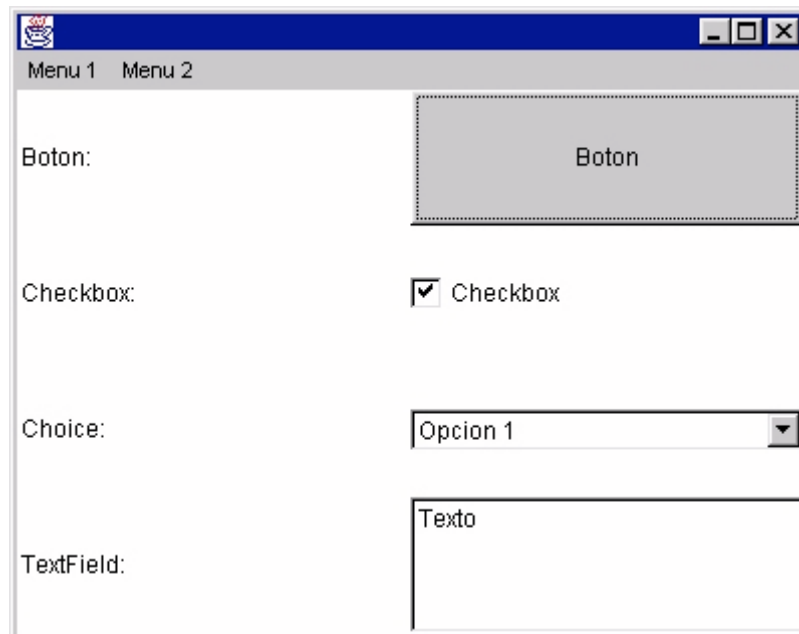
Ejemplo: Vemos el aspecto de algunos componentes de Swing, y el uso de gestores de disposición en este ejemplo:



[Código](#)

El código nos muestra cómo se crea una clase que es una ventana principal (hereda de *JFrame*), y define un gestor que es un *GridLayout*, con 4 filas y 2

columnas. En ellas vamos colocando etiquetas (*JLabel*), botones (*JButton*), casillas de verificación (*JCheckbox*), listas desplegables (*JComboBox*) y cuadros de texto (*JTextField*). Además, se crea un menú con diferentes opciones. El ejemplo sería muy similar en AWT, simplemente cambiando los controles por los equivalentes en AWT, y haciendo algún retoque a la hora de establecer gestores de disposición y añadir elementos a la ventana principal. Aquí tendríamos el ejemplo en AWT:

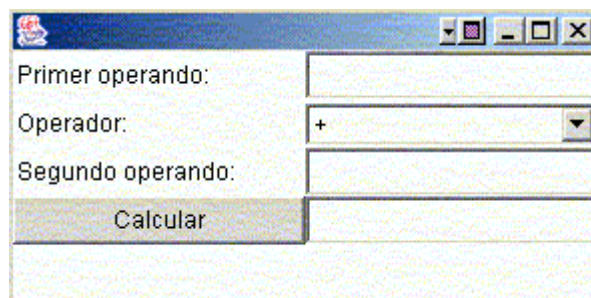


[Código](#)

Observad las diferencias entre una y otra aplicación, en cuanto a cómo se programan, y cómo se ven los componentes en pantalla (debido a la independencia de plataforma de Swing).

Ejercicio 1: Nuestra primera aplicación AWT

En la plantilla tenéis la clase *sesion09.Calculadora*. Pretendemos que sea una calculadora muy simplificada, con la siguiente apariencia:



En la casilla del primer operando pondremos la primera cifra para operar, después en el desplegable *Operador* elegiremos la operación a realizar (sumar, restar, multiplicar o dividir), y en la casilla del segundo operando pondremos el segundo operando de la operación. Finalmente, pulsando el botón de *Calcular*, en el último cuadro de texto (junto al botón) nos pondrá el resultado.

En este ejercicio UNICAMENTE vamos a dejar puestos los controles, para que queden como en la imagen. Verás que la clase 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.

- 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));
}
```

- 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:

```
// 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);
}
```

- 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).

Ejercicio 2: Nuestra primera aplicación Swing

La plantilla también tiene la clase *sesion09.JCalculadora*. Vamos a hacer algo parecido a lo que hemos hecho en el ejercicio anterior, pero en este caso para hacer una aplicación Swing.

Primer operando:	<input type="text"/>
Operador:	<input type="text" value="+"/>
Segundo operando:	<input type="text"/>
Calcular	<input type="text"/>

- 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 ...
```

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

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

- 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:");
...
```

- Prueba a compilar y ejecutar la clase... dará error si lo estás ejecutando con una versión inferior a Java 1.5. ¿A qué se debe el error?
- 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);
...
```

- Compila y comprueba que el programa se muestra como debe, aunque su apariencia sea distinta.

En el siguiente ejercicio completaremos este ejercicio para hacer que el programa "haga algo" (calcule las operaciones), aparte de mostrarse.

Modelo de Eventos en Java

Hasta ahora hemos visto qué tipos de elementos podemos colocar en una aplicación visual con AWT o Swing, y cómo colocarlos sobre los distintos contenedores que nos ofrece la librería. Pero sólo con esto nuestra aplicación no hace nada: no sabemos cómo emitir una determinada respuesta al pulsar un botón, o realizar una acción al seleccionar una opción del menú. Para definir todo esto se utilizan los llamados **eventos**.

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

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

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

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

Toda la gestión de eventos se lleva a cabo desde el paquete **java.awt.event**. Al igual que con los gestores de disposición, podemos utilizarlos indistintamente para AWT y para Swing. Además, Swing incorpora nuevos gestores de eventos en su paquete **javax.swing.event**, para implementar ciertas funcionalidades específicas sólo de Swing.

Modos de definir un oyente

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

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

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

```

    }

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

```

2. Definir otra clase aparte que implemente el oyente

```

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

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

```

3. Definir una instancia interna del oyente

```

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

```

Uso de los "adapters"

Algunos de los oyentes disponibles (como por ejemplo *MouseListener*, consultad su API) tienen varios métodos que hay que implementar si queremos definir el oyente. Este trabajo puede ser bastante pesado e innecesario si sólo queremos usar algunos métodos. Por ejemplo, si sólo queremos hacer algo al hacer click con el ratón, deberemos redefinir el método *mouseClicked*, pero deberíamos escribir también los métodos *mousePressed*, *mouseReleased*, etc, y dejarlos vacíos.

Una solución a esto es el uso de los *adapters*. Asociado a cada oyente con más de

un método hay una clase ...*Adapter* (para *MouseListener* está *MouseAdapter* , para *WindowListener* está *WindowAdapter*, etc). Estas clases implementan las interfaces con las que se asocian, de forma que se tienen los métodos implementados por defecto, y sólo tendremos que sobrescribir los que queramos modificar.

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

1. Mediante Listener:

```
class MiClase
{
    public MiClase()
    {
        ...
        JPanel panel = new JPanel();
        panel.addMouseListener(new MouseListener()
        {
            public void mouseClicked(MouseEvent e)
            {
                // Aquí va el código de la acción
            }

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

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

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

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

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

2. Mediante Adapter:

```
class MiClase
{
    public MiClase()
    {
        ...
        JPanel panel = new JPanel();
```

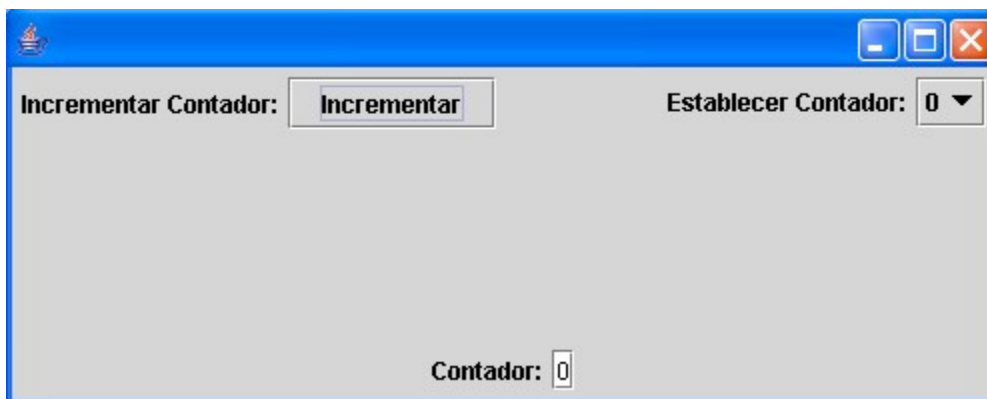
```

panel.addMouseListener(new MouseAdapter()
{
    public void mouseClicked(MouseEvent e)
    {
        // Aquí va el código de la acción
    }
});
...
}

```

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

Ejemplo: Vemos el uso de oyentes en este ejemplo:



[Código](#)

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

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

Ejercicio 3: Haciendo funcionar nuestra aplicación Swing

Retomemos la clase *sesion09.JCalculadora* que hemos empezado a hacer en un ejercicio previo. Ya tenemos puestos todos los controles, lo que nos queda es "hacer que el programa haga algo". Para ello vamos a definir los eventos.

- 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:

```
// 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 segun el operador seleccionado
            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");
        }
    }
});
```

- 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:

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

- Compila el programa... te dará errores de compilación.
- 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 JCalculadora()
{
    ...
    JTextField txtOp1 = new JTextField();
    ...
    JComboBox operadores = new JComboBox();
    ...
    JTextField txtOp2 = new JTextField();
    ...
    JTextField txtRes = new JTextField();
    ...
}
```

```
}
```

Por estas:

```
public JCalculadora()
{
    ...
    final JTextField txtOp1 = new JTextField();
    ...
    final JComboBox operadores = new JComboBox();
    ...
    final JTextField txtOp2 = new JTextField();
    ...
    final JTextField txtRes = new JTextField();
    ...
}
```

O bien colocarlas fuera del constructor, como variables globales:

```
public class JCalculadora ...
{
    JTextField txtOp1 = new JTextField();
    JComboBox operadores = new JComboBox();
    JTextField txtOp2 = new JTextField();
    JTextField txtRes = new JTextField();

    public JCalculadora()
    {
        ...
    }
    ...
}
```

- 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...)

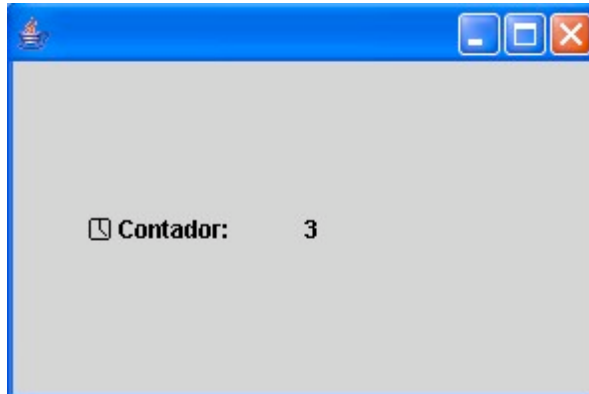
Otras características de Swing

Swing ofrece otras posibilidades, que se comentan brevemente:

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

queremos ejecutar en un momento determinado o con una periodicidad determinada.

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



[Código](#)

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

NOTA: en este caso se ha puesto un gif animado haciendo de reloj, pero la animación que tiene es independiente del *Timer* que hay en el programa, que sólo se utiliza para actualizar el valor del contador. Por casualidad, da el efecto de que se "animan" los dos a la vez.

Pasos generales para construir una aplicación gráfica

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

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

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

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

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

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

2. Colocar los controles en la ventana

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

```
public MiAplicacion()
{
    ...

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

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

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

3. Definir los eventos que sean necesarios

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

```
public MiAplicacion()
{
    ...

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

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

4. Mostrar la ventana

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

```

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

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

```

5. Definir otras subventanas o diálogos

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

Ejercicio 4: Una aplicación más compleja y completa

Para terminar, vamos a construir una aplicación que toque un poco de casi todo lo que hemos estado viendo en estas sesiones.

Vamos a hacer una aplicación Swing que nos sirva como agenda de contactos. Utilizaremos la clase *datos.Persona* que vimos en la sesión 7 para guardar los datos básicos de cada uno de nuestros contactos (nombre, apellido1, apellido2, direccion y teléfono), y los guardaremos en un fichero, ayudándonos de la clase *io.LeeGuardaPersona*, que hicimos en la sesión 8.

Lo que haremos en esta sesión es darle una apariencia gráfica a todo: construiremos una ventana (*JFrame*) en la clase ***gui.Agenda*** de la plantilla, que muestre todas las personas que tenemos guardadas en un determinado fichero (el fichero se lo pasaremos como parámetro a la aplicación a través de su método *main*).

La ventana podría tener una apariencia como la siguiente:

Rafael García		
Nombre:	Rafael	
Apellido 1:	García	
Apellido 2:	Hernández	
Direccion:	C/Aloma - 1	
Telefono:	601123546	
Ver Datos	Modificar Datos	Añadir Persona
Quitar Persona	Guardar Fichero	Borrar Cuadros

En la ventana, arriba tendremos una lista con nombres de personas, y al seleccionar alguna, abajo en los cuadros de texto aparecerá desglosada la información sobre él. En la parte inferior tendremos una serie de botones que nos servirán para diferentes tareas:

- **Ver Datos:** al pulsarlo mostrará en los cuadros de texto centrales los datos desglosados de la persona que tengamos seleccionada arriba.
- **Modificar Datos:** al pulsarlo modificará los datos que hay guardados de la persona seleccionada, por los que hay en ese momento en los cuadros de texto.
- **Añadir Persona:** añadirá una nueva persona en la lista, con los datos que haya escritos en los cuadros de texto.
- **Quitar Persona:** eliminará de la lista la persona actualmente seleccionada, con todos sus datos
- **Guardar Fichero:** guardará en el mismo fichero que se le pase como entrada los datos actualizados de la lista de personas.
- **Borrar Cuadros:** pondrá en blanco los cuadros de texto centrales, para cuando queramos introducir nueva información

Se os da libertad para que configuréis la apariencia como queráis, siempre que el programa tenga las siguientes funcionalidades:

1. Abrir correctamente el fichero de personas que se le pasa como parámetro. Deberá leer las personas del fichero, guardarlas en un *ArrayList* o estructura similar, y mostrar un listado de nombre y primer apellido en un combo, lista fija, o control similar.
2. Cuando se seleccione un nombre de la lista, se deberán mostrar en una serie de cuadros de texto todos los valores almacenados de esa persona (nombre, apellido1, apellido2, dirección y teléfono), para que se puedan modificar. Para mostrar estos datos, podéis hacer que se muestren cuando se pulse un botón, o directamente cuando se seleccione a alguien en la lista, como mejor sepáis.
3. Podremos modificar los datos de las personas seleccionadas en los cuadros de texto, y pulsando en un botón de *Modificar*, modificar dichos datos de la persona, y que se queden así guardados en el *ArrayList*.
4. También podremos añadir personas nuevas, y quitar existentes, de la lista, mediante botones para añadir y quitar.
5. Finalmente, podremos volver a guardar en el mismo fichero que se pasó

como parámetro al principio los datos actualizados que haya en el *ArrayList* en un momento dado.

A partir de aquí, daremos algunas indicaciones de cómo hacer la aplicación, por si no sabéis por dónde empezar, o cómo hacer determinados apartados. En cualquier caso, podéis implementar la aplicación como queráis. Y si queréis, podéis añadir contenidos opcionales, como los que se proponen al final del ejercicio. Si no queréis seguir la guía para probar vosotros mismos, seguid leyendo por [aquí](#)

NOTA IMPORTANTE: se obviarán algunas cosas, como los paquetes que se necesitan para usar algunas clases. Deberás importar los paquetes que necesites para poder trabajar con todas las clases que se proponen.

Cómo dar apariencia a la aplicación

Vamos a dar unas recomendaciones para que la aplicación quede visualmente como la de la imagen superior.

- En primer lugar, comprobamos que la clase herede de **JFrame**, y debemos definir unas variables globales de la clase, que serán las que utilizemos después en diferentes eventos y funciones. Dichas variables son los cuadros de texto, y el desplegable. El resto de etiquetas y botones sólo los referenciaremos en el constructor, y no hace falta hacerlos globales.

```
public class Agenda extends JFrame
{
    JComboBox personas = new JComboBox();
    JTextField txtNombre = new JTextField();
    JTextField txtApellido1 = new JTextField();
    JTextField txtApellido2 = new JTextField();
    JTextField txtDireccion = new JTextField();
    JTextField txtTelefono = new JTextField();

    public Agenda(String fich)
    {
        ...
    }
}
```

- Ahora definiremos un tamaño para la misma, de 400 x 300, añadiremos una línea que podemos usar en *JFrame*, para hacer que se cierre la ventana cuando pulsemos el botón de cerrar (así nos ahorramos definir el evento para cerrar la ventana, es una de la ventajas de Swing):

```
public Agenda(String fich)
{
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ...
}
```

Hay que tener en cuenta que, por defecto, el *JFrame* tiene un gestor del tipo *BorderLayout*, que será el que utilizemos para colocar los componentes a continuación.

- Colocamos ahora los controles. Definiremos 3 zonas:
 - Una zona superior, que contendrá el desplegable (zona NORTE)
 - Una zona central, con todos los cuadros de texto y etiquetas (zona CENTRO)
 - Una zona inferior, con los botones (zona SUR)

Para la zona superior, definimos un subpanel, que será un *GridLayout* de una sola celda, ponemos el desplegable dentro. Podríamos directamente añadir el desplegable en la zona norte del *JFrame*, pero lo hacemos así para que el desplegable ocupe todo el espacio, si no su tamaño sería más estrecho, y no se vería como en la figura.

```
public Agenda(String fich)
{
    ...
    JPanel panelSup = new JPanel();
    panelSup.setLayout(new GridLayout(1, 1));
    panelSup.add(personas);
    ...
}
```

Para la zona central, definimos otro panel que será un *GridLayout* de 5 filas y 2 columnas, para almacenar las 5 etiquetas que hay, y sus correspondientes 5 cuadros de texto:

```
public Agenda(String fich)
{
    ...
    JPanel panelCen = new JPanel();
    panelCen.setLayout(new GridLayout(5, 2));

    panelCen.add(new JLabel("Nombre:"));
    panelCen.add(txtNombre);
    panelCen.add(new JLabel("Apellido 1:"));
    panelCen.add(txtApellido1);
    panelCen.add(new JLabel("Apellido 2:"));
    panelCen.add(txtApellido2);
    panelCen.add(new JLabel("Direccion:"));
    panelCen.add(txtDireccion);
    panelCen.add(new JLabel("Telefono:"));
    panelCen.add(txtTelefono);

    ...
}
```

Finalmente, para la zona inferior definimos otro panel, con un *GridLayout* de 2 filas y 3 columnas, donde poner todos los botones que hemos visto. Crearemos una variable botón en el constructor para cada uno, y luego más adelante ya le pondremos los eventos.

```
public Agenda(String fich)
{
    ...
    JPanel panelInf = new JPanel();
    panelInf.setLayout(new GridLayout(2, 3));

    JButton btnVer = new JButton ("Ver Datos");
    JButton btnGuardar = new JButton ("Modificar Datos");
    JButton btnAnadir = new JButton ("Añadir Persona");
    JButton btnQuitar = new JButton ("Quitar Persona");
    JButton btnGuardarFich = new JButton ("Guardar Fichero");
    JButton btnBorrarCuadros = new JButton ("Borrar Cuadros");

    panelInf.add(btnVer);
}
```

```
panelInf.add(btnGuardar);
panelInf.add(btnAnadir);
panelInf.add(btnQuitar);
panelInf.add(btnGuardarFich);
panelInf.add(btnBorrarCuadros); JPanel panelSup = new JPanel();
...
```

Finalmente, añadimos cada uno de los subpaneles a la zona que le corresponde:

```
public Agenda(String fich)
{
    ...
    getContentPane().add(panelSup, BorderLayout.NORTH);
    getContentPane().add(panelCen, BorderLayout.CENTER);
    getContentPane().add(panelInf, BorderLayout.SOUTH);
    ...
}
```

Llegados a este punto, compila y ejecuta el programa, para ver si te funciona. Pásale como parámetro cualquier nombre de fichero, porque aún no lo utiliza para nada.

La lista de personas

Veamos ahora cómo gestionar la lista de personas. Para almacenarla en memoria, vamos a hacernos una variable global (junto con los cuadros de texto y el desplegable), de tipo *ArrayList*, que nos servirá de almacén temporal. También conviene que guardemos en variable global el nombre de fichero que pasamos como parámetro, para poderlo utilizar en eventos y otros métodos.

```
public class Agenda extends JFrame
{
    ArrayList listaPersonas = new ArrayList();
    String fichero = null;
    JComboBox personas = new JComboBox();
    ...
}
```

Después, al final del constructor, una vez estén todos los controles puestos, cargamos los datos en la lista, y los volcamos en el desplegable.

Para cargar los datos en la lista, utilizamos el método **io.LeeGuardaPersona.leePersonas** que implementaste en la sesión anterior. Le pasaremos como parámetro el fichero que le pasamos como parámetro al constructor:

```
public Agenda(String fich)
{
    fichero = fich;
    ...
    listaPersonas = io.LeeGuardaPersona.leePersonas(fichero);
}
```

IMPORTANTE: Podéis utilizar para probar el fichero **ficheroPersonas.dat** que se tiene en la plantilla, con varias personas ya guardadas. Así podréis comprobar si funciona vuestro código.

A lo largo del programa, mantendremos actualizado el *ArrayList* en todo momento, con inserciones, borrados y modificaciones. Por lo tanto, el desplegable no sólo vamos a necesitar actualizarlo ahora, al inicio, sino que después en cada inserción o borrado de personas lo necesitaremos reactualizar. Para volcar en cualquier momento la información actualizada en el desplegable, conviene que nos hagamos un método al que podamos llamar siempre que lo necesitemos. En este método hacemos un bucle donde recorremos el *ArrayList*, sacamos el nombre y primer apellido de cada persona y los añadimos concatenados al desplegable:

```
private void actualizaLista()
{
    personas.removeAllItems();

    for (int i = 0; i < listaPersonas.size(); i++)
    {
        Persona p = (Persona)(listaPersonas.get(i));
        personas.addItem(p.getNombre() + " " + p.getApellido1());
    }
}
```

Al principio del método limpiamos el desplegable, para que no se acumulen resultados en cada actualización.

Ahora sólo nos queda llamar a este método tras cargar por primera vez las personas del fichero, y así tendremos el desplegable actualizado al inicio:

```
public Agenda(String fich)
{
    ...
    listaPersonas = io.LeeGuardaPersona.leePersonas(fichero);
    actualizaLista();
}
```

Aunque ahora no lo necesitemos, también nos interesará hacernos otro método que nos devuelva el índice de la lista (*ArrayList*) donde se encuentra la persona actualmente seleccionada en el desplegable, con el fin de poder sacar su objeto *Persona* en cualquier momento, o borrar ese índice, o cualquier operación. Podemos hacer algo como:

```
private int buscaPersona()
{
    String cad = (String)(personas.getSelectedItem());
    int indice = 0;
    for (int i = 0; i < listaPersonas.size(); i++)
    {
        Persona p = (Persona)(listaPersonas.get(i));
        if (cad.equals(p.getNombre() + " " + p.getApellido1()))
        {
            indice = i;
            break;
        }
    }
    return indice;
}
```

Asumimos que la persona SIEMPRE se va a encontrar. Se deja como optativo controlar qué hacer si la persona no se encuentra. Observa cómo se busca a la

persona en la lista, concatenando nombre y apellido1 y comparándolo con lo que hay en el desplegable seleccionado.

Los eventos del programa

a) Ver los datos de la persona seleccionada

Para que en los cuadros de texto centrales nos aparezcan los datos de la persona que seleccionemos en el desplegable, rellenamos el evento para el botón de *Ver Datos* (variable *btnVer* en el constructor). Vamos al final del constructor, y añadimos un evento para este botón.

Lo que tendrá que hacer será tomar el objeto *Persona* de la persona que haya seleccionada en este momento (utilizaremos el método *buscaPersona* que hemos hecho antes, para que nos dé el índice de la persona), y luego volcar sus datos en los campos de texto.

```
public Agenda(String fich)
{
    ...
    btnVer.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            // Buscar en la lista la persona que coincida con el nombre y apellidos :
            indice = buscaPersona();

            // Quedarnos con los datos de la persona encontrada
            Persona p = (Persona)(listaPersonas.get(indice));

            // Actualizar los cuadros de texto con el dato correspondiente de la per:
            txtNombre.setText(p.getNombre());
            txtApellido1.setText(p.getApellido1());
            txtApellido2.setText(p.getApellido2());
            txtDireccion.setText(p.getDireccion());
            txtTelefono.setText(p.getTelefono());
        }
    });
}
```

b) Borrar todos los cuadros de texto

Para que los cuadros de texto centrales puedan borrarse de forma cómoda (para escribir datos nuevos, o para lo que sea), tenemos el botón *Borrar Cuadros* (variable *btnBorrarCuadros* en el constructor). Añádele un evento muy sencillo: tienes que tomar cada uno de los cuadros de texto, y hacerle un *setText("")* para borrarlo. Como puede ser una operación que se realice desde otras (por ejemplo, poner los cuadros en blanco después de borrar a una persona, o de añadirla), es mejor que hagas un método aparte que ponga los cuadros en blanco, y que desde el evento llames al método:

```
private void borraCampos()
{
    ... // Rellena esto como creas conveniente
}

btnBorrarcampos.addActionListener(new ActionListener()
```

```

{
    public void actionPerformed(ActionEvent e)
    {
        borraCampos();
    }
}

```

c) Añadir una nueva persona a la lista

Cuando pulsemos el botón de *Añadir Persona*, los datos que haya escritos en los cuadros de texto se añadirán como un nuevo objeto *Persona* a la lista de personas. Así que en el evento de este botón (variable *btnAnadir* del constructor) simplemente construirá un nuevo objeto *Persona* con los datos de los cuadros de texto, y lo añadirá a la lista. Después, llamará al método *actualizaLista* que hemos hecho antes, para mantener actualizado el desplegable con la nueva inserción. Opcionalmente, podemos volver a dejar los cuadros de texto en blanco, para que quede mejor estéticamente.

```

public Agenda(String fich)
{
    ...
    btnAnadir.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            Persona p = new Persona(txtNombre.getText(), txtApellido1.getText(), txt
                               txtDireccion.getText(), txtTelefon
            listaPersonas.add(p);
            actualizaLista();
            borraCampos()
        }
    });
}

```

d) Quitar una persona de la lista

Cuando pulsemos el botón de *Quitar Persona*, se eliminará del desplegable, y también del *ArrayList* interno. En el evento de este botón (variable *btnQuitar* del constructor) buscaremos la persona en la lista (con nuestro método *buscaPersona*), eliminaremos de la lista el índice seleccionado, y actualizaremos después el desplegable con los cambios. También opcionalmente, podemos volver a dejar los cuadros de texto en blanco, para que quede mejor.

```

public Agenda(String fich)
{
    ...
    btnQuitar.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            int indice = buscaPersona();
            listaPersonas.remove(indice);
            actualizaLista();
            borraCampos();
        }
    });
}

```

e) Modificar una persona de la lista

Cuando pulsemos el botón de *Modificar Datos*, se actualizarán los datos de la persona que hay actualmente en el desplegable, con los que haya en el cuadro de texto. En el evento de este botón (variable *btnGuardar* del constructor) buscaremos la persona en la lista (con nuestro método *buscaPersona*), actualizaremos los datos del objeto *Persona* utilizando sus métodos *setXXX()*, y actualizaremos después el desplegable con los cambios. También opcionalmente, podemos volver a dejar los cuadros de texto en blanco, para que quede mejor.

```
public Agenda(String fich)
{
    ...
    btnGuardar.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            int indice = buscaPersona();
            Persona p = (Persona)(listaPersonas.get(indice));
            p.setNombre(txtNombre.getText());
            p.setApellido1(txtApellido1.getText());
            p.setApellido2(txtApellido2.getText());
            p.setDireccion(txtDireccion.getText());
            p.setTelefono(txtTelefono.getText());
            actualizaLista();
            borraCampos();
        }
    });
}
```

Observa cuando lo ejecutes cómo los datos se modifican en la lista directamente, no hace falta llamar a *listaPersonas.set(...)* en el código. Asumimos que los cuadros de texto estarán debidamente rellenos. Dejamos como opcional controlar que lo estén.

f) Guardar en fichero los datos

Cuando pulsemos el botón de *Guardar Fichero*, se deberán volcar al fichero los datos de las personas en el *ArrayList*. Para ello, utilizaremos el método **io.LeeGuardaPersona.guardaPersonas(...)** que hiciste en la sesión anterior. Le pasaremos como parámetro el nombre del fichero que tiene como parámetro el constructor, y que guardamos en la variable global *fichero*.

```
public Agenda(String fich)
{
    ...
    btnGuardarFich.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            LeeGuardaPersona.guardaPersonas(fichero, listaPersonas);
        }
    });
}
```

Observa que, si no hubiésemos guardado el fichero en la variable global, ahora no podríamos acceder a su nombre, puesto que desde este evento no es visible el parámetro del constructor.

Hasta aquí llega la parte obligatoria de la aplicación. Prueba que funciona correctamente, y comenta en el fichero de respuestas qué te ha parecido hacerla, y qué ventajas e inconvenientes le ves a los pasos indicados en esta guía, si es que los has seguido.

(OPCIONAL) Además, si queréis podéis añadir cualquier característica adicional que se os ocurra. Aquí proponemos algunas:

- Pasa a menús las opciones que ofrece el programa en forma de botones:
 - *Ver* los datos de una persona
 - *Añadir* una nueva persona
 - *Quitar* una persona
 - *Guardar* los datos modificados de la persona
 - *Guardar* los datos de todos en el mismo fichero de entrada.
- En lugar de pasar como parámetro en el *main* el fichero con las personas, utiliza la clase *JFileChooser* de Swing para abrir y guardar ficheros. Así no necesitarás pasar nada como parámetro, incluso podrás crear ficheros nuevos de personas desde el programa. Mira el API para ver cómo utilizarla. Te serán de utilidad sus métodos *showOpenDialog* y *showSaveDialog*.
- Incluye una especie de "barra de herramientas", con botones para abrir y guardar ficheros, pero que en lugar de texto tengan los típicos iconos de abrir y guardar, y que al pulsarlos se llame a las opciones de *JFileChooser* del punto anterior. Puedes encontrar los iconos de abrir y guardar en casi cualquier aplicación de Windows, por ejemplo el Word o el WordPad. Captura un pantallazo del programa, recorta los iconos y guárdalos como imágenes para poderlos utilizar. Después, cárgalos en los botones utilizando objetos *ImageIcon*.
- Ordenar los nombres en la lista, según el criterio de ordenación que hicisteis para la clase *Persona* en la sesión 7.
- Controlar todo tipo de errores: al leer el fichero de entrada, al escribir mal los datos en los cuadros de texto, etc. Para sacar mensajes de error en una ventana, os interesará utilizar la clase **JOptionPane** de Swing, y sus métodos **showMessageDialog** y similares.

Comenta en el fichero de respuestas las características opcionales que hayas implementado.

PARA ENTREGAR

Guarda en la carpeta **modulo3** de tu CVS los siguientes elementos para esta sesión:

- Todos los ficheros fuente (**sesion09.Calculadora** y **sesion09.JCalculadora**, y también **gui.Agenda** con la aplicación final hecha). Los otros paquetes y clases necesarias para esta aplicación (**datos.Persona** e **io.LeeGuardaPersona**), ya los tendrás añadidos de sesiones anteriores, si no los has modificado en esta.
- El fichero de entrada **ficheroPersonas.dat** que se ha proporcionado.
- Fichero de texto **respuestas.txt** de esta sesión contestando a todas las preguntas formuladas.