Lenguaje Java Avanzado

Índice

1 Introducción al lenguaje Java.... 4
1.1 Java.... 4
1.2 Conceptos previos de POO.... 5
1.3 Componentes de un programa Java.... 7
1.4 Herencia e interfaces.... 16
1.5 Hilos.... 19
1.6 Clases útiles.... 23
1.7 Estructuras de datos.... 29
2 Ejercicios de Introducción al lenguaje Java.... 31
2.1 Uso de interfaces (1 punto).... 31
2.2 Refactorización (1 punto).... 32
2.3 Documentación (0.5 puntos).... 32
2.4 Centro cultural (1 punto).... 32
2.5 Copia de propiedades con BeanUtils (0.5 puntos).... 33
3 Colecciones de datos.... 34
3.1 Colecciones.... 34
3.2 Comparación de objetos.... 45
3.3 Polimorfismo e interfaces.... 47
3.4 Tipos de datos básicos en las colecciones.... 48
4 Ejercicios de colecciones.... 50
4.1 Implementaciones e interfaces (1 punto).... 50
4.2 Uso de listas (1 punto).... 50
5 Tratamiento de errores.... 52
5.1 Introducción.... 52
5.2 Errores en tiempo de ejecución: Excepciones.... 52
5.3 Errores en tiempo de compilación.... 57

6 Ejercicios de tratamiento de errores..... 64
6.1 Captura de excepciones (0.5 puntos)..... 64
6.2 Lanzamiento de excepciones (0.5 puntos)..... 64
6.3 Excepciones como tipos genéricos en la aplicación filmotecas(0.5 puntos)..... 66
6.4 Excepciones anidadas en la aplicación filmotecas (1.5 puntos)..... 66
7 Casos de prueba: JUnit..... 68
7.1 Introducción a JUnit..... 68
7.2 Integración de JUnit en Eclipse..... 68
7.3 Un ejemplo sencillo..... 70
7.4 Fixtures..... 79
7.5 Objetos mock..... 81
7.6 Suites de pruebas..... 87
8 Ejercicios de JUnit..... 89
8.1 Pruebas del gestor de filmotecas (2 puntos)..... 89
8.2 Desarrollo guiado por pruebas (1 punto)..... 90
9 Serialización de datos..... 92
9.1 Introducción..... 92
9.2 Flujos de datos de entrada/salida..... 92
9.3 Entrada, salida y salida de error estándar..... 93
9.4 Acceso a ficheros..... 94
9.5 Acceso a los recursos..... 95
9.6 Acceso a la red..... 96
9.7 Codificación de datos..... 96
9.8 Serialización de objetos..... 97
10 Ejercicios de Serialización..... 99
10.1 Leer un fichero de texto (0.5 puntos)..... 99
10.2 Lectura de una URL (0.5 puntos)..... 99
10.3 Gestión de productos (1 punto)..... 99
10.4 Guardar datos de la filmoteca (1 punto)..... 100
11 Depuración y gestión de logs..... 102
11.1 Depuración con Eclipse..... 102
11.2 Gestión de logs con Log4Java..... 107

11.3 La librería commons-logging....114
12 Ejercicios de depuración y logging....118
12.1 Depuración de código (1.5 puntos)....118
12.2 Logs al leer ficheros (1.5 puntos)....118
13 Java Database Connectivity....120
13.1 Introducción a JDBC....120
13.2 Consulta a una base de datos con JDBC....124
13.3 Restricciones y movimientos en el ResultSet....127
13.4 Sentencias de actualización....129
13.5 Otras llamadas a la BD....130
13.6 Optimización de sentencias....131
13.7 Transacciones....133
14 Ejercicios de Java Database Connectivity....136
14.1 Filmoteca en MySQL....136
14.2 Conectar con la base de de datos (0.5 puntos)....137
14.3 Consulta sin parámetros (0.5 puntos)....137
14.4 Sentencias preparadas (0.5 puntos)....138
14.5 Sentencias de borrado de registros (0.5 puntos)....138
14.6 Sentencias de inserción de registros y valores autoincrementados (0.5 puntos)138
14.7 Transacciones (0.5 puntos)....139
15 Pruebas con DBUnit....140
15.1 Introducción....140
15.2 DbUnit....140
15.3 Prácticas recomendadas....141
15.4 Clases e interfaces....141
15.5 Estructura en un proyecto....142
16 Ejercicios de DbUnit....146
16.1 Generar el XML de DbUnit (0.5 puntos)....146
16.2 Clase para los test de DbUnit (1 punto)....146
16.3 Test de borrado y añadido (0.5 puntos)....146
16.4 Test que espera una excepción (0.5 puntos)....147
16.5 Test de borrado (0.5 puntos)....147

1. Introducción al lenguaje Java

1.1. Java

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

El código Java, una vez compilado, puede llevarse sin modificación alguna sobre cualquier máquina, y ejecutarlo. Esto se debe a que el código se ejecuta sobre una máquina hipotética o virtual, la **Java Virtual Machine**, que se encarga de interpretar el código (ficheros compilados `.class`) y convertirlo a código particular de la CPU que se esté utilizando (siempre que se soporte dicha máquina virtual).

1.1.1. Certificación Sun / Oracle

Aunque el resto del curso de Experto está orientado a certificaciones en el ámbito de JEE, este módulo de Java y Herramientas de Desarrollo se basa en las certificaciones para Java básico o estándar. Dichas certificaciones son dos:

- *Oracle Certified Professional Java Programmer* (antes SCJP - *Sun Certified Java Programmer*): http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=320
- *Oracle Certified Master, Java SE 6 Developer* (antes SCJD - *Sun Certified Java Developper*): http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=321

Para la primera (**SCJP**), se debe completar un examen. En el caso del certificado para la plataforma Java 1.5, el examen abarca los siguientes apartados generales:

- **Sección 1: Declaraciones, inicializaciones y ámbitos:** evalúa si el alumno es capaz de escribir código que declare clases o interfaces, utilice adecuadamente la estructura de paquetes e imports, utilice código con tipos simples, arrays, objetos estáticos, variables locales, constructores, métodos estáticos y no estáticos, sobrecarga de métodos, etc
- **Sección 2: Control de flujo:** uso de sentencias `if`, `switch`, bucles (`for, do, while, break, continue`), manejo y uso de excepciones (`try-catch-finally`), etc
- **Sección 3: Contenidos del API:** uso de wrappers básicos (`Integer, Boolean`, etc), entrada/salida de ficheros, serialización de objetos para E/S, formateo de datos con el paquete `java.text`, y parseo de cadenas mediante expresiones regulares y similares (paquetes `java.util` y `java.util.regex`)
- **Sección 4: Concurrencia:** manejo de hilos (mediante `Thread` y mediante `Runnable`),

estados de los hilos, interbloqueos y sincronización
- **Sección 5: Conceptos sobre orientación a objetos:** desarrollo de código que cumpla los requerimientos de encapsulamiento, cohesión y acoplamiento entre clases (mucha cohesión, poco acoplamiento). Uso del polimorfismo y del casting. Uso de métodos sobrecargados, llamadas a la superclase, etc.
- **Sección 6: Colecciones:** determinar qué tipos de colecciones (listas, hashmaps, etc) utilizar en diferentes supuestos. Comparaciones y ordenaciones entre objetos de una colección, etc
- **Sección 7: Fundamentos:** uso correcto de los modificadores de acceso, declaraciones de paquetes, imports. Seguimiento de trazas. Manejo de referencias a objetos. Uso del recolector de basura... etc

Para la segunda (**SCJD**) es necesario haber obtenido antes la primera certificación (SCJP). Después, se deben superar dos pruebas: un supuesto de programación, y un examen.

La primera prueba (el supuesto de programación), consiste en escribir código para implementar una supuesta aplicación para empresa. Se evaluarán aspectos como documentación, diseño orientado a objetos, desarrollo de la interfaz gráfica, interbloqueos, etc.

La segunda prueba (el examen), es una explicación sobre el desarrollo que hayamos hecho en el supuesto de programación anterior, explicando las decisiones principales que hemos tenido que tomar, ventajas y desventajas de las mismas, y justificación de dichas decisiones, en función de los objetivos propuestos para la implementación.

Más información sobre las certificaciones en los enlaces respectivos vistos antes.

1.1.2. Recursos adicionales

1.1.2.1. Bibliografía

- **Curso de Java**, Ian F. Darwin, *Ed. Anaya Multimedia, Colección O'Reilly*
- **Java 2 v5.0**, Varios autores, *Ed. Anaya Multimedia, Colección Wrox*
- **Piensa en Java**, Bruce Eckel, *Ed. Prentice Hall*
- **Core Java 2**, Cay Horstmann y Gary Cornell, *Ed. Prentice Hall PTR*
- **Java in a Nutshell**, David Flanagan, *Ed. O'Reilly Media*

1.1.2.2. Enlaces

- Web oficial de Java, http://www.oracle.com/technetwork/java/index.html

1.2. Conceptos previos de POO

Java es un lenguaje orientado a objetos (OO), por lo que, antes de empezara ver qué

elementos componen los programas Java, conviene tener claros algunos conceptos de la programación orientada a objetos (POO).

1.2.1. Concepto de clase y objeto

El elemento fundamental a la hora de hablar de programación orientada a objetos es el concepto de objeto en sí, así como el concepto abstracto de clase. Un **objeto** es un conjunto de variables junto con los métodos relacionados con éstas. Contiene la información (las variables) y la forma de manipular la información (los métodos). Una **clase** es el prototipo que define las variables y métodos que va a emplear un determinado tipo de objeto, es la definición abstracta de lo que luego supone un objeto en memoria.

Poniendo un símil fuera del mundo de la informática, la clase podría ser el concepto de *coche*, donde nos vienen a la memoria los parámetros que definen un coche (dimensiones, cilindrada, maletero, etc), y las operaciones que podemos hacer con un coche (acelerar, frenar, adelantar, estacionar). La idea abstracta de coche que tenemos es lo que equivaldría a la clase, y la representación concreta de coches concretos (por ejemplo, Peugeot 307, Renault Megane, Volkswagen Polo...) serían los objetos de tipo coche.

1.2.2. Concepto de campo, método y constructor

Toda clase u objeto se compone internamente de constructores, campos y/o métodos. Veamos qué representa cada uno de estos conceptos: un **campo** es un elemento que contiene información relativa a la clase, y un **método** es un elemento que permite manipular la información de los campos. Por otra parte, un **constructor** es un elemento que permite reservar memoria para almacenar los campos y métodos de la clase, a la hora de crear un objeto de la misma.

1.2.3. Concepto de 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.

Por ejemplo, podríamos tener una clase genérica *Animal*, y heredamos de ella para formar clases más específicas, como *Pato*, *Elefante*, o *León*. Estas clases tendrían todo lo de la clase padre *Animal*, y además cada una podría tener sus propios elementos adicionales.

Una característica derivada de la herencia es que, por ejemplo, si tenemos un método `dibuja(Animal a)`, que se encarga de hacer un dibujo del animal que se le pasa como parámetro, podremos pasarle a este método como parámetro tanto un *Animal* como un *Pato*, *Elefante*, o cualquier otro subtipo directo o indirecto de *Animal*. Esto se conoce como **polimorfismo**.

1.2.4. Modificadores de acceso

Tanto las clases como sus elementos (constructores, campos y métodos) pueden verse modificados por lo que se suelen llamar modificadores de acceso, que indican hasta dónde es accesible el elemento que modifican. Tenemos tres tipos de modificadores:

- **privado:** el elemento es accesible únicamente dentro de la clase en la que se encuentra.
- **protegido:** el elemento es accesible desde la clase en la que se encuentra, y además desde las subclases que hereden de dicha clase.
- **público:** el elemento es accesible desde cualquier clase.

1.2.5. 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 interfaz implementen ese esqueleto para dicho subtipo concreto. Por ejemplo, volviendo con el ejemplo anterior, podemos definir en la clase *Animal* el método *dibuja()* y el método *imprime()*, y que *Animal* sea una clase abstracta o un interfaz.

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

- En una **clase**, al definir *Animal* tendríamos que implementar el código de los métodos *dibuja()* e *imprime()*. Las subclases que hereden de *Animal* no tendrían por qué implementar los métodos, a no ser que quieran redefinirlos para 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 hijas podrían implementar otros interfaces.

1.3. Componentes de un programa Java

En un programa Java podemos distinguir varios elementos:

1.3.1. Clases

Para definir una clase se utiliza la palabra reservada `class`, seguida del nombre de la clase:

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

Es recomendable que los nombres de las clases sean sustantivos (ya que suelen representar entidades), pudiendo estar formados por varias palabras. La primera letra de cada palabra estará en mayúscula y el resto de letras en minúscula. Por ejemplo, `DatosUsuario`, `Cliente`, `GestorMensajes`.

Cuando se trate de una clase encargada únicamente de agrupar un conjunto de recursos o de constantes, su nombre se escribirá en plural. Por ejemplo, `Recursos`, `MensajesError`.

1.3.2. Campos y variables

Dentro de una clase, o de un método, podemos definir campos o variables, respectivamente, que pueden ser de tipos simples, o clases complejas, bien de la API de Java, bien que hayamos definido nosotros mismos, o bien que hayamos copiado de otro lugar.

Al igual que los nombres de las clases, suele ser conveniente utilizar sustantivos que describan el significado del campo, pudiendo estar formados también por varias palabras. En este caso, la primera palabra comenzará por minúscula, y el resto por mayúscula. Por ejemplo, `apellidos`, `fechaNacimiento`, `numIteraciones`.

De forma excepcional, cuando se trate de variables auxiliares de corto alcance se puede poner como nombre las iniciales del tipo de datos correspondiente:

```
int i;
Vector v;
MiOtraClase moc;
```

Por otro lado, las constantes se declaran como `final static`, y sus nombres es escribirán totalmente en mayúsculas, separando las distintas palabras que los formen por caracteres de subrayado (`'_'`). Por ejemplo, `ANCHO_VENTANA`, `MSG_ERROR_FICHERO`.

1.3.3. Métodos

Los métodos o funciones se definen de forma similar a como se hacen en C: indicando el tipo de datos que devuelven, el nombre del método, y luego los argumentos entre paréntesis:

```
void imprimir(String mensaje)
{
        ... // Código del método
}

double sumar(double... numeros){
            //Número variable de argumentos
            //Se accede a ellos como a un vector:
            //numeros[0], numeros[1], ...
}

Vector insertarVector(Object elemento, int posicion)
{
        ... // Código del método
}
```

Al igual que los campos, se escriben con la primera palabra en minúsculas y el resto comenzando por mayúsculas. En este caso normalmente utilizaremos verbos.

Nota

Una vez hayamos creado cualquier clase, campo o método, podremos modificarlo pulsando con el botón derecho sobre él en el explorador de Eclipse y seleccionando la opción *Refactor > Rename...* del menú emergente. Al cambiar el nombre de cualquiera de estos elementos, Eclipse actualizará automáticamente todas las referencias que hubiese en otros lugares del código. Además de esta opción para renombrar, el menú *Refactor* contiene bastantes más opciones que nos permitirán reorganizar automáticamente el código de la aplicación de diferentes formas.

1.3.4. Constructores

Podemos interpretar los constructores como métodos que se llaman igual que la clase, y que se ejecutan con el operador `new` para reservar memoria para los objetos que se creen de dicha clase:

```
MiClase()
{
        ... // Código del constructor
}

MiClase(int valorA, Vector valorV)
{
        ... // Código de otro constructor
}
```

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

Si estamos utilizando una clase que hereda de otra, y dentro del constructor de la subclase queremos llamar a un determinado constructor de la superclase, utilizaremos `super`. Si no se hace la llamada a `super`, por defecto la superclase se construirá con su constructor vacío. Si esta superclase no tuviese definido ningún constructor vacío, o bien quisiesemos utilizar otro constructor, podremos llamar a `super` proporcionando los parámetros correspondientes al constructor al que queramos llamar. Por ejemplo, si heredamos de `MiClase` y desde la subclase queremos utilizar el segundo constructor de la superclase, al comienzo del constructor haremos la siguiente llamada a `super`:

```
SubMiClase()
{
    super(0, new Vector());
    ... // Código de constructor subclase
}
```

Nota

Podemos generar el constructor de una clase automáticamente con Eclipse, pulsando con el botón derecho sobre el código y seleccionando *Source > Generate Constructor Using Fields...* o *Source > Generate Constructors From Superclass...*

1.3.5. Paquetes

Las clases en Java se organizan (o pueden organizarse) en paquetes, de forma que cada paquete contenga un conjunto de clases. También puede haber subpaquetes especializados dentro de un paquete o subpaquete, formando así una jerarquía de paquetes, que después se plasma en el disco duro en una estructura de directorios y subdirectorios igual a la de paquetes y subpaquetes (cada clase irá en el directorio/subdirectorio correspondiente a su paquete/subpaquete).

Cuando queremos indicar que una clase pertenece a un determinado paquete o subpaquete, se coloca al principio del fichero la palabra reservada `package` seguida por los paquetes/subpaquetes, separados por '.' :

```
package paq1.subpaq1;
...
class MiClase {
...
```

Si queremos desde otra clase utilizar una clase de un paquete o subpaquete determinado (diferente al de la clase en la que estamos), incluimos una sentencia `import` antes de la clase (y después de la línea `package` que pueda tener la clase, si la tiene), indicando qué paquete o subpaquete queremos importar:

```
import paq1.subpaq1.*;
```

```
import paq1.subpaq1.MiClase;
```

La primera opción (*) se utiliza para importar todas las clases del paquete (se utiliza cuando queremos utilizar muchas clases del paquete, para no ir importando una a una). La segunda opción se utiliza para importar una clase en concreto.

Nota

Es recomendable indicar siempre las clases concretas que se están importando y no utilizar el `*`. De esta forma quedará más claro cuales son las clases que se utilizan realmente en nuestro código. Hay diferentes paquetes que contienen clases con el mismo nombre, y si se importasen usando `*` podríamos tener un problema de ambigüedad.

Al importar, ya podemos utilizar el nombre de la clase importada directamente en la clase que estamos construyendo. Si no colocásemos el `import` podríamos utilizar la clase igual, pero al referenciar su nombre tendríamos que ponerlo completo, con paquetes y subpaquetes:

```
MiClase mc;              // Si hemos hecho el 'import' antes
```

```
paq1.subpaq1.MiClase mc; // Si NO hemos hecho el 'import' antes
```

Existe un paquete en la API de Java, llamado `java.lang`, que no es necesario importar. Todas las clases que contiene dicho paquete son directamente utilizables. Para el resto de paquetes (bien sean de la API o nuestros propios), será necesario importarlos cuando

estemos creando una clase fuera de dichos paquetes.

Los paquetes normalmente se escribirán totalmente en minúsculas. Es recomendable utilizar nombres de paquetes similares a la URL de nuestra organización pero a la inversa, es decir, de más general a más concreto. Por ejemplo, si nuestra URL es `http://www.jtech.ua.es` los paquetes de nuestra aplicación podrían recibir nombres como `es.ua.jtech.proyecto.interfaz`, `es.ua.jtech.proyecto.datos`, etc.

Importante

Nunca se debe crear una clase sin asignarle nombre de paquete. En este caso la clase se encontraría en el paquete `sin nombre`, y no podría ser referenciada por las clases del resto de paquetes de la aplicación.

Con Eclipse podemos importar de forma automática los paquetes necesarios. Para ello podemos pulsar sobre el código con el botón derecho y seleccionar *Source* > *Organize imports*. Esto añadirá y ordenará todos los `imports` necesarios. Sin embargo, esto no funcionará si el código tiene errores de sintaxis. En ese caso si que podríamos añadir un `import` individual, situando el cursor sobre el nombre que se quiera importar, pulsando con el botón derecho, y seleccionando *Source* > *Add import*.

1.3.6. Tipo enumerado

El tipo `enum` permite definir un conjunto de posibles valores o estados, que luego podremos utilizar donde queramos:

Ejemplo

```
// Define una lista de 3 valores y luego comprueba en un switch
// cuál es el valor que tiene un objeto de ese tipo
enum EstadoCivil {soltero, casado, divorciado};
EstadoCivil ec = EstadoCivil.casado;
ec = EstadoCivil.soltero;
switch(ec)
{
        case soltero: System.out.println("Es soltero");
                      break;
        case casado: System.out.println("Es casado");
                      break;
        case divorciado:System.out.println("Es divorciado");
                      break;
}
```

Los elementos de una enumeración se comportan como objetos Java. Por lo tanto, la forma de nombrar las enumeraciones será similar a la de las clases (cada palabra empezando por mayúscula, y el resto de clases en minúscula).

Como objetos Java que son, estos elementos pueden tener definidos campos, métodos e incluso constructores. Imaginemos por ejemplo que de cada tipo de estado civil nos interesase conocer la retención que se les aplica en el sueldo. Podríamos introducir esta información de la siguiente forma:

```
enum EstadoCivil {soltero(0.14f), casado(0.18f), divorciado(0.14f);
        private float retencion;

        EstadoCivil(float retencion) {
                this.retencion = retencion;
        }

        public float getRetencion() {
                return retencion;
        }
};
```

De esta forma podríamos calcular de forma sencilla la retención que se le aplica a una persona dado su salario y su estado civil de la siguiente forma:

```
public float calculaRetencion(EstadoCivil ec, float salario) {
    return salario * ec.getRetencion();
}
```

Dado que los elementos de la enumeración son objetos, podríamos crear nuevos métodos o bien sobrescribir métodos de la clase `Object`. Por ejemplo, podríamos redefinir el método `toString` para especificar la forma en la que se imprime cada elemento de la enumeración (por defecto imprime una cadena con el nombre del elemento, por ejemplo `"soltero"`).

1.3.7. Modificadores de acceso

Tanto las clases como los campos y métodos admiten modificadores de acceso, para indicar si dichos elementos tienen ámbito *público, protegido* o *privado*. Dichos modificadores se marcan con las palabras reservadas `public`, `protected` y `private`, respectivamente, y se colocan al principio de la declaración:

```
public class MiClase {
...
protected int b;
...
private int miMetodo(int b) {
...
```

El modificador `protected` implica que los elementos que lo llevan son visibles desde la clase, sus subclases, y las demás clases del mismo paquete que la clase.

Si no se especifica ningún modificador, el elemento será considerado de tipo *paquete*. Este tipo de elementos podrán ser visibles desde la clase o desde clases del mismo paquete, pero no desde las subclases.

Cada fichero Java que creemos debe tener una y sólo una **clase pública** (que será la clase principal del fichero). Dicha clase debe llamarse igual que el fichero. Aparte, el fichero podrá tener otras clases internas, pero ya no podrán ser públicas.

Por ejemplo, si tenemos un fichero `MiClase.java`, podría tener esta apariencia:

```
public class MiClase
{
```

```
        ...
}

class OtraClase
{
        ...
}

class UnaClaseMas
{
        ...
}
```

Si queremos tener acceso a estas clases internas desde otras clases, deberemos declararlas como estáticas. Por ejemplo, si queremos definir una etiqueta para incluir en los puntos 2D definidos en el ejemplo anterior, podemos definir esta etiqueta como clase interna (para dejar claro de esta forma que dicha etiqueta es para utilizarse en `Punto2D`). Para poder manipular esta clase interna desde fuera deberemos declararla como estática de la siguiente forma:

```
public class Punto2D {
    ...
        static class Etiqueta {
                String texto;
                int tam;
                Color color;
        }
}
```

Podremos hacer referencia a ella desde fuera de `Punto2D` de la siguiente forma:

```
Punto2D.Etiqueta etiq = new Punto2D.Etiqueta();
```

1.3.8. Otros modificadores

Además de los modificadores de acceso vistos antes, en clases, métodos y/o campos se pueden utilizar también estos modificadores:

- `abstract`: elemento base para la herencia (los objetos subtipo deberán definir este elemento). Se utiliza para definir clases abstractas, y métodos abstractos dentro de dichas clases, para que los implementen las subclases que hereden de ella.
- `static`: elemento compartido por todos los objetos de la misma clase. Con este modificador, no se crea una copia del elemento en cada objeto que se cree de la clase, sino que todos comparten una sola copia en memoria del elemento, que se crea sin necesidad de crear un objeto de la clase que lo contiene. Como se ha visto anteriormente, también puede ser aplicado sobre clases, con un significado diferente en este caso.
- `final`: objeto final, no modificable (se utiliza para definir constantes) ni heredable (en caso de aplicarlo a clases).
- `synchronized`: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución.

Estos modificadores se colocan tras los modificadores de acceso:

```
// Clase abstracta para heredar de ella
public abstract class Ejemplo
{
  // Constante estática de valor 10
  public static final TAM = 10;

  // Método abstracto a implementar
  public abstract void metodo();

  public synchronized void otroMetodo()
  {
    ... // Aquí dentro sólo puede haber un hilo a la vez
  }
}
```

Nota

Si tenemos un método estático (`static`), dentro de él sólo podremos utilizar elementos estáticos (campos o métodos estáticos), o bien campos y métodos de objetos que hayamos creado dentro del método.

Por ejemplo, si tenemos:

```
public class UnaClase
{
  public int a;

  public static int metodo()
  {
    return a + 1;
  }
}
```

Dará error, porque el campo a no es estático, y lo estamos utilizando dentro del método estático. Para solucionarlo tenemos dos posibilidades: definir a como estático (si el diseño del programa lo permite), o bien crear un objeto de tipo `UnaClase` en el método, y utilizar su campo a (que ya no hará falta que sea estático, porque hemos creado un objeto y ya podemos acceder a su campo a):

```
public class UnaClase
{
  public int a;

  public static int metodo()
  {
    UnaClase uc = new UnaClase();
    // ... Aquí haríamos que uc.a tuviese el valor adecuado
    return uc.a + 1;
  }
}
```

Nota

Para hacer referencia a un elemento estático utilizaremos siempre el nombre de la clase a la que pertenece, y no la referencia a una instancia concreta de dicha clase.

Por ejemplo, si hacemos lo siguiente:

```
UnaClase uc = new UnaClase();
uc.metodo();
```

Aparecerá un *warning*, debido a que el método `metodo` no es propio de la instancia concreta `uc`, sino da la clase `UnaClase` en general. Por lo tanto, deberemos llamarlo con:

```
UnaClase.metodo();
```

1.3.9. Imports estáticos

Los *imports* estáticos permiten importar los elementos estáticos de una clase, de forma que para referenciarlos no tengamos que poner siempre como prefijo el nombre de la clase. Por ejemplo, podemos utilizar las constantes de color de la clase `java.awt.Color`, o bien los métodos matemáticos de la case `Math`.

Ejemplo

```
import static java.awt.Color;
import static java.lang.Math;

public class...
{
...
  JLabel lbl = new JLabel();
  lbl.setBackground(white);      // Antes sería Color.white
...
  double raiz = sqrt(1252.2);  // Antes sería Math.sqrt(...)
}
```

1.3.10. Argumentos variables

Java permite pasar un número variable de argumentos a una función (como sucede con funciones como `printf` en C). Esto se consigue mediante la expresión "..." a partir del momento en que queramos tener un número variable de argumentos.

Ejemplo

```
// Funcion que tiene un parámetro String obligatorio
// y n parámetros int opcionales

public void miFunc(String param, int... args)
{
...
        // Una forma de procesar n parametros variables
        for (int argumento: args)
        {
                ...
        }
...
}

...
miFunc("Hola", 1, 20, 30, 2);
miFunc("Adios");
```

1.3.11. Metainformación o anotaciones

Se tiene la posibilidad de añadir ciertas **anotaciones** en campos, métodos, clases y otros elementos, que permitan a las herramientas de desarrollo o de despliegue leerlas y realizar ciertas tareas. Por ejemplo, generar ficheros fuentes, ficheros XML, o un *Stub* de métodos para utilizar remotamente con RMI.

Un ejemplo más claro lo tenemos en las anotaciones que ya se utilizan para la herramienta Javadoc. Las marcas `@deprecated` no afectan al comportamiento de los métodos que las llevan, pero previenen al compilador para que muestre una advertencia indicando que el método que se utiliza está desaconsejado. También se tienen otras marcas `@param`, `@return`, `@see`, etc, que utiliza Javadoc para generar las páginas de documentación y las relaciones entre ellas.

1.3.12. Ejecución de clases: método main

En las clases principales de una aplicación (las clases que queramos ejecutar) debe haber un método `main` con la siguiente estructura:

```
public static void main(String[] args)
{
        ... // Código del método
}
```

Dentro pondremos el código que queramos ejecutar desde esa clase. Hay que tener en cuenta que `main` es estático, con lo que dentro sólo podremos utilizar campos y métodos estáticos, o bien campos y métodos de objetos que creemos dentro del `main`.

1.4. Herencia e interfaces

1.4.1. Herencia

Cuando queremos que una clase herede de otra, se utiliza al declararla la palabra `extends` tras el nombre de la clase, para decir de qué clase se hereda. Para hacer que `Pato` herede de `Animal`:

```
class Pato extends Animal
```

Con esto automáticamente `Pato` tomaría todo lo que tuviese `Animal` (aparte, `Pato` puede añadir sus características propias). Si `Animal` fuese una clase abstracta, `Pato` debería implementar los métodos abstractos que tuviese.

1.4.2. Punteros this y super

El puntero `this` apunta al objeto en el que nos encontramos. Se utiliza normalmente cuando hay variables locales con el mismo nombre que variables de instancia de nuestro

objeto:

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

También se suele utilizar para remarcar que se está accediendo a variables de instancia.

El puntero `super` se usa para acceder a un elemento en la clase padre. Si la clase `Usuario` tiene un método `getPermisos`, y una subclase `UsuarioAdministrador` sobrescribe dicho método, podríamos llamar al método de la super-clase con:

```
public class UsuarioAdministrador extends Usuario {
  public List<String> getPermisos() {
    List<String> permisos = super.getPermisos();
    permisos.add(PERMISO_ADMINISTRADOR);
    return permisos;
  }
}
```

También podemos utilizar `this` y `super` como primera instrucción dentro de un constructor para invocar a otros constructores. Dado que toda clase en Java hereda de otra clase, siempre será necesario llamar a alguno de los constructores de la super-clase para que se construya la parte relativa a ella. Por lo tanto, si al comienzo del constructor no se especifica ninguna llamada a `this` o `super`, se considera que hay una llamada implícita al constructor sin parámetros de la super-clase (`super()`). Es decir, los dos constructores siguientes son equivalentes:

```
public Punto2D(int x, int y, String etiq) {
    // Existe una llamada implicita a super()

    this.x = x;
    this.y = y;
    this.etiq = etiq;
}

public Punto2D(int x, int y, String etiq) {
    super();

    this.x = x;
    this.y = y;
    this.etiq = etiq;
}
```

Pero es posible que la super-clase no disponga de un constructor sin parámetros. En ese caso, si no hacemos una llamada explícita a `super` nos dará un error de compilación, ya que estará intentando llamar a un constructor inexistente de forma implícita. Es posible también, que aunque el constructor sin parámetros exista, nos interese llamar a otro constructor a la hora de construir la parte relativa a la super-clase. Imaginemos por ejemplo que la clase `Punto2D` anterior deriva de una clase `PrimitivaGeometrica` que almacena, como información común de todas las primitivas, una etiqueta de texto, y

ofrece un constructor que toma como parámetro dicha etiqueta. Podríamos utilizar dicho constructor desde la subclase de la siguiente forma:

```
public Punto2D(int x, int y, String etiq) {
    super(etiq);

    this.x = x;
    this.y = y;
}
```

También puede ocurrir que en lugar de querer llamar directamente al constructor de la super-clase nos interese basar nuestro constructor en otro de los constructores de nuestra misma clase. En tal caso llamaremos a `this` al comienzo de nuestro constructor, pasándole los parámetros correspondientes al constructor en el que queremos basarnos. Por ejemplo, podríamos definir un constructor sin parámetros de nuestra clase punto, que se base en el constructor anterior (más específico) para crear un punto con una serie de datos por defecto:

```
public Punto2D() {
    this(DEFAULT_X, DEFAULT_Y, DEFAULT_ETIQ);
}
```

Es importante recalcar que las llamadas a `this` o `super` deben ser siempre la primera instrucción del constructor.

1.4.3. Interfaces y clases abstractas

Ya hemos visto cómo definir clases normales, y clases abstractas. Si queremos definir un interfaz, se utiliza la palabra reservada `interface`, en lugar de `class`, y dentro declaramos (no implementamos), los métodos que queremos que tenga la interfaz:

```
public interface MiInterfaz
{
        public void metodoInterfaz();
        public float otroMetodoInterfaz();
}
```

Después, para que una clase implemente los métodos de esta interfaz, se utiliza la palabra reservada `implements` tras el nombre de la clase:

```
public class UnaClase implements MiInterfaz
{
        public void metodoInterfaz()
        {
                ... // Código del método
        }

        public float otroMetodoInterfaz()
        {
                ... // Código del método
        }
}
```

Notar que si en lugar de poner `implements` ponemos `extends`, en ese caso `UnaClase` debería ser un `interfaz`, que heredaría del interfaz `MiInterfaz` para definir más

métodos, pero no para implementar los que tiene la interfaz. Esto se utilizaría para definir interfaces partiendo de un interfaz base, para añadir más métodos a implementar.

Una clase puede heredar sólo de otra única clase, pero puede implementar cuantos interfaces necesite:

```
public class UnaClase extends MiClase
           implements MiInterfaz, MiInterfaz2, MiInterfaz3
{
        ...
}
```

Cuando una clase implementa una interfaz se está asegurando que dicha clase va a ofrecer los métodos definidos en la interfaz, es decir, que la clase al menos nos ofrece esa interfaz para acceder a ella. Cuando heredamos de una clase abstracta, heredamos todos los campos y el comportamiento de la superclase, y además deberemos definir algunos métodos que no habían sido implementados en la superclase.

Desde el punto de vista del diseño, podemos ver la herencia como una relación *ES*, mientras que la implementación de una interfaz sería una relación *ACTÚA COMO*.

1.5. Hilos

En este curso no vamos a profundizar en la programación de hilos. Si bien las aplicaciones en servidor funcionan sobre múltiples hilos, de su gestión ya se ocupan los servidores de aplicaciones, como se verá. Independientemente de esto, es positivo conocer el mecanismo de hilos y sobre todo, su sincronización.

Un hilo es un flujo de control dentro de un programa. 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.

1.5.1. Creación de hilos

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

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

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

método `run()`.

Para crear nuestro hilo mediante herencia haremos lo siguiente:

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

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

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

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

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

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

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

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

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

1.5.2. Ciclo de vida y prioridades

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

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

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

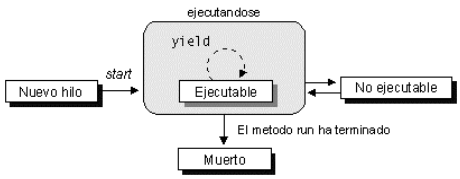
```
t.start();
```

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

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

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

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



Ciclo de vida de los hilos

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

Prioridades de los hilos

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

Ejecutable, o cuando el tiempo que se le haya asignado expire.

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

1.5.3. Sincronización de hilos

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

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

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

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

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

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

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

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

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

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la

sección crítica del objeto y desbloquearlo.

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

1.6. Clases útiles

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).

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

1.6.1. Object

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

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

```
new MiClase()
```

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

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

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

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

Podremos hacer referencia a ella de diferentes formas:

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

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

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

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

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

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

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

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

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

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

El método `clone` es un método de la clase `Object` que estará disponible para cualquier objeto Java, y nos devuelve un `Object` genérico, ya que al ser un método que puede servir para cualquier objeto nos debe devolver la copia de este tipo. De él tendremos que hacer una conversión cast a la clase de la que se trate como hemos visto en el ejemplo. Al hacer una copia con `clone` se copiarán los valores de todas las variables de instancia, pero si estas variables son referencias a objetos sólo se copiará la referencia, no el objeto. Es

decir, no se hará una copia en profundidad. Si queremos hacer una copia en profundidad deberemos sobrescribir el método `clone` para hacer una copia de cada uno de estos objetos. Para copiar objetos también podríamos definir un constructor de copia, al que se le pase como parámetro el objeto original a copiar.

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. Para que funcione correctamente, este método deberán ser redefinido en nuestras clases para indicar cuando se considera que dos objetos son iguales. Por ejemplo podemos tener una clase como la siguiente:

```
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);
        }
```

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;
        }
}
```

1.6.2. Properties

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

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

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

Equivalente al método *put*.

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

Equivalente al método *get*.

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

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

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

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

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

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

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

1.6.3. System

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

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

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

Otros métodos útiles que encontramos son:

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

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

```
void gc()
```

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

```
long currentTimeMillis()
```

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

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

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

```
Properties getProperties()
```

Devuelve un objeto Properties con las propiedades del sistema. En estas propiedades podremos encontrar la siguiente información:

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

1.6.4. Runtime

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

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

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

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

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

1.6.5. 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*.

1.6.6. Otras clases

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

Entre ellas tenemos la clase `Locale` que almacena información sobre una determinada región del mundo (país e idioma), y que podrá ser utilizada para internacionalizar nuestra aplicación de forma sencilla. Una clase relacionada con esta última es `ResourceBundle`, con la que podemos definir las cadenas de texto de nuestra aplicación en una serie de ficheros de propiedades (uno para cada idioma). Por ejemplo, podríamos tener dos ficheros `Textos_en.properties` y `Textos_es.properties` con los textos en inglés y en castellano respectivamente. Si abrimos el *bundle* de nombre `Textos`, se utilizará el *locale* de nuestro sistema para cargar los textos del fichero que corresponda. También

encontramos otras clases relacionadas con `Locale`, como por ejemplo `Currency` con información monetaria adaptada a nuestra zona, clases que nos permiten formatear números o fechas (`NumberFormat` y `DateFormat` respectivamente) según las convenciones de nuestra zona, o bien de forma personalizada, y la clase `Calendar`, que nos será útil cuando trabajemos con fechas y horas, para realizar operaciones con fechas, compararlas, o acceder a sus campos por separado.

1.7. Estructuras de datos

En nuestras aplicaciones normalmente trabajamos con diversos conjuntos de atributos que son siempre utilizados de forma conjunta (por ejemplo, los datos de un punto en un mapa: coordenada x, coordenada y, descripcion). Estos datos se deberán ir pasando entre las diferentes capas de la aplicación.

Podemos utilizar el patrón *Transfer Object* para encapsular estos datos en un objeto, y tratarlos así de forma eficiente. Este objeto tendrá como campos los datos que encapsula. En el caso de que estos campos sean privados, nos deberá proporcionar métodos para acceder a ellos. Estos métodos son conocidos como *getters* y *setters*, y nos permitirán consultar o modificar su valor respectivamente. Una vez escritos los campos privados, Eclipse puede generar los *getters* y *setters* de forma automática pinchando sobre el código fuente con el botón derecho del ratón y seleccionando la opción *Source* > *Generate Getters and Setters...*. Por ejemplo, si creamos una clase como la siguiente:

```
public class Punto2D {
        private int x;
        private int y;
        private String descripcion;
}
```

Al generar los *getters* y *setters* con Eclipse aparecerán los siguientes métodos:

```
        public String getDescripcion() {
                return descripcion;
        }
        public void setDescripcion(String descripcion) {
                this.descripcion = descripcion;
        }
        public int getX() {
                return x;
        }
        public void setX(int x) {
                this.x = x;
        }
        public int getY() {
                return y;
        }
        public void setY(int y) {
                this.y = y;
        }
```

Con Eclipse también podremos generar diferentes tipos de constructores para estos objetos. Por ejemplo, con la opción *Source* > *Generate Constructor Using Fields...* generará un constructor que tomará como entrada los campos del objeto que le

indiquemos.

1.7.1. BeanUtils

Idealmente un mismo campo sólo estará en una clase, por ejemplo, los campos correspondientes a los datos personales de un cliente no tienen por qué repetirse una y otra vez en distintas clases. Sin embargo cuando construimos un Transfer Object es bastante común que copiemos datos entre campos que tienen una correspondencia exacta. Por ejemplo, tenemos el siguiente Transfer Object que es muy similar al `Punto2D`:

```
public class Punto3D {
private int x;
private int y;
private int z;
private String descripcion;
/* ...y los getters y setters para los cuatro campos */
}
```

Si necesitamos copiar los datos de `Punto3D` a `Punto2D`, tres de los campos coinciden. (Esta operación sería una proyección del punto sobre el plano XY). Manualmente necesitaríamos hacer:

```
punto2D.setX(punto3D.getX());
punto2D.setY(punto3D.getY());
punto2D.setDescripcion(punto3D.getDescripcion());
```

La clase `BeanUtils`, perteneciente a la biblioteca `commons-beanutils` de Apache, nos proporciona el método `copyProperties(objetoDestino, objetoOrigen)` que permite hacer lo mismo en una sola línea. Se trata de un wrapper que hace uso de la API de `Reflection`. Esta API nos permite, entre otras cosas, examinar modificadores, tipos y campos de un objeto en tiempo de ejecución.

```
BeanUtils.copyProperties(punto2D, punto3D);
```

Lo que importa es que los getters y setters que se vayan a copiar coincidan en el tipo y en su nombre a partir del prefijo `get` y `set`. Aparte de incluir la biblioteca `commons-beanutils` también se requiere incluir `commons-logging`, de la cuál hace uso el método `copyProperties(...)`.

2. Ejercicios de Introducción al lenguaje Java

2.1. Uso de interfaces (1 punto)

Tenemos una aplicación para gestionar nuestra colección de DVDs, en la que podemos añadir películas a la colección, eliminarlas, o consultar la lista de todas las películas que poseemos. En esta aplicación tenemos una clase `FilePeliculaDAO` que nos ofrece los siguientes métodos:

```
public void addPelicula(PeliculaTO p);
public void delPelicula(int idPelicula);
public List<PeliculaTO> getAllPeliculas();
```

Esta clase nos permitirá acceder a los datos de nuestra colección almacenados en un fichero en disco. Siempre que en algún lugar de la aplicación se haga un acceso a los datos se utilizará esta clase. Por ejemplo, si introducimos los datos de una nueva película en un formulario y pulsamos el botón para añadir la película, se invocaría un código como el siguiente:

```
FilePeliculaDAO fpdao = GestorDAO.getPeliculaDAO();
fpdao.addPelicula(pelicula);
```

La clase auxiliar `GestorDAO` tiene un método estático que nos permitirá obtener una instancia de los objetos de acceso a datos desde cualquier lugar de nuestro código. En el caso anterior este método sería algo así como:

```
public static FilePeliculaDAO getPeliculaDAO() {
  return new FilePeliculaDAO();
}
```

Como la aplicación crece de tamaño decidimos pasar a almacenar los datos en una BD en lugar de hacerlo en un fichero. Para ello creamos una nueva clase `JDBCPeliculaDAO`, que deberá ofrecer las mismas operaciones que la clase anterior. ¿Qué cambios tendremos que hacer en nuestro código para que nuestra aplicación pase a almacenar los datos en una BD? (Imaginemos que estamos accediendo a `FilePeliculaDAO` desde 20 puntos distintos de nuestra aplicación).

En una segunda versión de nuestra aplicación tenemos definida una interfaz `IPeliculaDAO` con los mismos métodos que comentados anteriormente. Tendremos también la clase `FilePeliculaDAO` que en este caso implementa la interfaz `IPeliculaDAO`. En este caso el acceso a los datos desde el código de nuestra aplicación se hace de la siguiente forma:

```
IPeliculaDAO pdao = GestorDAO.getPeliculaDAO();
pdao.addPelicula(pelicula);
```

El `GestorDAO` ahora será como se muestra a continuación:

```
public static IPeliculaDAO getPeliculaDAO() {
```

```
  return new FilePeliculaDAO();
}
```

¿Qué cambios tendremos que realizar en este segundo caso para pasar a utilizar una BD? Por lo tanto, ¿qué versión consideras más adecuada?

2.2. Refactorización (1 punto)

En las plantillas de la sesión podemos encontrar un proyecto `lja-filmoteca` en el que tenemos implementada la primera versión de la aplicación anterior. Realiza una refactorización del código para facilitar los cambios en el acceso a datos (deberá quedar como la segunda versión comentada en el ejercicio anterior).

Ayuda

Resultará útil el menú *Refactor* de Eclipse. En él podemos encontrar opciones que nos permitan hacer los cambios necesarios de forma automática.

Una vez hechos los cambios añadir el nuevo DAO `JDBCPeliculaDAO` y probar a cambiar de un DAO a otro (si se ha hecho bien sólo hará falta modificar una línea de código). No hará falta implementar las operaciones de los DAO. Bastará con imprimir mensajes en la consola que nos digan lo que estaría haciendo cada operación.

2.3. Documentación (0.5 puntos)

El proyecto anterior tiene una serie de anotaciones en los comentarios que nos permiten generar documentación Javadoc de forma automática desde Eclipse. Observa las anotaciones puestas en los comentarios, las marcas `@param`, `@return`, `@deprecated`...

Genera la documentación de este proyecto (menú *Project > Generate Javadoc*...) en una subcarpeta `doc` dentro del proyecto actual. Eclipse nos preguntará si queremos establecer este directorio como el directorio de documentación de nuestro proyecto, a lo que responderemos afirmativamente.

Añade comentarios Javadoc a las nuevas clases creadas en el ejercicio anterior y genera nuevamente la documentación. Prueba ahora a escribir código que utilice alguna de las clases de nuestro proyecto, usando la opción de autocompletar de Eclipse. Veremos que junto a cada miembro de la clase nos aparecerá su documentación.

2.4. Centro cultural (1 punto)

Un centro cultural se dedica al préstamo de dos tipos de materiales de préstamo: discos y libros. Para los dos se guarda información general, como su código identificativo, el título y el autor. En el caso de los libros, almacenamos también su número de páginas y un capítulo de muestra, y para los discos el nombre de la discográfica.

También podemos encontrar una serie de documentos con las normas e información sobre el centro de los que guardamos su título, fecha de publicación y texto. Tanto estos documentos como los libros se podrán imprimir (en el caso de los libros se imprimirá el título, los autores, y el capítulo de muestra, mientras que de los documentos se imprimirá su título, su fecha de publicación y su texto).

Escribe la estructura de clases que consideres adecuada para representar esta información en un nuevo proyecto `lja-centrocultural`. Utiliza las facilidades que ofrece Eclipse para generar de forma automática los constructores y *getters* y *setters* necesarios.

2.5. Copia de propiedades con BeanUtils (0.5 puntos)

En las plantillas de la sesión contamos con un proyecto llamado `lja-copyproperties` en el que hay una clase principal que crea dos objetos: uno de clase `PeliculaTO` y otro de clase `Mpeg4fileTO`. Se trata de dos transfer objects muy similares que comparten varios campos.

Copia todos los campos que puedas desde `PeliculaTO` hacia `Mpeg4fileTO`, sobreescribiendo los campos que se pueda y dejando como están los campos que no coincidan.

Ahora utiliza el método `BeanUtils.copyProperties(dest,orig)` para hacer lo mismo en una sola línea. Tendrás que incluir las bibliotecas correspondientes, que se encuentran en la carpeta `lib`.

Observa el resultado para ver si se han copiado bien todos los campos que esperabas. ¿Encuentras algún fallo en la copia de la fecha de estreno? Corrígelo y comprueba que se copia correctamente.

3. Colecciones de datos

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

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

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

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

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

3.1. Colecciones

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

Es el tipo más genérico en cuanto a que se refiere a cualquier tipo que contenga un grupo de elementos. Viene definido por la interfaz `Collection`, de la cual heredará cada

subtipo específico. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

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

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

```
void clear()
```

Elimina todos los elementos de la colección.

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

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

```
boolean isEmpty()
```

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

```
Iterator iterator()
```

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

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

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

```
int size()
```

Nos devuelve el número de elementos que contiene la colección.

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

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

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

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

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

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

3.1.1. Listas de elementos

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

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

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

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

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

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

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

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

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

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

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

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

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

ArrayList

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

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

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

Vector

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

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

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

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

Añade un elemento al final del vector.

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

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

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

Inserta un elemento en la posición indicada.

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

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

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

Elimina el elemento de la posición indicada en el índice.

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

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

```
int size()
```

Devuelve el número de elementos del vector.

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

muchos navegadores corresponde a versiones antiguas.

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

LinkedList

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

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

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

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

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

Obtiene el primer / último objeto de la lista respectivamente.

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

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

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

3.1.2. Conjuntos

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

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

HashSet

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

LinkedHashSet

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

TreeSet

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

3.1.3. Mapas

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

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

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

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

Nos devuelve el valor asociado a la clave indicada

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

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

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

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

```
Set keySet()
```

Nos devuelve el conjunto de claves registradas

```
int size()
```

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

Encontramos distintas implementaciones de los mapas:

HashMap

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

TreeMap

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

Hashtable

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

```
Enumeration keys()
```

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

3.1.4. Wrappers

La clase `Collections` aporta una serie métodos para cambiar ciertas propiedades de las listas. Estos métodos nos proporcionan los denominados *wrappers* de los distintos tipos de colecciones. Estos *wrappers* son objetos que 'envuelven' al objeto de nuestra colección, pudiendo de esta forma hacer que la colección esté sincronizada, o que la colección pase a ser de solo lectura.

Como dijimos anteriormente, todos los tipos de colecciones no están sincronizados,

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

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

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

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

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

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

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

Podemos encontrar también una serie de *wrappers* para obtener versiones de sólo lectura

de nuestras colecciones. Se obtienen con los siguientes métodos:

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

3.1.5. Genéricos

Podemos tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos. Por ejemplo, podemos crear un `ArrayList` que sólo almacene `Strings`, o una `HashMap` que tome como claves `Integers` y como valores `ArrayLists`. Además, con esto nos ahorramos las conversiones *cast* al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

Ejemplo

```
// Vector de cadenas
ArrayList<String> a = new ArrayList<String>();
a.add("Hola");
String s = a.get(0);
a.add(new Integer(20));                // Daría error!!

// HashMap con claves enteras y valores de vectores
HashMap<Integer, ArrayList> hm = new HashMap<Integer, ArrayList>();
hm.put(1, a);
ArrayList a2 = hm.get(1);
```

A partir de JDK 1.5 deberemos utilizar genéricos siempre que sea posible. Si creamos una colección sin especificar el tipo de datos que contendrá normalmente obtendremos un *warning*.

Los genéricos no son una característica exclusiva de las colecciones, sino que se pueden utilizar en muchas otras clases, incluso podemos parametrizar de esta forma nuestras propias clases.

3.1.6. Recorrer las colecciones

Vamos a ver ahora como podemos iterar por los elementos de una colección de forma eficiente y segura, evitando salirnos del rango de datos. Dos elementos utilizados comunmente para ello son las enumeraciones y los iteradores.

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

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

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

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

```
enum.hasMoreElements()
```

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

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

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

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

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

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

```
iter.hasNext()
```

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

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

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

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

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

A partir de JDK 1.5 podemos recorrer colecciones y arrays sin necesidad de acceder a sus iteradores, previniendo índices fuera de rango.

Ejemplo

```
// Recorre e imprime todos los elementos de un array
int[] arrayInt = {1, 20, 30, 2, 3, 5};
for(int elemento: arrayInt)
        System.out.println (elemento);
```

```
// Recorre e imprime todos los elementos de un ArrayList
ArrayList<String> a = new ArrayList<String>();
for(String cadena: a)
        System.out.println (cadena);
```

3.1.7. Cuestiones de eficiencia

Tradicionalmente Java se ha considerado un lenguaje lento. Hoy en día Java se utiliza en aplicaciones con altísimas exigencias de rendimiento y rapidez de respuesta, por ejemplo, Apache SolR. Para obtener un rendimiento adecuado es fundamental utilizar las estructuras de datos idóneas para cada caso, así como los métodos adecuados.

Por ejemplo hay que tener en cuenta que una lista mantiene un orden (anterior y siguiente), mientras que un `ArrayList` mantiene elementos en posiciones. Si eliminamos un elemento al principio de la lista, todos los demás son desplazados una posición.

Métodos como `addAll` o `removeAll` son preferibles a un bucle que itere sobre la lista.

En general es bueno pensar en cuál va a ser el principal uso de una estructura de datos y considerar su complejidad computacional. Hacer una prueba de tiempos con una cantidad limitada de datos puede darnos una idea errónea, si no probamos distintos tamaños de los datos. En la siguiente figura se muestran las complejidades computacionales de algunos métodos de colecciones:

Listas y conjuntos

Estructura	get	add	remove	contains
ArrayList	O(1)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(1)	O(n)
HashSet	O(1)	O(1)	O(1)	O(1)
LinkedHashSet	O(1)	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)

Mapas:

Estructura	get	put	remove	containsKey
HashMap	O(1)	O(1)	O(1)	O(1)
LinkedHashMap	O(1)	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)

Complejidad computacional de métodos de colecciones

Otras curiosidades que vale la pena conocer están enumeradas en "5 things you didn't know about the Java Collecitons Api":

http://www.ibm.com/developerworks/java/library/j-5things2/index.html
http://www.ibm.com/developerworks/java/library/j-5things3/index.html.

3.2. Comparación de objetos

Comparar objetos es fundamental para hacer ciertas operaciones y manipulaciones en estructuras de datos. Por ejemplo, saber si un objeto es igual a otro es necesario a la hora de buscarlo en una estructura de datos.

3.2.1. Sobrecarga de equals

Todos los `Object` y clases derivadas tienen un método `equals(Object o)` que compara un objeto con otro devolviendo un booleano verdadero en caso de igualdad. El criterio de igualdad puede ser personalizado, según la clase. Para personalizarlo se puede sobrecargar el método de comparación:

```
public class MiClase {
        ...
        @Override
        public boolean equals(Object o) {
                // return true o false, según un criterio
        }
}
```

El método `equals` no debe sobrecargarse si no es necesario. Sobre todo hay que evitar sobrecargarlo en casos como los siguientes:

- Cada instancia es intrínsecamente única. Por ejemplo, instancias de hilos, que representan entidades activas, y no tan sólo un conjunto de valores.
- Cuando no es necesaria una comparación lógica. Por ejemplo, dos números aleatorios, donde la igualdad puede ocurrir pero su comprobación no es necesaria.
- Una superclase ya sobrecarga `equals`, y el comportamiento de éste es apropiado para la clase actual.

Cuando se sobrecarga el método `equals` se deben cumplir las siguientes propiedades:

- Reflexividad: `x.equals(x)` devuelve siempre verdadero, si no es nulo.
- Simetría: para cualquier par de instancias no nulas, `x.equals(y)` devuelve verdadero si y sólo si `y.equals(x)` también devuelve verdadero.
- Transitividad: si `x.equals(y)==true` y `y.equals(z)==true`, entonces `x.equals(z)` también será verdadero, para cualesquiera instancias no nulas.
- Consistencia: múltiples llamadas al método con las mismas instancias devuelven el mismo resultado.
- Comparación con `null` falsa: `x.equals(null)` devuelve falso

Para asegurar la propiedad de consistencia también conviene sobrecargar el método `hashCode`, que es necesario para que funcionen correctamente todas las colecciones basadas en códigos hash, como `HashMap`, `HashSet`, `Hashtable`. Objetos que se consideren iguales deben devolver `hashCode` iguales. Debe cumplirse:

- Cuando `hashCode` es invocado varias veces para el mismo objeto, debe devolver consistentemente el mismo entero, siempre que no se haya modificado ninguna información que afecte al resultado de `equals`. Esta consistencia debe mantenerse entre distintas ejecuciones de la misma aplicación.
- Si dos objetos son iguales según `equals`, entonces los métodos `hashCode` de ambos deben devolver el mismo entero.
- Si dos objetos no son iguales según `equals`, **no** se require que devuelvan `hashCode` diferentes. No obstante en la medida de lo posible deben ser distintos porque esto puede mejorar la eficiencia de las tablas hash.

3.2.2. Implementación de Comparable

Hay algoritmos, como `Collections.sort( )`, que requieren que los objetos tengan un método `compareTo()` que devuelva un número negativo, positivo o cero, según si un objeto es menor que el otro, mayor, o igual. Este método no viene en `Object` para poder sobrecargarlo, sino en la interfaz `Comparable` que tenemos que implementar, y que nos obligará a implementar también el método `compareTo`.

Por supuesto, no todos los objetos se pueden comparar en términos de mayor o menor. Así, el hecho de que una clase implemente `Comparable` nos indica que se trata de una estructura de datos cuyos objetos sí son comparables, y por tanto podrían ordenarse.

Un ejemplo de implementación de `Comparable`:

```
public class Persona implements Comparable<Persona> {
        public int id;
        public String apellido;
        ...
        @Override
        public int compareTo(Persona p) {
                return this.id - p.id;
        }

}
```

3.2.3. Comparador externo

En muchas estructuras de datos la ordenación podría ser subjetiva. Por ejemplo, las fichas de clientes podrían considerarse mayores o menores según el identificador, según el apellido o según la fecha de alta. La estructura de datos no tiene por qué ofrecer todas las posibilidades de comparación. En estos casos, en los que no hay un sólo orden inherente a la estructura de datos, podemos utilizar un comparador externo.

Para ello tenemos que implementar la interfaz `Comparator` que nos obliga a implementar el método `compare`. Al tratarse, una vez más, de una interfaz, podríamos hacerlo dentro de la propia clase cuyas instancias vamos a comparar, o bien en otra clase aparte, como en el siguiente ejemplo:

```
public class ComparaPersonaPorNombre implements Comparator<Persona>{
        public int compare(Persona p1, Persona p2) {
                return p1.apellido.compareToIgnoreCase(p2.apellido);
        }
}
```

Para hacer uso de ese comparador externo en algún método, debemos indicarlo pasando una instancia del `Comparator`. En cambio si queremos utilizar el método de comparación `Comparable.compareTo( )`, sobra con que la clase implemente `Comparable`.

```
List personas = new ArrayList<Persona>();
personas.add(p1); personas.add(p2); personas.add(p3); //...

Collections.sort(personas); //Comparable.compareTo
Collections.sort(personas, new ComparaPersonaPorNombre());
//Comparator.compare
```

3.3. Polimorfismo e interfaces

En Java podemos conseguir tener objetos polimórficos mediante la implementación de interfaces. Un claro ejemplo está en las colecciones vistas anteriormente. Por ejemplo, todos los tipos de listas implementan la interfaz `List`. De esta forma, en un método que acepte como entrada un objeto de tipo `List` podremos utilizar cualquier tipo que implemente esta interfaz, independientemente del tipo concreto del que se trate.

Es por lo tanto recomendable hacer referencia siempre a estos objetos mediante la interfaz que implementa, y no por su tipo concreto. De esta forma posteriormente podríamos cambiar la implementación del tipo de datos sin que afecte al resto del programa. Lo único que tendremos que cambiar es el momento en el que se instancia.

Por ejemplo, si tenemos una clase `Cliente` que contiene una serie de cuentas, tendremos algo como:

```
public class Cliente {
    String nombre;
    List<Cuenta> cuentas;

    public Cliente(String nombre) {
        this.nombre = nombre;
        this.cuentas = new ArrayList<Cuenta>();
    }

    public List<Cuenta> getCuentas() {
        return cuentas;
    }

    public void setCuentas(List<Cuenta> cuentas) {
        this.cuentas = cuentas;
    }

    public void addCuenta(Cuenta cuenta) {
        this.cuentas.add(cuenta);
    }
```

```
}
```

Si posteriormente queremos cambiar la implementación de la lista a `LinkedList` por ejemplo, sólo tendremos que cambiar la línea del constructor en la que se hace la instanciación.

Como ejemplo de la utilidad que tiene el polimorfismo podemos ver los algoritmos predefinidos con los que contamos en el marco de colecciones.

3.3.1. Ejemplo: Algoritmos

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

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

Estos métodos tienen como parámetro de entrada un objeto de tipo `List`. De esta forma, podremos utilizar estos algoritmos para cualquier tipo de lista.

3.4. Tipos de datos básicos en las colecciones

3.4.1. Wrappers de tipos básicos

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

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

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

3.4.2. Autoboxing

Esta característica aparecida en JDK 1.5 evita al programador tener que establecer correspondencias manuales entre los tipos simples (`int`, `double`, etc) y sus correspondientes *wrappers* o tipos complejos (`Integer`, `Double`, etc). Podremos utilizar un `int` donde se espere un objeto complejo (`Integer`), y viceversa.

Ejemplo

```
ArrayList<Integer> a = new ArrayList<Integer>();
a.add(30);
Integer n = v.get(0);
n = n+1;
int num = n;
```

4. Ejercicios de colecciones

4.1. Implementaciones e interfaces (1 punto)

Tenemos una clase como la siguiente para encapsular los datos de una película, en la que se almacena, entre otros campos, la lista de actores:

```
public class PeliculaTO {
    String titulo;
    ArrayList<String> actores;
    ArrayList<String> directores;

    public PeliculaTO() {
            actores = new ArrayList<String>();
            directores = new ArrayList<String>();
    }

    public ArrayList<String> getActores() {
        return actores;
    }

    public void addActor(String actor) {
        actores.add(actor);
    }
}
```

Como segunda opción, tenemos una implementación alternativa como la siguiente:

```
public class PeliculaTO {
    String titulo;
    List<String> actores;
    List<String> directores;

    public PeliculaTO() {
            actores = new ArrayList<String>();
            directores = new ArrayList<String>();
    }

    public List<String> getActores() {
        return actores;
    }

    public void addActor(String actor) {
        actores.add(actor);
    }
}
```

Imaginemos que más adelante comprobamos que se hacen un gran número de operaciones de borrado o inserción en mitad de la lista, y decidimos que sería más eficiente utilizar un `LinkedList`. Si nuestra clase pertenece a una librería que se está utilizando desde múltiples puntos de una gran aplicación, ¿qué cambios implicaría el pasar a utilizar una lista enlazada en cada una de las dos versiones? ¿Cuál de ellas consideras por lo tanto más apropiada?

4.2. Uso de listas (1 punto)

Vamos a añadir a nuestro proyecto de gestión de filmotecas de la sesión anterior un nuevo DAO que manejará datos en memoria, en lugar de guardarlos en fichero o base de datos. A este DAO le llamaremos `MemoryPeliculaDAO`, y utilizará internamente colecciones para almacenar las películas. Deberemos poder añadir películas, eliminarlas, o ver la lista de todas las películas que tengamos. Se pide:

a) Las operaciones más comunes que haremos en la aplicación van a consistir en añadir una película al final de la lista, eliminar una película dado su identificador (habrá que buscarla en la lista), u obtener el listado de todas las películas y recorrerlo entero para mostrarlo. ¿Qué tipo de colección consideras más apropiada para almacenar esta información?.

b) Añadir el código necesario a las operaciones para agregar películas y consultar la lista de películas disponibles. Comprobar que la aplicación funciona correctamente.

c) Consideraremos que dos películas son la misma si su identificador coincide. Añadir el código necesario a la clase `PeliculaTO` para que esto sea así. Comprobar que funciona correctamente implementando el método para eliminar películas (si al método `remove` de la colección se le pasa como parámetro un objeto `PeliculaTO` con el mismo identificador que una de las películas ya existentes en dicha colección, deberá eliminarla de la lista).

d) Al obtener la lista de películas almacenadas, mostrarlas ordenadas alfabéticamente. Utilizar para ello los algoritmos que se nos proporcionan en el marco de colecciones.

Nota

Si internamente estamos almacenando las películas en un tipo de colección sin información de orden, para ordenarlas tendríamos que volcar esta colección a otro tipo de colección que si que sea ordenable. Es más, aunque internamente se almacenen en una colección con orden, siempre es recomendable volcar los elementos a otra colección antes de devolverla, para evitar que se pueda manipular directamente desde cualquier lugar del código la estructura interna en la que guardamos la información. Además las películas deberán ser *comparables*, para que los algoritmos sepan en qué orden se deben ordenar.

e) Utilizar otro tipo de colección para almacenar las películas en memoria. Si se ha hecho un diseño correcto, esto no debería implicar más que el cambio de una línea de código.

f) Para cada película, en lugar de almacenar únicamente el nombre de los actores, nos interesa almacenar el nombre del actor y el nombre del personaje al que representa en la película. Utiliza el tipo de datos que consideres más adecuado para almacenar esta información.

5. Tratamiento de errores

5.1. Introducción

Java es un lenguaje compilado, por tanto durante el desarrollo pueden darse dos tipos de errores: los de tiempo de compilación y los de tiempo de ejecución. En general es preferible que los lenguajes de compilación estén diseñados de tal manera que la compilación pueda detectar el máximo número posible de errores. Es preferible que los errores de tiempo de tiempo de ejecución se deban a situaciones inesperadas y no a descuidos del programador. Errores de tiempo de ejecución siempre habrá, y su gestión a través de excepciones es fundamental en cualquier lenguaje de programación actual.

5.2. Errores en tiempo de ejecución: Excepciones

Los errores en tiempo de ejecución son aquellos que ocurren de manera inesperada: disco duro lleno, error de red, división por cero, cast inválido, etc. Todos estos errores pueden ser manejados a través de excepciones. También hay errores debidos a tareas multihilo que ocurren en tiempo de ejecución y no todos se pueden controlar. Por ejemplo un bloqueo entre hilos sería muy difícil de controlar y habría que añadir algún mecanismo que detecte esta situación y mate los hilos que corresponda.

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.

5.2.1. Tipos de excepciones

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

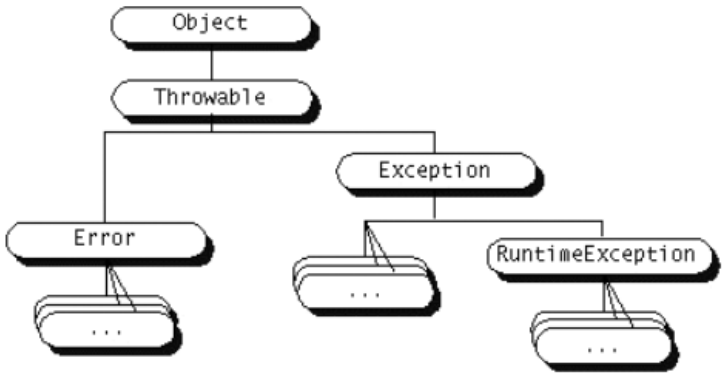
- **`Error`**: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **`Exception`**: Representa errores que no son críticos y por lo tanto pueden ser tratados y continuar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

Dentro de `Exception`, cabe destacar una subclase especial de excepciones denominada `RuntimeException`, de la cual derivarán todas aquellas excepciones referidas a los errores que comúnmente se pueden producir dentro de cualquier fragmento de código,

como por ejemplo hacer una referencia a un puntero `null`, o acceder fuera de los límites de un *array*.

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

- Utilizar excepciones *unchecked* (no predecibles) para indicar errores graves en la lógica del programa, que normalmente no deberían ocurrir. Se utilizarán para comprobar la consistencia interna del programa.
- Utilizar excepciones *checked* para mostrar errores que pueden ocurrir durante la ejecución de la aplicación, normalmente debidos a factores externos como por ejemplo la lectura de un fichero con formato incorrecto, un fallo en la conexión, o la entrada de datos por parte del usuario.



Tipos de excepciones

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

5.2.2. Captura de excepciones

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

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

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

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

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

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

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

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

Un ejemplo de uso:

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

Nunca deberemos dejar vacío el cuerpo del `catch`, porque si se produce el error, nadie se va a dar cuenta de que se ha producido. En especial, cuando estemos con excepciones *no-checked*.

5.2.3. Lanzamiento de excepciones

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

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

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

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

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

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

- Juntando estos dos pasos:

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

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

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 la salida de nuestro programa mostrando el error correspondiente.

5.2.4. Creación de nuevas excepciones

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

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

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

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

5.2.5. Nested exceptions

Cuando dentro de un método de una librería se produce una excepción, normalmente se propagará dicha excepción al llamador en lugar de gestionar el error dentro de la librería, para que de esta forma el llamador tenga constancia de que se ha producido un determinado error y pueda tomar las medidas que crea oportunas en cada momento. Para pasar esta excepción al nivel superior puede optar por propagar la misma excepción que le ha llegado, o bien crear y lanzar una nueva excepción. En este segundo caso la nueva excepción deberá contener la excepción anterior, ya que de no ser así perderíamos la información sobre la causa que ha producido el error dentro de la librería, que podría sernos de utilidad para depurar la aplicación. Para hacer esto deberemos proporcionar la excepción que ha causado el error como parámetro del constructor de nuestra nueva excepción:

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

En el método de nuestra librería en el que se produzca el error deberemos capturar la excepción que ha causado el error y lanzar nuestra propia excepción al llamador:

```
try {
    ...
} catch(IOException e) {
    throw new MiExcepcion("Mensaje de error", e);
}
```

Cuando capturemos una excepción, podemos consultar la excepción previa que la ha causado (si existe) con el método:

```
Exception causa = (Exception)e.getCause();
```

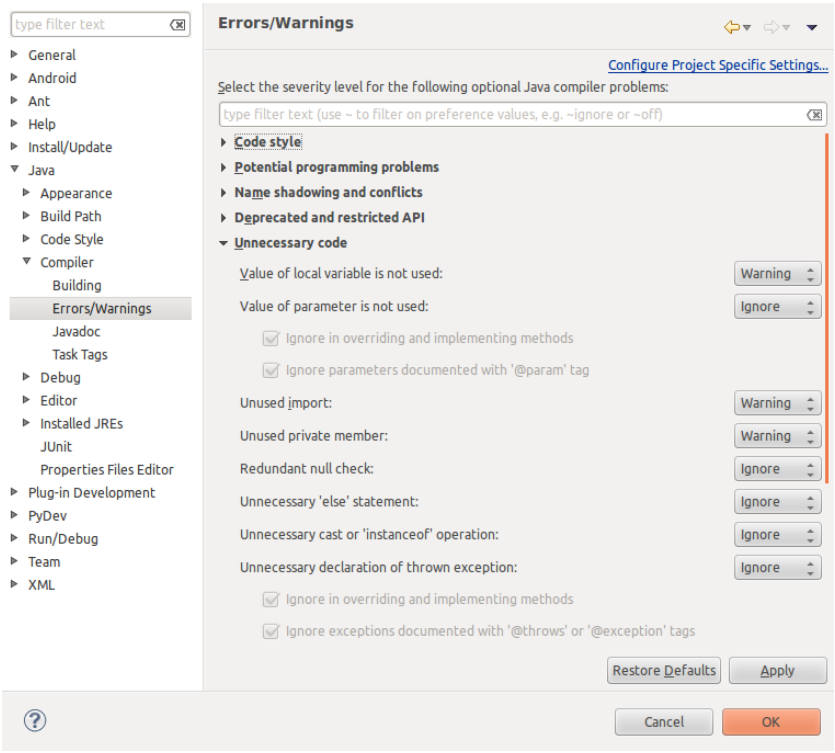
Las *nested exceptions* son útiles para:

- Encadenar errores producidos en la secuencia de métodos a los que se ha llamado.
- Facilitan la depuración de la aplicación, ya que nos permite conocer de dónde viene el error y por qué métodos ha pasado.
- El lanzar una excepción propia de cada método permite ofrecer información más detallada que si utilizásemos una única excepción genérica. Por ejemplo, aunque en varios casos el origen del error puede ser una `IOException`, nos será de utilidad saber si ésta se ha producido al guardar un fichero de datos, al guardar datos de la configuración de la aplicación, al intentar obtener datos de la red, etc.
- Aislar al llamador de la implementación concreta de una librería. Por ejemplo, cuando utilicemos los objetos de acceso a datos de nuestra aplicación, en caso de error recibiremos una excepción propia de nuestra capa de acceso a datos, en lugar de una excepción propia de la implementación concreta de esta capa, como pudiera ser `SQLException` si estamos utilizando una BD SQL o `IOException` si estamos accediendo a ficheros.

5.3. Errores en tiempo de compilación

En Java existen los errores de compilación y las advertencias (warnings). Las advertencias no son de resolución obligatoria mientras que los errores sí, porque no dejan al compilador compilar el código. Es preferible no dejar advertencias porque suelen indicar algún tipo de incorrección. Además, en versiones antiguas de Java, cosas que se consideraban una advertencia han pasado a ser un error. Sobre todo para el trabajo en equipo es una buena práctica no dejar ninguna advertencia en el código que subimos al repositorio.

Eclipse nos ayuda enormemente indicando los errores y advertencias conforme escribimos. Para obligarnos a mejorar la calidad de nuestro código podemos indicar a Eclipse que incremente el nivel de advertencias/errores en gran diversidad de casos. Se puede configurar en el menú de `Preferences / Java / Compiler / Errors`.



Configuración de errrores y advertencias en Eclipse

Además existen herramientas más avanzadas que nos analizan el código en busca de errores de más alto nivel que los que detecta el compilador. Por ejemplo las herramientas PMD, cuyo nombre se debe a que estas tres letras suenan bien juntas, nos detectan posibles bugs debidos a try/catch o switch vacíos, código que no se alcanza o variables y parámetros que no se usan, expresiones innecesariamente complejas, código que maneja strings y buffers de manera subóptima, clases con complejidad cyclomática alta, y código duplicado. Es fácil utilizar PMD a través de su plugin para Eclipse.

5.3.1. Tipos de errores

Los errores en tiempo de compilación son un mal menor de la programación, ya que el compilador los detecta e indica la causa, a veces incluso proponiendo una solución. Se pueden clasificar en los siguientes tipos de error de compilación:

Errores de sintaxis: el código tecleado no cumple las reglas sintácticas del lenguaje Java, por ejemplo, falta un punto y coma al final de una sentencia o se teclea mal el nombre de una variable (que había sido declarada con otro nombre).

Errores semánticos: código que, siendo sintácticamente correcto, no cumple reglas de más alto nivel, por ejemplo imprimir el valor de una variable a la que no se ha asignado valor tras declararla:

```
public void funcion()
{
    int a;
    Console.println(a);
}
```

```
Prueba.java:12: variable a might not have been initialized
Console.println(a);
                ^
1 error
```

Errores en cascada: no son otro tipo de error, pero son errores que confunden al compilador y el mensaje que éste devuelve puede indicar la causa del error lejos de donde realmente está. Por ejemplo en el siguiente código la sentencia `for` está mal escrita:

```
fo ( int i = 0; i < 4; i++ )
{
}
```

```
Prueba.java:24: '.class' expected
        fo ( int i = 0; i < 4; i++ )
                  ^
Prueba.java:24: ')' expected
        fo ( int i = 0; i < 4; i++ )
                    ^
Prueba.java:24: not a statement
        fo ( int i = 0; i < 4; i++ )
                       ^
Pueba.java:24: ';' expected
        fo ( int i = 0; i < 4; i++ )
                                 ^
Prueba.java:24: unexpected type
required: value
found   : class
        fo ( int i = 0; i < 4; i++ )
             ^
Prueba.java:24: cannot resolve symbol
symbol  : variable i
location: class Prueba
        fo ( int i = 0; i < 4; i++ )
                               ^
6 errors
```

Otro problema que crea confusión con respecto a la localización del error son las llaves mal cerradas. Esto se debe a que el compilador de Java no tiene en cuenta la indentación de nuestro código. Mientras que el programador puede ver, a través de la indentación, dónde falta cerrar la llave de una función, bucle o clase, el compilador podría darse cuenta al terminar de leer el archivo e indicarlo ahí.

5.3.2. Comprobación de tipos: Tipos genéricos

En Java hay muchas estructuras de datos que están preparadas para almacenar cualquier

tipo de objeto. Así, en lugar de que exista un `ArrayList` que reciba y devuelva enteros, éste recibe y devuelve objetos. Devolver objetos se convierte en una molestia porque hay que hacer un cast explícito, por ejemplo, `Integer i = (Integer)v.get(0);` cuando el programador sabe perfectamente que este array sólo podrá tener enteros. Pero el problema es mayor, este cast, si no es correcto, provoca un error en tiempo de ejecución. Véase el ejemplo:

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Error en tiempo de ejecución
```

Para evitar esta situación a partir de Java 1.5 se introdujeron los tipos genéricos, que nos fuerzan a indicar el tipo devuelto, únicamente en la declaración de la clase de la instancia. A partir de ahí se hará uso de la estructura sin tener que hacer cast explícitos. El anterior ejemplo quedaría así:

```
List<String> v = new ArrayList<String>();
v.add("test");
String  s = v.get(0); // Correcto (sin necesidad de cast explícito)
Integer i = v.get(0); // Error en tiempo ce compilación
```

Los tipos básicos como `int, float,` etc, no se pueden utilizar en los tipos genéricos.

5.3.2.1. Definición de genéricos

Para definir que una clase trabaja con un tipo genérico, se añade un identificador, por ejemplo `<E>` entre los símbolos menor y mayor, al final del nombre de dicha clase. En el siguiente código se muestra un pequeño extracto de la definición de las **interfaces** `List` e `Iterator`:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

No sólo las interfaces pueden tener tipos genéricos, sino también las **clases**, siguiendo la misma sintaxis:

```
public class Entry<K, V> {

  private final K key;
  private final V value;

  public Entry(K k,V v) {
    key = k;
```

```
    value = v;
  }

  public K getKey() {
    return key;
  }

  public V getValue() {
    return value;
  }

  public String toString() {
    return "(" + key + ", " + value + ")";
  }
}
```

Para usar la clase genérica del ejemplo anterior, declaramos objetos de esa clase, indicando con qué tipos concretos trabajan en cada caso:

```
Entry<String, String>  grade440 = new Entry<String, String>("mike", "A");
Entry<String, Integer> marks440 = new Entry<String, Integer>("mike", 100);
System.out.println("grade: " + grade440);
System.out.println("marks: " + marks440);
```

Por último, también los **métodos** se pueden definir con tipos genéricos:

```
public static <T> Entry<T,T> twice(T value) {
   return new SimpleImmutableEntry<T,T>(value, value);
}
```

Este método utiliza el tipo genérico para indicar qué genéricos tiene la clase que el método devuelve, y también utiliza ese mismo tipo genérico para indicar de qué tipo es el argumento del método. Al usar el método, el tipo podría ser indicado o podría ser inferido por el compilador en lugar de declararlo:

```
Entry<String, String> pair = this.<String>twice("Hello"); // Declarado
Entry<String, String> pair = twice("Hello");              // Inferido
```

5.3.2.2. Subtipos y comodines

Se debe advertir que, contrariamente a la intuición, si una clase `Hija` es subtipo (subclase o subinterfaz) de una clase `Padre`, y por ejemplo `ArrayList` es una clase genérica, entonces `ArrayList<Hija>` **no** es subtipo de `ArrayList<Padre>`

Existe una forma de flexibilizar el tipado genérico a través de "wildcards" o comodines. Si queremos que una clase con genéricos funcione para tipos y sus subtipos, podemos utilizar el comodín `<?>` junto con la palabra clave `extends` para indicar a continuación cuál es la clase/interfaz de la que hereda:

```
ArrayList<? extends Padre>
```

```
```

De esta manera serían válidos tanto `Padre` como sus clases derivadas: `Hija`. Supongamos ahora que `Padre` hereda de `Abuelo` y sólo queremos que sean válidas estas dos clases. Entonces utilizaremos la palabra clave `super`, como en el siguiente ejemplo, que permitiría `Padre`, `Abuelo` y `Object`, suponiendo que no hay más superclases antes de llegar a `Object`:

```
ArrayList<? super Padre>
```

También está permitido utilizar el comodín sólo, indicando que cualquier tipo es válido.

5.3.2.3. Genéricos y excepciones

Es posible indicar a un método o a una clase el tipo de excepción que debe lanzar, a través de un genérico.

```
public <T extends Throwable> void metodo() throws T {
  throw new T();
}
```

O bien:

```
public class Clase<T extends Throwable>
  public void metodo() throws T {
    throw new T();
  }
}
```

Lo que no es posible es crear excepciones con tipos genéricos, por ejemplo, si creamos nuestra propia excepción para que pueda incluir distintos tipos:

```
public class MiExcepcion<T extends Object> extends Exception {
  private T someObject;

  public MiExcepcion(T someObject) {
    this.someObject = someObject;
  }

  public T getSomeObject() {
    return someObject;
  }
}
```

Tendríamos un problema con las cláusulas `catch`, puesto que cada una debe corresponderse con determinado tipo:

```
```

```
try {
  //Código que lanza o bien MiExcepcion<String>, o bien
MiExcepcion<Integer>
}
catch(MiExcepcion<String> ex) {
  // A
}
catch(MiExcepcion<Integer> ex) {
  // B
}
```

En este código no sería posible saber en qué bloque `catch` entrar, ya que serían idénticos tras la compilación debido al **borrado de tipos**, o "type erasure".

Nota:

Lo que en realidad hace el compilador es comprobar si el uso de tipos genéricos es consistente y después borrarlos, dejando el código sin tipos, como antes de Java 1.5. Los tipos genéricos sólo sirven para restringir el tipado en tiempo de compilación, poniendo en evidencia errores que de otra manera ocurrirían en tiemo de ejecución.

Teniendo eso en cuenta, entendemos por qué no funciona el código del anterior ejemplo. Tras el borrado de tipos queda así:

```
try {
  //Código que lanza o bien MiExcepcion<String>, o bien
MiExcepcion<Integer>
}
catch(MiExcepcion ex) {
  // A
}
catch(MiExcepcion ex) {
  // B
}
```

6. Ejercicios de tratamiento de errores

6.1. Captura de excepciones (0.5 puntos)

En el proyecto `lja-excepciones` de las plantillas de la sesión tenemos una aplicación `Ej1.java` que toma un número como parámetro, y como salida muestra el logaritmo de dicho número. Sin embargo, en ningún momento comprueba si se ha proporcionado algún parámetro, ni si ese parámetro es un número. Se pide:

a) Compilar el programa y ejecutadlo de tres formas distintas:

- Sin parámetros

```
java Ej1
```

- Poniendo un parámetro no numérico

```
java Ej1 pepe
```

- Poniendo un parámetro numérico

```
java Ej1 30
```

Anotad las excepciones que se lanzan en cada caso (si se lanzan)

b) Modificar el código de `main` para que capture las excepciones producidas y muestre los errores correspondientes en cada caso:

- Para comprobar si no hay parámetros se capturará una excepción de tipo `ArrayIndexOutOfBoundsException` (para ver si el *array* de `String` que se pasa en el `main` tiene algún elemento).
- Para comprobar si el parámetro es numérico, se capturará una excepción de tipo `NumberFormatException`.

Así, tendremos en el `main` algo como:

```
try
{
        // Tomar parámetro y asignarlo a un double
} catch (ArrayIndexOutOfBoundsException e1) {
        // Codigo a realizar si no hay parametros
} catch (NumberFormatException e2) {
        // Codigo a realizar con parametro no numerico
}
```

Probad de nuevo el programa igual que en el caso anterior comprobando que las excepciones son capturadas y tratadas.

6.2. Lanzamiento de excepciones (0.5 puntos)

El fichero `Ej2.java` es similar al anterior, aunque ahora no vamos a tratar las excepciones del `main`, sino las del método `logaritmo`: en la función que calcula el logaritmo se comprueba si el valor introducido es menor o igual que 0, ya que para estos valores la función logaritmo no está definida. Se pide:

a) Buscar entre las excepciones de Java la más adecuada para lanzar en este caso, que indique que a un método se le ha pasado un argumento ilegal. (Pista: Buscar entre las clases derivadas de `Exception`. En este caso la más adecuada se encuentra entre las derivadas de `RuntimeException`).

b) Una vez elegida la excepción adecuada, añadir código (en el método `logaritmo`) para que en el caso de haber introducido un parámetro incorrecto se lance dicha excepción.

```
throw new ... // excepcion elegida
```

Probar el programa para comprobar el efecto que tiene el lanzamiento de la excepción.

c) Al no ser una excepción del tipo *checked* no hará falta que la capturemos ni que declaremos que puede ser lanzada. Vamos a crear nuestro propio tipo de excepción derivada de `Exception` (de tipo *checked*) para ser lanzada en caso de introducir un valor no válido como parámetro. La excepción se llamará `WrongParameterException` y tendrá la siguiente forma:

```
public class WrongParameterException extends Exception
{
        public WrongParameterException(String msg) {
                super(msg);
        }
}
```

Deberemos lanzarla en lugar de la escogida en el punto anterior.

```
throw new WrongParameterException(...);
```

Intentar compilar el programa y observar los errores que aparecen. ¿Por qué ocurre esto? Añadir los elementos necesarios al código para que compile y probarlo.

d) Por el momento controlamos que no se pase un número negativo como entrada. ¿Pero qué ocurre si la entrada no es un número válido? En ese caso se producirá una excepción al convertir el valor de entrada y esa excepción se propagará automáticamente al nivel superior. Ya que tenemos una excepción que indica cuando el parámetro de entrada de nuestra función es incorrecto, sería conveniente que siempre que esto ocurra se lance dicha excepción, independientemente de si ha sido causada por un número negativo o por algo que no es un número, pero siempre conservando la información sobre la causa que produjo el error. Utilizar *nested exceptions* para realizar esto.

Ayuda

Deberemos añadir un nuevo constructor a `WrongParameterException` en el que se proporcione la excepción que causó el error. En la función `logaritmo` capturaremos cualquier excepción que se produzca al convertir la cadena a número, y lanzaremos una excepción `WrongParameterException` que incluya la excepción causante.

6.3. Excepciones como tipos genéricos en la aplicación filmotecas(0.5 puntos)

Realiza una copia del proyecto `lja-filmoteca` con otro nombre: `lja-filmoteca-exc`. En este nuevo branch realizaremos cambios que **no** mantendremos para el siguiente ejercicio.

Los métodos de nuestros DAO pueden devolver excepciones. Según si el DAO se dedica a escribir en disco, en memoria o en una base de datos, podrá devolver unas excepciones u otras. Busca la clase de excepción que te parezca adecuada para cada uno de los DAO que tiene la aplicación. Por ejemplo, la `IOException` para `FilePeliculaDAO`.

Añade un tipo genérico `E` a `IPeliculaDAO`, que sólo pueda ser una `Exception` o clases derivadas. De esta manera si tenemos que crear un nuevo `FilePeliculaDAO` y asignarlo a una variable que cumple con la interfaz, tendremos que introducir el tipo de excepción:

```
IPeliculaDAO<IOException> dao = new FilePeliculaDAO();
```

Reliza los cambios necesarios a `FilePeliculaDAO` para que, efectivamente, sus métodos lancen esa excepción (sólo con `throws`, sin implementar código dentro de los métodos).

Supongamos que `MemoryPeliculaDAO` no tuviera que devolver ninguna excepción. Declara adecuadamente la clase

```
public class MemoryPeliculaDAO implements IPeliculaDAO<...> {
```

Comprueba que todos los DAO se pueden devolver correctamente desde GestorDAO, tanto los que devuelven excepciones como los que no.

¿Qué limitación tiene esta aproximación? En el siguiente ejercicio añadiremos tratamiento de excepciones siguiendo otra aproximación diferente, sin usar tipos genéricos.

6.4. Excepciones anidadas en la aplicación filmotecas (1.5 puntos)

En este ejercicio plantearemos el tratamiento de errores de los distintos DAO anidando excepciones en una excepción general para todos los DAO. Así las partes de código que utilicen un DAO de cualquier tipo, sabrán qué excepciones se deben al DAO y además se incluirá excepción concreta que causó el error, por si es necesario saberlo.

Para empezar, en el proyecto `lja-filmoteca`, añadiremos excepciones para tratar los errores en la clase `MemoryPeliculaDAO`. Cuando se produzca un error en esta clase lanzaremos una excepción `DAOException`, de tipo *checked*, que deberemos implementar. Se pide:

a) La excepción `DAOExcepcion` se creará en el mismo paquete que el resto de las clases del DAO. Utilizaremos las facilidades que nos ofrece Eclipse para generar automáticamente los constructores de dicha clase (tendremos suficiente con tener acceso a los constructores de la super-clase).

b) ¿Qué ocurre si declaramos que los métodos de `MemoryPeliculaDAO` pueden lanzar la excepción creada, pero no lo hacemos en la interfaz `IPeliculaDAO`? ¿Y si en `IPeliculaDAO` si que se declara, pero en alguna de las clases que implementan esta interfaz no se hace (`FilePeliculaDAO`, `JDBCPeliculaDAO`)? ¿Por qué crees que esto es así? Todos los métodos de `IPeliculaDAO` podrán lanzar este tipo de excepción.

c) El método `addPelicula` lanzará la excepción si le pasamos como parámetro `null`, si intentamos añadir una película cuyo título sea `null` o cadena vacía, o si ya existe una película en la lista con el mismo título.

d) El método `delPelicula` lanzará la excepción cuando no exista ninguna película con el identificador proporcionado.

e) En algunos casos es posible que fallen las propias operaciones de añadir, eliminar o listar los elementos de las colecciones. Vamos a utilizar *nested exceptions* para tratar estos posibles errores. Para hacer esto añadiremos bloques `try-catch` alrededor de las operaciones con colecciones, y en caso de que se produjese un error lanzaremos una excepción de tipo `DAOException`. Podemos conseguir que se produzca una excepción de este tipo si utilizamos como colección un tipo `TreeSet`. ¿Por qué se producen errores al utilizar este tipo? ¿Qué habría que hacer para solucionarlo?

f) Comprobar que los errores se tratan correctamente utilizando el programa de pruebas que tenemos (clase `Main`). Habrá que hacer una serie de modificaciones en dicho programa principal para tratar las excepciones. Debemos destacar que el haber tratado las posibles excepciones internas de las colecciones mediante *nested exceptions* ahora nos facilitará el trabajo, ya que sólo deberemos preocuparnos de capturar la excepción de tipo `DAOException`, sin importarnos cómo se haya implementado internamente el DAO.

7. Casos de prueba: JUnit

En este tema veremos **JUnit**, una librería desarrollada para poder probar el funcionamiento de las clases y métodos que componen nuestra aplicación, y asegurarnos de que se comportan como deben ante distintas situaciones de entrada.

7.1. Introducción a JUnit

Cuando probamos un programa, lo ejecutamos con unos datos de entrada (casos de prueba) para verificar que el funcionamiento cumple los requisitos esperados. Definimos **prueba unitaria** como la prueba de uno de los módulos que componen un programa.

En los últimos años se han desarrollado un conjunto de herramientas que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. Dicho conjunto se denomina *XUnit*. De entre dicho conjunto, **JUnit** es la herramienta utilizada para realizar pruebas unitarias en Java.

El concepto fundamental en estas herramientas es el **caso de prueba** (*test case*), y la **suite** de prueba (*test suite*). Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba. Mediante las suites podemos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).

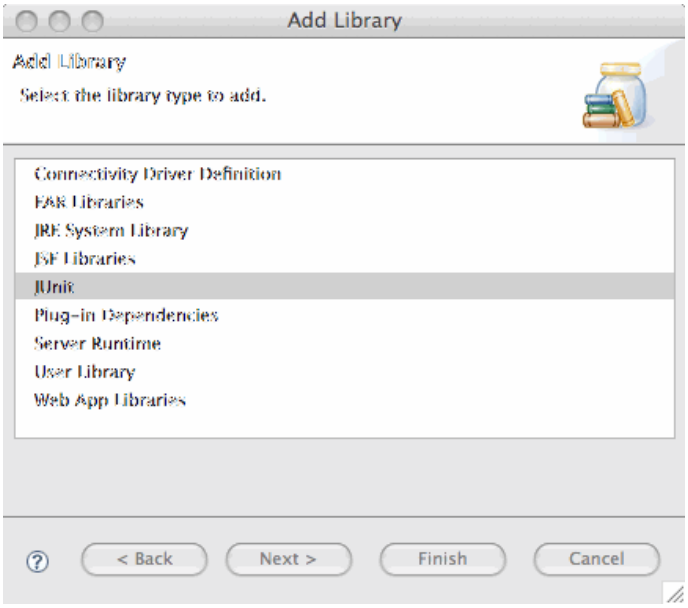
De esta forma, construimos programas que sirven para probar nuestros módulos, y que podremos ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba, que servirá para hacer pruebas de regresión. Eso es importante, puesto que cuando cambiamos un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

Aplicando lo anterior a Java, JUnit es un conjunto de clases opensource que nos permiten probar nuestras aplicaciones Java. Podemos encontrar información actualizada de JUnit en http://www.junit.org

Encontraremos una distribución de JUnit, en la que habrá un fichero JAR, **junit.jar**, que contendrá las clases que deberemos tener en el CLASSPATH a la hora de implementar y ejecutar los casos de prueba.

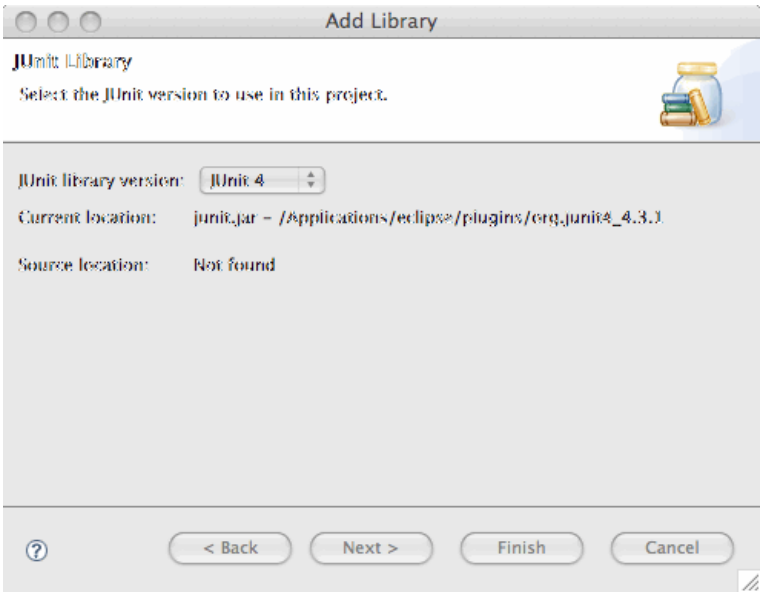
7.2. Integración de JUnit en Eclipse

Eclipse incorpora opciones para poder trabajar con JUnit desde él. Antes de nada, debemos tener nuestro proyecto Java ya creado, o bien crearlo nuevo. Una vez hecho esto, deberemos añadir la librería de JUnit al *build path* del proyecto. La forma más rápida de hacer esto es pulsar con el botón derecho sobre nuestro proyecto y seleccionar *Java Build Path* > *Add Libraries* ..., de forma que nos aparecerá una ventana en la que podremos elegir la librería a añadir, siendo una de ellas JUnit:



Añadir la librería JUnit a nuestro proyecto Eclipse

Una vez seleccionada la librería JUnit, nos aparecerá otra ventana en la que podremos elegir tanto JUnit 3 como JUnit 4.



Seleccionar versión de la librería

La principal diferencia entre ambas versiones es que en JUnit 3 los casos de prueba se crean mediante herencia, y sobrescribiendo una serie de métodos, mientras que en la nueva versión se hace mediante anotaciones, por lo que es necesario utilizar Java 5 o posterior. También sería posible tener ambas librerías en un mismo proyecto.

Vamos a centrarnos en el estudio de JUnit 4. Una vez añadida la librería, ya podemos crear los casos de prueba de nuestro proyecto proyecto. Si creamos un caso de prueba sin tener todavía añadida la librería JUnit, veremos que Eclipse nos preguntará si queremos añadirla y si respondemos afirmativamente la librería se añadirá automáticamente.

7.3. Un ejemplo sencillo

Supongamos que tenemos una clase `EmpleadoBR` con las reglas de negocio aplicables a los empleados de una tienda. En esta clase encontramos los siguientes métodos con sus respectivas especificaciones:

Método	Especificación

`float calculaSalarioBruto(`
`  TipoEmpleado tipo,`
`  float ventasMes,`
`  float horasExtra)`
	El salario base será 1000 euros si el empleado es de tipo `TipoEmpleado.vendedor`, y de 1500 euros si es de tipo `TipoVendedor.encargado`. A esta cantidad se le sumará una prima de 100 euros si `ventasMes` es mayor o igual que 1000 euros, y de 200 euros si fuese al menos de 1500 euros. Por último, cada hora extra se pagará a 20 euros. Si `tipo` es `null`, o `ventasMes` o `horasExtra` toman valores negativos el

	método lanzará una excepción de tipo `BRException`.
`float calculaSalarioNeto(` `  float salarioBruto)`	Si el salario bruto es menor de 1000 euros, no se aplicará ninguna retención. Para salarios a partir de 1000 euros, y menores de 1500 euros se les aplicará un 16%, y a los salarios a partir de 1500 euros se les aplicará un 18%. El método nos devolverá `salarioBruto * (1-retencion)`, o `BRExcepcion` si el salario es menor que cero.

A partir de dichas especificaciones podemos diseñar un conjunto de casos de prueba siguiendo métodos como el método de pruebas de particiones, también conocido como caja negra. Si en lugar de contar con la especificación, contásemos con el código del método a probar también podríamos diseñar a partir de él un conjunto de casos de prueba utilizando otro tipo de métodos (en este caso se podría utilizar el método de caja blanca). No vamos a entrar en el estudio de estos métodos de prueba, sino que nos centraremos en el estudio de la herramienta JUnit. Por lo tanto, supondremos que después de aplicar un método de pruebas hemos obtenido los siguientes casos de prueba:

Método a probar	Entrada	Salida esperada
`calculaSalarioNeto`	`2000`	`1640`
`calculaSalarioNeto`	`1500`	`1230`
`calculaSalarioNeto`	`1499.99`	`1259.9916`
`calculaSalarioNeto`	`1250`	`1050`
`calculaSalarioNeto`	`1000`	`840`
`calculaSalarioNeto`	`999.99`	`999.99`
`calculaSalarioNeto`	`500`	`500`
`calculaSalarioNeto`	`0`	`0`
`calculaSalarioNeto`	`-1`	`BRException`
`calculaSalarioBruto`	`vendedor`, 2000 euros, 8h	`1360`
`calculaSalarioBruto`	`vendedor`, 1500 euros, 3h	`1260`
`calculaSalarioBruto`	`vendedor`, 1499.99 euros, 0h	`1100`
`calculaSalarioBruto`	`encargado`, 1250 euros, 8h	`1760`
`calculaSalarioBruto`	`encargado`, 1000 euros, 0h	`1600`
`calculaSalarioBruto`	`encargado`, 999.99 euros, 3h	`1560`
`calculaSalarioBruto`	`encargado`, 500 euros, 0h	`1500`

`calculaSalarioBruto`	`encargado`, 0 euros, 8h	`1660`
`calculaSalarioBruto`	`vendedor`, -1 euros, 8h	`BRException`
`calculaSalarioBruto`	`vendedor`, 1500 euros, -1h	`BRException`
`calculaSalarioBruto`	`null`, 1500 euros, 8h	`BRException`

Nota

Los casos de prueba se pueden diseñar e implementar antes de haber implementado el método a probar. De hecho, es recomendable hacerlo así, ya que de esta forma las pruebas comprobarán si el método implementado se ajusta a las especificaciones que se dieron en un principio. Evidentemente, esto no se podrá hacer en las pruebas que diseñemos siguiendo el método de caja blanca. También resulta conveniente que sean personas distintas las que se encargan de las pruebas y de la implementación del método, por el mismo motivo comentado anteriormente.

A continuación veremos como implementar estas pruebas utilizando JUnit 4.

7.3.1. Implementación de los casos de prueba

Vamos a utilizar JUnit para probar los métodos anteriores. Para ello deberemos crear una serie de clases en las que implementaremos las pruebas diseñadas. Esta implementación consistirá básicamente en invocar el método que está siendo probado pasándole los parámetros de entrada establecidos para cada caso de prueba, y comprobar si la salida real coincide con la salida esperada. Esto en principio lo podríamos hacer sin necesidad de utilizar JUnit, pero el utilizar esta herramienta nos va a ser de gran utilidad ya que nos proporciona un *framework* que nos obligará a implementar las pruebas en un formato estándar que podrá ser reutilizable y entendible por cualquiera que conozca la librería. El aplicar este *framework* también nos ayudará a tener una batería de pruebas ordenada, que pueda ser ejecutada fácilmente y que nos muestre los resultados de forma clara mediante una interfaz gráfica que proporciona la herramienta. Esto nos ayudará a realizar pruebas de regresión, es decir, ejecutar la misma batería de pruebas en varios momentos del desarollo, para así asegurarnos de que lo que nos había funcionado antes siga funcionando bien.

Para implementar las pruebas en JUnit utilizaremos dos elementos básicos:

- Por un lado, marcaremos con la anotación `@Test` los métodos que queramos que JUnit ejecute. Estos serán los métodos en los que implementemos nuestras pruebas. En estos métodos llamaremos al método probado y comprobaremos si el resultado obtenido es igual al esperado.
- Para comprobar si el resultado obtenido coincide con el esperado utilizaremos los métodos `assert` de la librería JUnit. Estos son una serie de métodos estáticos de la clase `Assert` (para simplificar el código podríamos hacer un *import* estático de dicha clase), todos ellos con el prefijo `assert-`. Existen multitud de variantes de estos métodos, según el tipo de datos que estemos comprobando (`assertTrue`, `assertFalse`, `assertEquals`, `assertNull`, etc). Las llamadas a estos métodos

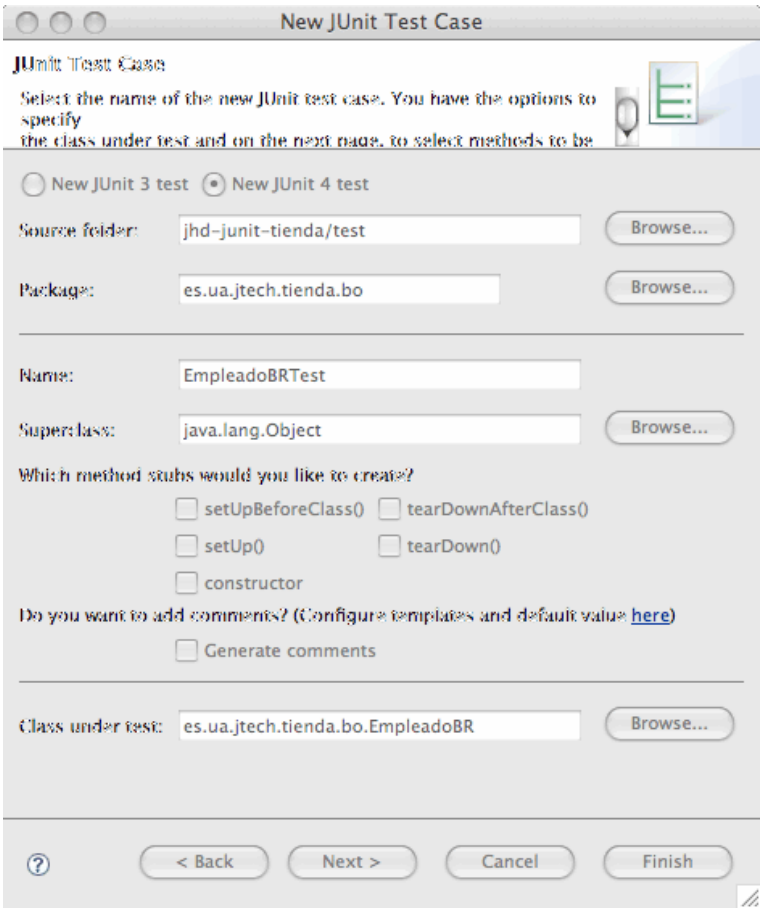
servirán para que JUnit sepa qué pruebas han tenido éxito y cuáles no.

Cuando ejecutemos nuestras pruebas con JUnit, se nos mostrará un informe con el número de pruebas éxitosas y fallidas, y un detalle desglosado por casos de prueba. Para los casos de prueba que hayan fallado, nos indicará además el valor que se ha obtenido y el que se esperaba.

Además de estos elementos básicos anteriores, a la hora de implementar las pruebas con JUnit deberemos seguir una serie de buenas prácticas que se detallan a continuación:

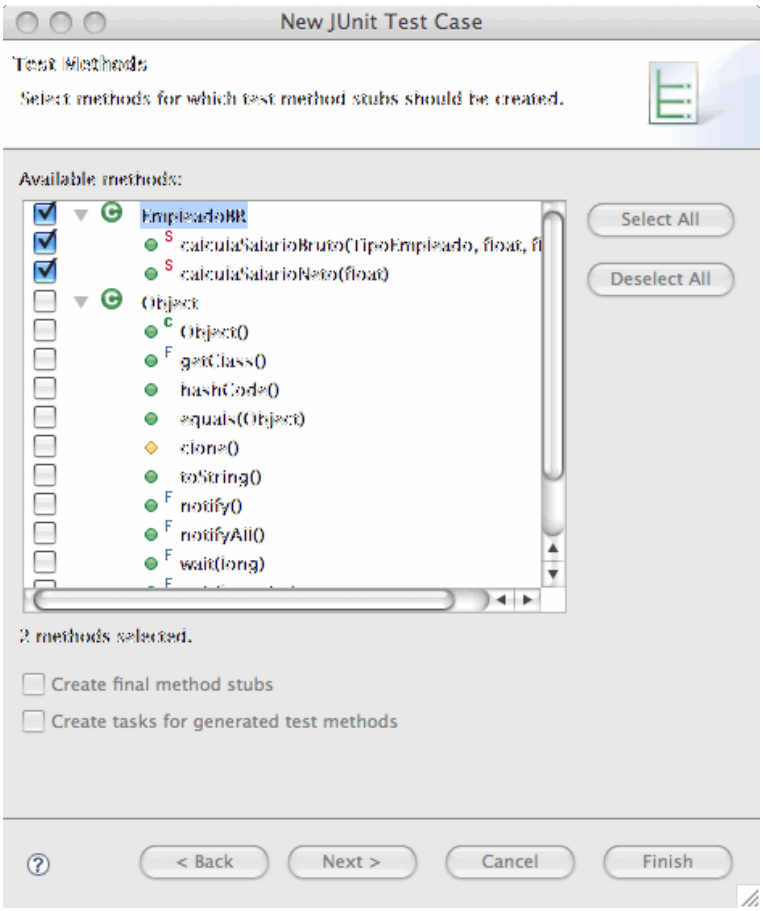
- La clase de pruebas se llamará igual que la clase a probar, pero con el sufijo `-Test`. Por ejemplo, si queremos probar la clase `MiClase`, la clase de pruebas se llamará `MiClaseTest`.
- La clase de pruebas se ubicará en el mismo paquete en el que estaba la clase probada. Si `MiClase` está en el paquete `es.ua.jtech.lja`, `MiClaseTest` pertenecerá e ese mismo paquete. De esta forma nos aseguramos tener acceso a todos los miembros de tipo protegido y paquete de la clase a probar.
- Mezclar clases reales de la aplicación con clases que sólo nos servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no queremos renunciar a poner la clase de pruebas en el mismo paquete que la clase probada. Para solucionar este problema lo que se hará es crear las clases de prueba en un directorio de fuentes diferente. Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado `src`, los fuentes de pruebas irían en un directorio llamado `test`.
- Los métodos de prueba (los que están anotados con `@Test`), tendrán como nombre el mismo nombre que el del método probado, pero con prefijo `test-`. Por ejemplo, para probar `miMetodo` tendríamos un método de prueba llamado `testMiMetodo`.
- Aunque dentro de un método de prueba podemos poner tantos `assert` como queramos, es recomendable crear un método de prueba diferente por cada caso de prueba que tengamos. Por ejemplo, si para `miMetodo` hemos diseñado tres casos de prueba, podríamos tener tres métodos de prueba distintos: `testMiMetodo1`, `testMiMetodo2`, y `testMiMetodo3`. De esta forma, cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado.

Vamos a ver ahora cómo hacer esto desde Eclipse. Lo primero que deberemos hacer es crear un nuevo directorio de fuentes en nuestro proyecto, para tener separados en él los fuentes de prueba. Por ejemplo, podemos llamar a este directorio `test`. Una vez hayamos hecho esto, pincharemos con el botón derecho sobre la clase que queramos probar y seleccionaremos *New > JUnit Test Case*. Nos aparecerá un asitente para crear nuestro caso de prueba JUnit, en el que muchos campos estarán ya rellenos:



Asistente de JUnit

Como podemos observar, tanto el nombre de la nueva clase como el paquete ya lo indica correctamente, pero deberemos cambiar el directorio de fuentes a `test` en lugar de `src`, y la libería utilizada será JUnit 4 en lugar de JUnit 3. Una vez hemos introducido esta información pulsaremos sobre *Next*. Veremos una pantalla en la que podremos seleccionar los métodos que queremos probar:



Selección de métodos a probar

Una vez seleccionados los métodos que nos interesen podremos pulsar el botón `Finish` y nos creará el esqueleto de nuestra clase JUnit. En él deberemos rellenar los métodos de pruebas, o crear nuevos métodos si lo consideramos oportuno. Por ejemplo, la implementación de alguno de los casos de prueba diseñados anteriormente podría quedar como se muestra a continuación:

```
public class EmpleadoBRTest {
  @Test
  public void testCalculaSalarioBruto1() {
    float resultadoReal = EmpleadoBR.calculaSalarioBruto(
                          TipoEmpleado.vendedor, 2000.0f, 8.0f);
    float resultadoEsperado = 1360.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
  }

  @Test
  public void testCalculaSalarioBruto2() {
    float resultadoReal = EmpleadoBR.calculaSalarioBruto(
                          TipoEmpleado.vendedor, 1500.0f, 3.0f);
    float resultadoEsperado = 1260.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
  }
```

```
  @Test
  public void testCalculaSalarioNeto1() {
    float resultadoReal = EmpleadoBR.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
  }

  @Test
  public void testCalculaSalarioNeto2() {
    float resultadoReal = EmpleadoBR.calculaSalarioNeto(1500.0f);
    float resultadoEsperado = 1230.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
  }
}
```

En general la construcción de pruebas sigue siempre estos mismos patrones: llamar al método probado y comprobar si la salida real coincide con la esperada utilizando los métodos `assert`.

7.3.2. Pruebas con lanzamiento de excepciones

En algunos casos de prueba, lo que se espera como salida no es que el método nos devuelva un determinado valor, sino que se produzca una excepción. Para comprobar con JUnit que la excepción se ha lanzado podemos optar por dos métodos diferentes. El más sencillo de ellos es indicar la excepción esperada en la anotación `@Test`:

```
@Test(expected=BRException.class)
public void testCalculaSalarioNeto9() {
  EmpleadoBR.calculaSalarioNeto(-1.0f);
}
```

Otra posibilidad es utilizar el método `fail` de JUnit, que nos permite indicar que hay un fallo en la prueba. En este caso lo que haríamos sería llamar al método probado dentro de un bloque `try-catch` que capture la excepción esperada. Si al llamar al método no saltase la excepción, llamaríamos a `fail` para indicar que no se ha comportado como debería según su especificación.

```
@Test
public void testCalculaSalarioNeto9() {
  try {
    EmpleadoBR.calculaSalarioNeto(-1.0f);
    fail("Se esperaba excepcion BRException");
  } catch(BRException e) {}
}
```

Cuando el método que estemos probando pueda lanzar una excepción de tipo *checked* que debamos capturar de forma obligatoria en JUnit, también podemos utilizar `fail` dentro del bloque `catch` para notificar del fallo en caso de que se lance la excepción de forma no esperada:

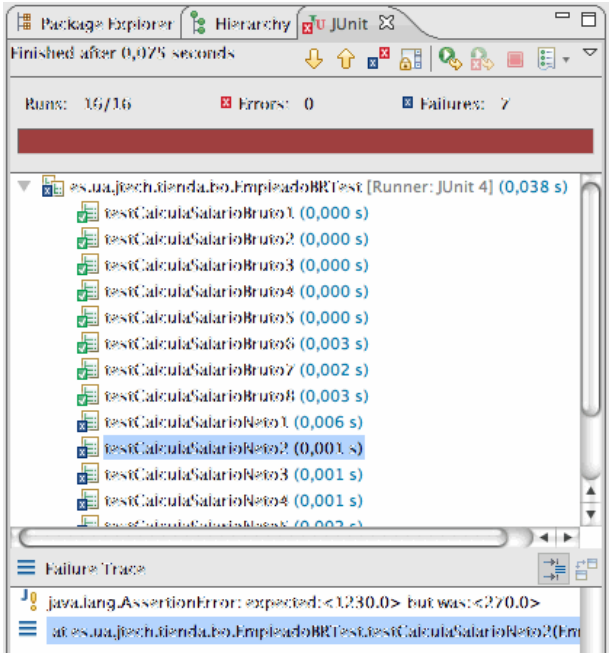
```
public void testCalculaSalarioBruto1() {
  float resultadoReal;
  try {
    resultadoReal = EmpleadoBR.calculaSalarioBruto(
                            TipoEmpleado.vendedor, 2000.0f, 8.0f);
```

```
    float resultadoEsperado = 1360.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
  } catch (BRException e) {
    fail("Lanzada excepcion no esperada BRException");
  }
}
```

Si el método probado lanzase una excepción de tipo *unchecked* no esperada, y no la capturásemos, JUnit en lugar de marcarlo como fallo lo marcará como error.

7.3.3. Ejecución de pruebas

Cuando tengamos definida la clase de prueba que queramos ejecutar, y la clase a probar, pulsamos con el botón derecho sobre la clase de prueba y selecionamos *Run As > JUnit test*. Nos aparecerá la ventana de JUnit en Eclipse con los resultados:



Resultados de las pruebas

Arriba tendremos una barra roja o verde (según si ha habido fallos o no), y el número de pruebas fallidas, y de errores producidos durante la ejecución de las pruebas. En la parte central vemos la jerarquía de todas las pruebas ejecutadas, con un icono que nos indicará si ha tenido éxito o no, y si seleccionamos alguna de las fallidas abajo vemos los detalles del error (resultado obtenido, resultado esperado, y línea de código en la que se ha producido). Haciendo doble click en los errores iremos a la línea de código que los provocó. Se puede relanzar un test pulsando *Ctrl + F11*, o pulsando sobre el botón "*play*".

Ejecutar pruebas fuera de Eclipse

Para ejecutar pruebas por sí solas, debemos utilizar un ejecutor de pruebas (*test runner*).

JUnit proporciona algunos de ellos, como `junit.textui.TestRunner` (para mostrar los resultados en modo texto), o `junit.swingui.TestRunner` (para mostrar los resultados gráficamente). Para ejecutarlos podemos incluir el jar *junit.jar* en el CLASSPATH al ejecutar:

```
java -cp ./junit.jar junit.swingui.TestRunner
```

Nos aparecería una ventana donde indicamos el nombre del caso de prueba que queremos ejecutar (o lo elegimos de una lista), y luego pulsando *Run* nos mostrará los resultados. La barra verde aparece si las pruebas han ido bien, y si no aparecerá en rojo. En la pestaña *Failures* podemos ver qué pruebas han fallado, y en *Test Hierarchy* podemos ver todas las pruebas que se han realizado, y los resultados para cada una. En el cuadro inferior nos aparecen los errores que se han producido en las pruebas erróneas.

Para ejecutar el *TestRunner* u otro ejecutor de pruebas, podemos también definirnos un método `main` en nuestra clase de prueba que lance el ejecutor, en nuestro caso:

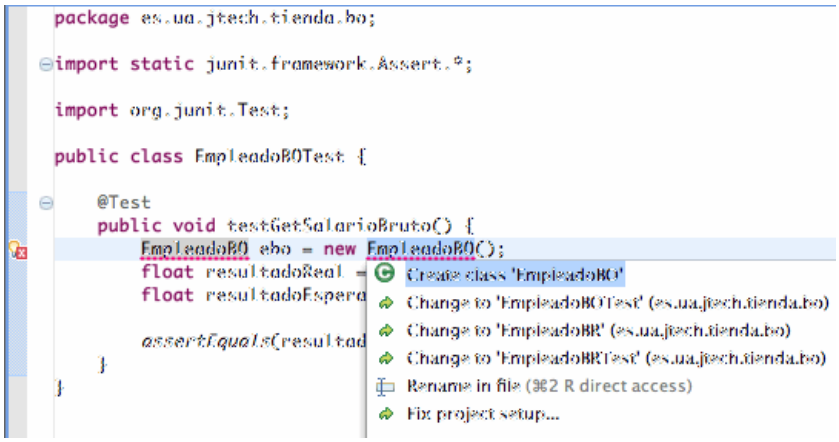
```
public static void main (String[] args)
{
        String[] nombresTest = {EmpleadoBRTest.class.getName()};
        junit.swingui.TestRunner.main(nombresTest);
}
```

Vemos que al `main` del *TestRunner* se le pueden pasar como parámetros los nombres de las clases de prueba que queremos probar.

7.3.4. Desarrollo guiado por pruebas

Anteriormente se ha comentado que puede resultar recomendable implementar las pruebas antes que el método a probar. Existe una metodología de desarrollo denominada desarrollo guiado por pruebas (*Test Driven Development, TDD*) que se basa precisamente en implementar primero las pruebas y a partir de ellas implementar las funcionalidades de nuestra aplicación. Esta metodología consiste en los siguientes pasos:

- Seleccionar una funcionalidad de la aplicación a implementar.
- Diseñar e implementar una serie de casos de prueba a partir de la especificación de dicha funcionalidad. Evidentemente, en un principio la prueba ni siquiera compilará, ya que no existirá ni la clase ni el método a probar. Con Eclipse podemos hacer que genere automáticamente los componentes que necesitamos. Podemos ver que al utilizar una clase que no existe en Eclipse nos mostrará un icono de error en la línea correspondiente. Pinchando sobre dicho icono podremos hacer que se cree automáticamente la clase necesaria:



Crear una clase a partir de la prueba

- Una vez compile, se deberá ejecutar la prueba para comprobar que falla. De esta forma nos aseguramos de que realmente las pruebas están comprobando algo.
- El siguiente paso consiste en hacer el mínimo código necesario para que se pasen los *tests*. Aquí se deberá implementar el código más sencillo posible, y no hacer nada que no sea necesario para pasar las pruebas, aunque nosotros veamos que pudiese ser conveniente. Si creemos que se debería añadir algo, la forma de proceder sería anotar la mejora, para más adelante añadirla a la especificación, y cuando las pruebas reflejen esa mejora se implementará, pero no antes. De esta forma nos aseguramos de que siempre todas las funcionalidades implementadas están siendo verificadas por alguna de las pruebas.
- Una vez las pruebas funcionan, refactorizaremos el código escrito para conseguir un código más limpio (por ejemplo para eliminar segmentos de código duplicados). Volviendo a ejecutar las pruebas escritas podremos comprobar que la refactorización no ha hecho que nada deje de funcionar.
- Por último, si todavía nos quedan funcionalidades por implementar, repetiremos el proceso seleccionando una nueva funcionalidad.

Estas pruebas también se denominan pruebas *red-green-refactor*, debido a que primero deberemos comprobar que las pruebas fallan (luz roja), y de esta forma tener confianza en que nuestras pruebas están comprobando algo. Después implementamos el código para conseguir que el test funcione (luz verde), y por último refactorizamos el código.

Una de las ventajas de esta tecnología es que se consigue un código de gran calidad, en el que vamos a tener una gran confianza, ya que va a estar probado desde el primer momento. Además, nos asegura que las pruebas verifican todos los requerimientos de la aplicación. Al eliminar los errores de forma temprana, con esta metodología se evita tener que depurar un código más complejo.

7.4. Fixtures

Es probable que en varias de las pruebas implementadas se utilicen los mismos datos de

entrada o de salida esperada, o que se requieran los mismos recursos. Para evitar tener código repetido en los diferentes métodos de *test*, podemos utilizar los llamados *fixtures*, que son elementos fijos que se crearán antes de ejecutar cada prueba. Para inicializar estos elementos fijos utilizaremos métodos marcados con las siguientes anotaciones:

Anotación	*Comportamiento*
`@Before`	El método se ejecutará antes de cada prueba (antes de ejecutar cada uno de los métodos marcados con `@Test`). Será útil para inicializar los datos de entrada y de salida esperada que se vayan a utilizar en las pruebas.
`@After`	Se ejecuta después de cada *test*. Nos servirá para liberar recursos que se hubiesen inicializado en el método marcado con `@Before`.
`@BeforeClass`	Se ejecuta una sola vez antes de ejecutar todos los *tests* de la clase. Se utilizarán para crear estructuras de datos y componentes que vayan a ser necesarios para todas las pruebas. Los métodos marcados con esta anotación deben ser estáticos.
`@AfterClass`	Se ejecuta una única vez después de todos los *tests* de la clase. Nos servirá para liberar los recursos inicializados en el método marcado con `@BeforeClass`, y al igual que este último, sólo se puede aplicar a métodos estáticos.

Imaginemos que tenemos una clase `ColaMensajes` a la que se pueden añadir una serie de mensajes de texto hasta llegar a un límite de capacidad. Cuando se rebase dicho límite, el mensaje más antiguo será eliminado. Para probar los métodos de esta clase en muchos casos nos interesará tener como entrada una cola llena. Para evitar repetir el código en el que se inicializa dicha cola, podemos hacer uso de *fixtures*:

```
public class ColaMensajesTest {

  ColaMensajes colaLlena3;

  @Before
  public void setUp() throws Exception {
    colaLlena3 = new ColaMensajes(3);
    colaLlena3.insertarMensaje("1");
    colaLlena3.insertarMensaje("2");
    colaLlena3.insertarMensaje("3");
  }

  @Test
  public void testInsertarMensaje() {
    List<String> listaEsperada = new ArrayList<String>();
    listaEsperada.add("2");
    listaEsperada.add("3");
    listaEsperada.add("4");
```

```
    colaLlena3.insertarMensaje("4");
    assertEquals(listaEsperada, colaLlena3.obtenerMensajes());
  }

  @Test
  public void testNumMensajes() {
    assertEquals(3, colaLlena3.numMensajes());
  }

  @Test
  public void testExtraerMensaje() {
    assertEquals("1", colaLlena3.extraerMensaje());
  }
}
```

7.5. Objetos mock

Hasta ahora hemos visto ejemplos muy sencillos, en los que el método a probar recibe todos los datos de entrada necesarios mediante parámetros. Sin embargo, en una aplicación real en la mayoría de los casos el comportamiento de los métodos no dependerá únicamente de los parámetros de entrada, sino que también dependerá de otros datos, como por ejemplo datos almacenados en una base de datos, o datos a los que se accede a través de la red. Estos datos también son una entrada del método de probar, pues el resultado del mismo depende de ellos, por lo que en nuestras pruebas de JUnit deberíamos ser capaces de fijarlos para poder predecir de forma determinista el resultado del método a probar.

Sin embargo, dado que muchas veces dependen de factores externos al código de nuestra aplicación, es imposible establecer su valor en el código de JUnit. Por ejemplo, imaginemos que el método `calculaSalarioNeto` visto como ejemplo anteriormente, dado que los tramos de las retenciones varían cada año, en lugar de utilizar unos valores fijos se conecta a una aplicación de la Agencia Tributaria a través de Internet para obtener los tramos actuales. En ese caso el resultado que devolverá dependerá de la información almacenada en un servidor remoto que no controlamos. Es más, imaginemos que la especificación del método nos dice que nos devolverá `BRException` si no puede conectar al servidor para obtener la información. Deberíamos implementar dicho caso de prueba, pero en principio nos es imposible especificar como entrada en JUnit que se produzca un fallo en la red. Tampoco nos vale cortar la red manualmente, ya que nos interesa tener una batería de pruebas automatizadas.

La solución para este problema es utilizar objetos *mock*. Éstos son objetos "impostores" que implementaremos nosotros y que se harán pasar por componentes utilizados en nuestra aplicación, permitiéndonos establecer su comportamiento según nuestros intereses. Por ejemplo, supongamos que el método `calculaSalarioNeto` está accediendo al servidor remoto mediante un objeto `ProxyAeat` que es quien se encarga de conectarse al servidor remoto y obtener la información necesaria de él. Podríamos crearnos un objeto `MockProxyAeat`, que se hiciese pasar por el objeto original, pero que nos permitiese establecer el resultado que queremos que nos devuelva, e incluso si queremos que

produzca alguna excepción. A continuación mostramos el código que tendría el método a probar dentro de la clase `EmpleadoBR`:

```
public float calculaSalarioNeto(float salarioBruto) {
  float retencion = 0.0f;

  if(salarioBruto < 0) {
    throw new BRException("El salario bruto debe ser positivo");
  }

  ProxyAeat proxy = getProxyAeat();
  List<TramoRetencion> tramos;
  try {
    tramos = proxy.getTramosRetencion();
  } catch (IOException e) {
    throw new BRException(
                  "Error al conectar al servidor de la AEAT", e);
  }

  for(TramoRetencion tr: tramos) {
    if(salarioBruto < tr.getLimiteSalario()) {
      retencion = tr.getRetencion();
      break;
    }
  }

  return salarioBruto * (1 - retencion);
}

ProxyAeat getProxyAeat() {
  ProxyAeat proxy = new ProxyAeat();
  return proxy;
}
```

Ahora necesitamos crear un objeto `MockProxyAeat` que pueda hacerse pasar por el objeto original. Para ello haremos que `MockProxyAeat` herede de `ProxyAeat`, sobrescribiendo los métodos para los que queramos cambiar el comportamiento, y añadiendo los constructores y métodos auxiliares que necesitemos. Debido al polimorfismo, este nuevo objeto podrá utilizarse en todos los lugares en los que se utilizaba el objeto original:

```
public class MockProxyAeat extends ProxyAeat {

  boolean lanzarExcepcion;

  public MockProxyAeat(boolean lanzarExcepcion) {
    this.lanzarExcepcion = lanzarExcepcion;
  }

  @Override
  public List<TramoRetencion> getTramosRetencion()
                                              throws IOException {
    if(lanzarExcepcion) {
      throw new IOException("Error al conectar al servidor");
    }

    List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
    tramos.add(new TramoRetencion(1000.0f, 0.0f));
    tramos.add(new TramoRetencion(1500.0f, 0.16f));
    tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));

    return tramos;
  }
```

```
}
```

Ahora debemos conseguir que dentro del método a probar se utilice el objeto *mock* en lugar del auténtico, pero deberíamos hacerlo sin modificar ni el método ni la clase a probar. Podremos hacer esto de forma sencilla si hemos utilizado métodos de *factoría* para tener acceso a estos componentes. Podemos crear una subclase de `EmpleadoBR` en la que se sobrescriba el método de factoría que se encarga de obtener el objeto `ProxyAeat`, para que en su lugar nos instancie el *mock*:

```
class TestableEmpleadoBR extends EmpleadoBR {

  ProxyAeat proxy;

  public void setProxyAeat(ProxyAeat proxy) {
    this.proxy = proxy;
  }

  @Override
  ProxyAeat getProxyAeat() {
    return proxy;
  }
}
```

Si nuestra clase a probar no tuviese un método de factoría, siempre podríamos refactorizarla para extraer la creación del componente que queramos sustituir a un método independiente y así permitir introducir el *mock* de forma limpia.

Nota

Tanto los objetos *mock* como cualquier clase auxiliar que hayamos creado para las pruebas, deberá estar contenida en el directorio de código de pruebas (`test`). En el directorio de código de la aplicación (`src`) sólo deberán quedar los componentes que sean necesarios para que la aplicación funcione cuando sea puesta en producción.

El código de JUnit para probar nuestro método podría quedar como se muestra a continuación:

```
TestableEmpleadoBR ebr;
TestableEmpleadoBR ebrFail;

@Before
public void setUpClass() {
  ebr = new TestableEmpleadoBR();
  ebr.setProxyAeat(new MockProxyAeat(false));

  ebrFail = new TestableEmpleadoBR();
  ebrFail.setProxyAeat(new MockProxyAeat(true));
}

@Test
public void testCalculaSalarioNeto1() {
  float resultadoReal = ebr.calculaSalarioNeto(2000.0f);
  float resultadoEsperado = 1640.0f;
  assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=BRException.class)
```

```
public void testCalculaSalarioNeto10() {
  ebrFail.calculaSalarioNeto(1000.0f);
}
```

Podremos utilizar *mocks* para cualquier otro tipo de componente del que dependa nuestro método. Por ejemplo, si en nuestro método se utiliza un generador de números aleatorios, y el comportamiento varía según el número obtenido, podríamos sustituir dicho generador por un *mock*, para así poder predecir en nuestro código el resultado que dará. De especial interés son las pruebas de métodos que dependen de los datos almacenados en una base de datos, ya que son los que nos encontraremos con más frecuencia, y en los que los *mocks* también nos pueden resultar de ayuda. Por este motivo, pasamos a tratarlos de forma especial a continuación.

7.5.1. Pruebas con base de datos

Normalmente en nuestra aplicación tendremos una serie de objetos que se encargan del acceso a datos (*Data Access Objects*, DAO). El resto de componentes de la aplicación, como pueden ser nuestros componentes de negocio, utilizarán los DAO para acceder a los datos. Por lo tanto, dichos componentes de negocio podrían ser probados sustituyendo los DAO por objetos *mock* en los que podamos establecer de forma sencilla el estado de la base de datos que queremos simular en las pruebas.

Supongamos que tenemos una clase `EmpleadoBO` que contiene un método `getSalarioBruto(int idEmpleado)`. Dicho método toma como entrada el identificador de un empleado en la base de datos, y nos devolverá su salario bruto. En caso de que el empleado no exista en la base de datos, lanzará una excepción de tipo `BOException`. Lo mismo ocurrirá si no puede acceder a la base de datos. La implementación será como se muestra a continuación:

```
public class EmpleadoBO {
  public float getSalarioBruto(int idEmpleado) throws BOException {

    IEmpleadoDAO edao = getEmpleadoDAO();
    EmpleadoBR ebr = new EmpleadoBR();

    try {
      EmpleadoTO empleado = edao.getEmpleado(idEmpleado);
      if(empleado==null) {
        throw new BOException("El usuario no existe");
      }

      return ebr.calculaSalarioBruto(empleado.getTipo(),
              empleado.getVentasMes(), empleado.getHorasExtra());
    } catch (DAOException e) {
      throw new BOException("Error al obtener el salario bruto",
                            e);
    } catch(BRException e) {
      throw new BOException("Error al calcular el salario bruto",
                            e);
    }
  }

  IEmpleadoDAO getEmpleadoDAO() {
    IEmpleadoDAO edao = new JDBCEmpleadoDAO();
```

```
    return edao;
  }
}
```

En este caso será sencillo crear un *mock* de nuestro DAO de empleados y sustuir el DAO original por el impostor. Nos será de ayuda el haber definido una interfaz para el DAO (`IEmpleadoDAO`), ya que de esta forma implementaremos el *mock* como otra versión del DAO que implemente la misma interfaz que la original:

```
public class MockEmpleadoDAO implements IEmpleadoDAO {

  List<EmpleadoTO> listaEmpleados;
  boolean falloConexion;

  public MockEmpleadoDAO(boolean falloConexion) {
    listaEmpleados = new ArrayList<EmpleadoTO>();
    this.falloConexion = falloConexion;
  }

  private void compruebaConexion() throws DAOException {
    if(falloConexion) {
      throw new DAOException("Fallo al conectar a la BD");
    }
  }

  public void addEmpleado(EmpleadoTO empleado)
                                          throws DAOException {
    this.compruebaConexion();
    if(this.getEmpleado(empleado.getId()) == null) {
      listaEmpleados.add(empleado);
    } else {
      throw new DAOException("El empleado la existe en la BD");
    }
  }

  public void delEmpleado(int idEmpleado) throws DAOException {
    this.compruebaConexion();
    EmpleadoTO empleado = this.getEmpleado(idEmpleado);
    if(empleado != null) {
      listaEmpleados.remove(empleado);
    } else {
      throw new DAOException("El empleado no existe en la BD");
    }
  }

  public EmpleadoTO getEmpleado(int idEmpleado)
                                          throws DAOException {
    this.compruebaConexion();
    for(EmpleadoTO empleado: listaEmpleados) {
      if(empleado.getId() == idEmpleado) {
        return empleado;
      }
    }

    return null;
  }

  public List<EmpleadoTO> getEmpleados() throws DAOException {
    this.compruebaConexion();
    return listaEmpleados;
  }
}
```

De esta forma podríamos implementar algunas de nuestras pruebas en JUnit como sigue a continuación:

```
public class EmpleadoBOTest {

  TestableEmpleadoBO ebo;

  @Before
  public void setUp() {
    EmpleadoDAO edao = new MockEmpleadoDAO(false);
    try {
      edao.addEmpleado(new EmpleadoTO(1,"12345678X",
             "Paco Garcia", TipoEmpleado.vendedor, 1250, 8));
      edao.addEmpleado(new EmpleadoTO(2,"65645342B",
             "Maria Gomez", TipoEmpleado.encargado, 1600, 2));
      edao.addEmpleado(new EmpleadoTO(3,"45452343F",
             "Manolo Gutierrez", TipoEmpleado.vendedor, 800, 0));
    } catch (DAOException e) {
      e.printStackTrace();
    }

    ebo = new TestableEmpleadoBO();
    ebo.setEmpleadoDAO(edao);
  }

  @Test
  public void testGetSalarioBruto() {
    try {
      float resultadoReal = ebo.getSalarioBruto(3);
      float resultadoEsperado = 1000.0f;

      assertEquals(resultadoEsperado, resultadoReal, 0.01);
    } catch (BOException e) {
      fail("Lanzada excepcion no esperada BOException");
    }
  }
}

class TestableEmpleadoBO extends EmpleadoBO {

  EmpleadoDAO edao;

  public void setEmpleadoDAO(EmpleadoDAO edao) {
    this.edao = edao;
  }

  @Override
  EmpleadoDAO getEmpleadoDAO() {
    return edao;
  }
}
```

Destacamos aquí que los *fixtures* han sido de gran ayuda para crear los objetos *mock* necesarios.

Sin embargo, si lo que nos interesa es probar nuestro DAO, ya no tiene sentido utilizar *mocks*, ya que estos estarían sustituyendo a las propias clases a probar. En estos casos será necesario probar la base de datos real. Para ello tenemos herramientas especializadas como DBUnit, que nos permiten establecer de forma sencilla el estado en el que queremos dejar la base de datos antes de cada prueba.

Otra opción sería utilizar los propios *fixtures* de JUnit para establecer el estado de la base de datos antes de cada prueba. Podríamos utilizar el método marcado con `@Before` para vaciar la tabla que estemos probando e insertar en ella los datos de prueba.

Importante

Nunca deberemos modificar ni una línea de código del método a probar, ya que nuestro objetivo es comprobar que dicho método se comporta como es debido tal como está implementado en la aplicación. Sólo podremos cambiar componentes de los que depende, pero sin afectar al código de propio método probado.

7.6. Suites de pruebas

A medida que vamos acumulando métodos de prueba, nos podría interesar organizarlos o agruparlos de determinada forma. Mediante las **suites** podemos asignar métodos de prueba a grupos, y así poder ejecutar todas nuestras pruebas (o un grupo de ellas) de forma conjunta.

Para crear la `suite` con JUnit 4 simplemente deberemos crear una nueva clase Java, y anotarla como se muestra a continuación:

```
import org.junit.*;
import org.junit.runner.*;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses( { EmpleadoBRTest.class, EmpleadoBOTest.class })
public class MiSuite {}
```

La anotación `@RunWith` simplemente hace referencia a la clase de JUnit 4 que se encarga de ejecutar las suites. Por otro lado, la anotación `SuiteClasses` sirve para poner entre llaves, y separadas por comas, todas las clases de prueba que queramos incorporar a la suite. De esta forma, la clase, como se ve, puede quedar vacía perfectamente. Luego el compilador, al procesar las anotaciones, ya se encarga de construir la suite correspondiente con las clases de prueba. Utilizando este método, todas las clases de prueba que agrupemos en la suite deberán estar implementadas en JUnit 4.

Para ejecutar la suite, la ejecutamos (*Run As*) como un *JUnit Test*. Aparecerá la ventana de JUnit con los tests realizados, agrupados por las clases a las que pertenece cada uno.

Para crear suites de pruebas en Eclipse, podemos ir a *File* > *New* > *JUnit Test Suite*. Ahí le indicamos el nombre de la suite, y las clases de prueba que queremos incluir en ella. Sin embargo, esto sólo funciona cuando utilizamos JUnit 3. Existe una pequeña "laguna" aún en la adaptación de las suites a JUnit 4, que Eclipse no ha cubierto aún. Sin embargo, existe una forma de hacer que las pruebas hechas en JUnit 4 funcionen en una suite creada por Eclipse.

Deberemos modificar el método `suite` creado automáticamente, añadiendo una nueva

clase, que compatibiliza las pruebas de JUnit 4 para que funcionen en entornos de ejecución de JUnit 3 (como el *runner* que lanza el método `suite`). Esta clase se llama **`JUnit4TestAdapter`**, y bastaría con sustituir las dos líneas anteriores en negrita por estas dos:

```
public static Test suite() {
  ...
  suite.addTest(
    new junit.framework.JUnit4TestAdapter(EmpleadoBOTest.class));
  suite.addTest(
    new junit.framework.JUnit4TestAdapter(EmpleadoBRTest.class));
  ...
}
```

Esta forma de crear la suite tiene la ventaja de que nos permitirá agrupar pruebas de JUnit 3 y 4.

8. Ejercicios de JUnit

8.1. Pruebas del gestor de filmotecas (2 puntos)

Vamos a implementar un conjunto de casos de prueba para probar `IPeliculaDAO`. En nuestro caso, la implementación que someteremos a nuestras pruebas será `MemoryPeliculaDAO`. Vamos a implementar en JUnit los siguientes casos de prueba:

Método a probar	Entrada	Salida esperada
`addPelicula( Pelicula p)`	Lista actual: vacía `p.id`: 1 `p.titulo`: "El Resplandor"	Lista: {1, "El Resplandor"}
`addPelicula( Pelicula p)`	Lista actual: {1, "El Resplandor"} `p.id`: 3 `p.titulo`: "Casablanca"	Lista: {1, "El Resplandor"}, {3, "Casablanca"}
`addPelicula( Pelicula p)`	Lista actual: {1, "El Resplandor"} `p.id`: 1 `p.titulo`: "El Resplandor"	`DAOException`
`addPelicula( Pelicula p)`	Lista actual: {3, "Casablanca"} `p.id`: 4 `p.titulo`: null	`DAOException`
`addPelicula( Pelicula p)`	Lista actual: {3, "Casablanca"} `p.id`: 5 `p.titulo`: ""	`DAOException`
`addPelicula( Pelicula p)`	Lista actual: {3, "Casablanca"} p: null	`DAOException`
`delPelicula(int i)`	Lista actual: vacía `i`: 1	`DAOException`
`delPelicula(int i)`	Lista actual: {1, "El Resplandor"} `i`: 1	Lista: vacía
`delPelicula(int i)`	Lista actual: {3, "Casablanca"} `i`: 1	`DAOException`
`delPelicula(int i)`	Lista actual: {1, "El Resplandor"}, {3, "Casablanca"} `i`: 1	Lista: {3, "Casablanca"}
`delPelicula(int i)`	Lista actual: {1, "El	`DAOException`

	Resplandor"}, {3, "Casablanca"} `i`: 2	

Utiliza *fixtures* para inicializar los distintos tipos de entradas y salidas esperadas que vamos a probar. Fíjate en que en muchas de las pruebas se repiten las estructuras de datos que se tienen como entrada y que se esperan como salida. Podemos aprovechar esto para simplificar nuestro código, dejando dichas estructuras predefinidas como *fixtures*.

Podría ser de ayuda no probar directamente la clase `MemoryPeliculaDAO`, sino crear una subclase a la que añadamos un método que nos permita establecer el contenido de la lista actual de películas. Esta clase siempre deberá estar en el directorio de fuentes de pruebas, ya que será una clase que sólo servirá para realizar estas pruebas, y en ella nunca deberemos sobrescribir los métodos a probar, ya que lo que nos interesa es probar los métodos reales, no los que creemos como ayuda para las pruebas.

8.2. Desarrollo guiado por pruebas (1 punto)

Vamos a utilizar la metodología de desarrollo guiado por pruebas para implementar las operaciones `calculaSalarioBruto` y `calculaSalarioNeto` especificadas en el tema de teoría. Tenemos en las plantillas de la sesión un proyecto `lja-junit` en el que podemos encontrar la aplicación parcialmente implementada. Tenemos una clase `EmpleadoBR` con el esqueleto de los métodos anteriores, que deberán ser implementados. Se pide:

a) Tenemos implementada una clase `EmpleadoBO` con las operaciones de negocio de los empleados. En este caso únicamente tenemos el método `getSalarioBruto` que nos proporciona el salario bruto de un empleado dado su identificador. Para ello primero utiliza el DAO para obtener los datos del empleado de la base de datos, y después utiliza las reglas de negocio de los empleados, contenidas en `EmpleadoBR`, para calcular el salario a partir de dichos datos. Tenemos también definida una prueba para esta clase en `EmpleadoBOTest`. El DAO JDBC para acceso a los datos de los empleados en una base de datos no está implementado, pero tenemos un *mock* que lo sustituye en las pruebas (`MockEmpleadoDAO`), así que eso no supone ningún problema para poder probar el método. Sin embargo, nos falta implementar la regla de negocio necesaria, y como es evidente, si ejecutamos la prueba veremos que falla.

b) Implementar los casos de prueba especificados en el tema de teoría para la operación `calculaSalarioBruto`. Ejecutar dichos casos de prueba y comprobar que el test falla.

c) Implementar la operación `calculaSalarioBruto` con el código más sencillo posible que consiga que las pruebas tengan éxito. Tras implementarlo, comprobar que JUnit nos da luz verde. Si fuese necesario, refactorizar el código para dejarlo limpio y bien organizado, y comprobar que las pruebas sigan funcionando.

d) Ahora podríamos ejecutar las pruebas de `EmpleadoBO`. Al estar implementada la regla de negocio necesaria debería funcionar.

e) Repetir el proceso para la operación `calculaSalarioNeto`. Para obtener la información de los tramos de retención utilizaremos la clase `ProxyAeat`. El acceso al servidor no está implementado todavía en esta clase, pero en lugar de implementarlo lo que haremos será crear un *mock* para poder ejecutar las pruebas. En el tema de teoría puedes encontrar ayuda para la implementación de este método y del *mock* necesario para las pruebas.

f) Implementar una suite de pruebas que agrupe tanto las pruebas de `EmpleadoBRTest` como las de `EmpleadoBOTest`. La suite se llamará `AllTests`, y se encontrará en el paquete `es.ua.jtech.lja.tienda`.

9. Serialización de datos

9.1. Introducción

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. Si queremos transferir estructuras de datos complejas, deberemos convertir estas estructuras en secuencias de bytes que puedan ser enviadas a través de un flujo. Esto es lo que se conoce como serialización. Comenzaremos viendo los fundamentos de los flujos de entrada y salida en Java, para a continuación pasar a estudiar los flujos que nos permitirán serializar diferentes tipos de datos Java de forma sencilla.

9.2. 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
Caractéres	`_Reader`	`_Writer`
Bytes	`_InputStream`	`_OutputStream`

Donde el prefijo se referirá a la fuente o sumidero de los datos que puede tomar valores como los que se muestran a continuación:

`File_`	Acceso a ficheros
`Piped_`	Comunicación entre programas mediante tuberías (*pipes*)
`String_`	Acceso a una cadena en memoria (solo caracteres)
`CharArray_`	Acceso a un *array* de caracteres en memoria (solo caracteres)

`ByteArray_`	Acceso a un *array* de *bytes* en memoria (solo *bytes*)

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

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún procesamiento con ellos. Tenemos por ejemplo flujos que realizan un filtrado de los datos que viajan a través de ellos (con prefijo `Filter`), conversores datos (con prefijo `Data`), *bufferes* de datos (con prefijo `Buffered`), preparados para la impresión de elementos (con prefijo `Print`), etc.

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

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

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

A parte 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.

9.3. 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` para escribir *bytes*, dos métodos más: `print` y `println`. 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.

Truco

Podemos ahorrar tiempo si en Eclipse en lugar de escribir `System.out.println` escribimos simplemente `sysout` y tras esto pulsamos *Ctrl + Espacio*.

9.4. 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();
  }
}
```

9.5. Acceso a los recursos

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.

9.6. Acceso a la red

Podemos también obtener flujos para leer datos a través de la red a partir de una URL. De esta forma podremos obtener por ejemplo información ofrecida por una aplicación web. Lo primero que debemos hacer es crear un objeto `URL` especificando la dirección a la que queremos acceder:

```
URL url = new URL("http://www.ua.es/es/index.html");
```

A partir de esta URL podemos obtener directamente un flujo de entrada mediante el método `openStream`:

```
InputStream in = url.openStream();
```

Una vez obtenido este flujo de entrada podremos leer de él o bien transformarlo a otro tipo de flujo como por ejemplo a un flujo de caracteres o de procesamiento. La lectura se hará de la misma forma que cualquier otro tipo de flujo.

9.7. Codificación de datos

Si queremos guardar datos en un fichero binario, enviarlos a través de la red, o en general transferirlos mediante cualquier flujo de E/S, deberemos codificar estos datos en forma de *array* de *bytes*. Los flujos de procesamiento `DataInputStream` y `DataOutputStream` nos permitirán codificar y descodificar respectivamente los tipos de datos simples en forma de *array* de *bytes* para ser enviados a través de un flujo de datos.

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

```
String nombre = "Jose";
String edad = 25;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

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

```
baos.close();

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

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

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);
String nombre = dis.readUTF();
int edad = dis.readInt();
```

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

9.8. Serialización de objetos

Si queremos enviar un objeto 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 directamente.

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject` y `writeObject` 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*.

Un uso común de la serialización se realiza en los *Transfer Objetcs*. Este tipo de objetos deben ser serializables para así poderse intercambiar entre todas las capas de la aplicación, aunque se encuentren en máquinas diferentes.

Por ejemplo, si tenemos un objeto como el siguiente:

```
public class Punto2D implements Serializable {
        private int x;
        private int y;

        public int getX() {
                return x;
        }
        public void setX(int x) {
                this.x = x;
        }
```

```
        public int getY() {
                return y;
        }
        public void setY(int y) {
                this.y = y;
        }
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

```
Punto2D p = crearPunto();
FileOutputStream fos = new FileOutputStream(FICHERO_DATOS);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos *serializables* nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Muchas clases de la API de Java son *serializables*, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a `writeObject`.

Cuando una clase implemente la interfaz `Serializable` veremos que Eclipse nos da un *warning* si no añadimos un campo `serialVersionUID`. Este es un código numérico que se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso que una de ellas esté en una máquina cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.

Eclipse nos ofrece dos formas de generar este código pulsando sobre el icono del *warning*: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utiliar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.

10. Ejercicios de Serialización

10.1. Leer un fichero de texto (0.5 puntos)

Vamos a realizar un programa que lea un fichero de texto ASCII, y lo vaya mostrando por pantalla. El esqueleto del programa se encuentra en el fichero `Ej1.java` dentro del proyecto `lja-serializacion` de las plantillas de la sesión. Se pide:

a) ¿Qué tipo de flujo de datos utilizaremos para leer el fichero? Añadir al programa la creación del flujo de datos adecuado (variable `in`), compilar y comprobar su correcto funcionamiento.

b) Podemos utilizar un flujo de procesamiento llamado `BufferedReader` que mantendrá un *buffer* de los caracteres leídos y nos permitirá leer el fichero línea a línea además de utilizar los métodos de lectura a más bajo nivel que estamos usando en el ejemplo. Consultar la documentación de la clase `BufferedReader` y aplicar la transformación sobre el flujo de entrada (ahora `in` deberá ser un objeto `BufferedReader`). Compilar y comprobar que sigue funcionando correctamente el método de lectura implementado.

c) Ahora vamos a cambiar la forma de leer el fichero y lo vamos a hacer línea a línea aprovechando el `BufferedReader`. ¿Qué método de este objeto nos permite leer líneas de la entrada? ¿Qué nos devolverá este método cuando se haya llegado al final el fichero? Implementar un bucle que vaya leyendo estas líneas y las vaya imprimiendo, hasta llegar al final del fichero. Compilar y comprobar que sigue funcionando de la misma forma.

10.2. Lectura de una URL (0.5 puntos)

El fichero `Ej2.java` es un programa que tomará una URL como parámetro, accederá a ella y leerá su contenido mostrándolo por pantalla. Deberemos añadir código para:

a) Crear un objeto URL para la `url` especificada en el método `creaURL`, capturando las posibles excepciones que se pueden producir si está mal formada y mostrando el mensaje de error correspondiente por la salida de error. Compilar y comprobar que ocurre al pasar URLs correctas e incorrectas.

b) Abrir un flujo de entrada desde la URL indicada en el método `leeURL`. Deberemos obtener un `InputStream`de la URL, y convertirlo a un objeto `BufferedReader`, aplicando las transformaciones intermedias necesarias, para poder leer de la URL los caracteres línea a línea. Comprobar que lee correctamente algunas URLs conocidas. Descomentar el bloque de código que realiza la lectura de la URL.

10.3. Gestión de productos (1 punto)

Vamos a hacer una aplicación para gestionar una lista de productos que vende nuestra empresa. Escribiremos la información de estos productos en un fichero, para almacenarlos de forma persistente. Se pide:

a) Introducir el código necesario en el método `almacenar` de la clase `GestorProductos` para guardar la información de los productos en el fichero definido en la constante `FICHERO_DATOS`. Guardaremos esta información codificada en un fichero binario. Deberemos codificar los datos de cada producto (titulo, autor, precio y disponibilidad) utilizando un objeto `DataOutputStream`.

b) Introducir en el método `recuperar` el código para cargar la información de este fichero. Para hacer esto deberemos realizar el procedimiento inverso, utilizando un objeto `DataInputStream` para leer los datos de los productos almacenados. Leeremos productos hasta llegar al final del fichero, cuando esto ocurra se producirá una excepción del tipo `EOFException` que podremos utilizar como criterio de parada.

c) Modificar el código anterior para, en lugar de codificar manualmente los datos en el fichero, utilizar la serialización de objetos para almacenar y recuperar objetos `ProductoTO` del fichero.

10.4. Guardar datos de la filmoteca (1 punto)

Vamos a implementar la clase `FilePeliculaDAO` de nuestra aplicación de gestión de filmotecas. De esta forma, nuestra lista de películas quedará guardada de forma persistente. En el fichero guardaremos directamente una lista de objetos `PeliculaTO` serializados. Se pide:

a) Implementar la operación de consulta del listado de películas. En este caso, si no existiese todavía el fichero devolveríamos una lista de películas vacía.

Nota

Siempre que las operaciones de acceso a ficheros puedan lanzar alguna excepción *checked*, deberemos capturarla y lanzarla como *nested exception* dentro de una excepción de tipo `DAOException`.

b) Implementar las operaciones para añadir y eliminar películas. En estas operaciones primero obtendremos el listado de películas actuales del fichero, realizaremos con esta lista la operación correspondiente, y volveremos a guardar la lista resultante en el fichero. La especificación de estos métodos será la misma que en el caso de `MemoryPeliculaDAO`, pero en este caso en lugar de tener la lista en memoria, tendremos que recuperarla del fichero, realizar la operación correspondiente con ella, y volverla a guardar.

Nota

Utiliza los bloques `finally` para asegurarnos de que los ficheros se cierren.

c) ¿Hay algún bloque de código común en las operaciones de añadir y eliminar películas? Si tienes código repetido utiliza las opciones de refactorización de Eclipse para extraer el código duplicado a un nuevo método y de esta forma evitar este problema.

11. Depuración y gestión de logs

En esta sesión veremos cómo podemos depurar nuestro código al ejecutarlo, mediante el depurador incorporado en Eclipse. Para las aplicaciones donde no sea posible su auto-ejecución (por ejemplo, aplicaciones basadas en web), debemos recurrir a otras técnicas de depurado, como son la emisión de mensajes de estado (logs), que indiquen qué va haciendo el programa en cada momento. Para esto veremos cómo utilizar una librería llamada **Log4Java**, y las posibilidades que ofrece.

11.1. Depuración con Eclipse

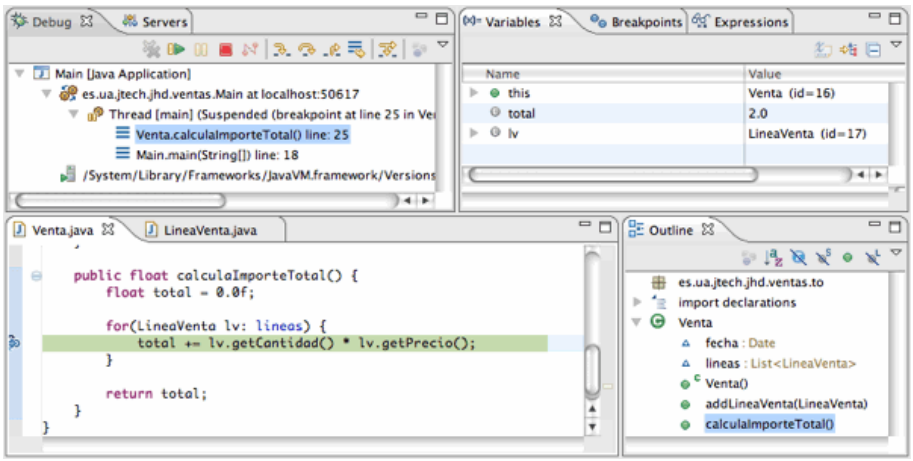
En este apartado veremos cómo podemos depurar el código de nuestras aplicaciones desde el depurador que incorpora Eclipse, encontrando el origen de los errores que provoque nuestro programa. Este depurador incorpora muchas funcionalidades, como la ejecución paso a paso, establecimiento de *breakpoints*, consulta y establecimiento de valores de variables, parar y reanudar hilos de ejecución, etc. También podremos depurar aplicaciones que se estén ejecutando en máquinas remotas, e incluso utilizar el depurador a la hora de trabajar con otros lenguajes como C o C++ (instalando las *C/C++ Development Tools (CDT)*).

Eclipse proporciona una vista de depuración (*debug view*) que permite controlar el depurado y ejecución de programas. En él se muestra la pila de los procesos e hilos que tengamos ejecutando en cada momento.

11.1.1. Primeros pasos para depurar un proyecto

En primer lugar, debemos tener nuestro proyecto hecho y correctamente compilado. Una vez hecho eso, ejecutaremos la aplicación, pero en lugar de hacerlo desde el menú *Run As* lo haremos desde el menú *Debug As* (pinchando sobre el botón derecho sobre la clase que queramos ejecutar/depurar).

En algunas versiones de Eclipse, al depurar el código pasamos directamente a la perspectiva de depuración (*Debug perspective*), pero para cambiar manualmente, vamos a **Window - Open perspective - Debug**.



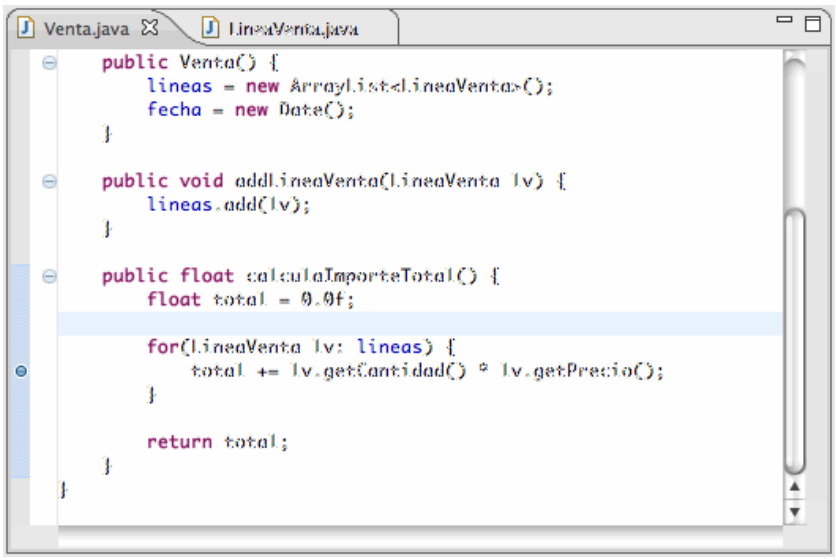
Perspectiva de depuración

En la parte superior izquierda vemos los hilos que se ejecutan, y su estado. Arriba a la derecha vemos los *breakpoints* que establezcamos, y los valores de las variables que entran en juego. Después tenemos el código fuente, para poder establecer marcas y *breakpoints* en él, y debajo la ventana con la salida del proyecto.

11.1.2. Establecer breakpoints

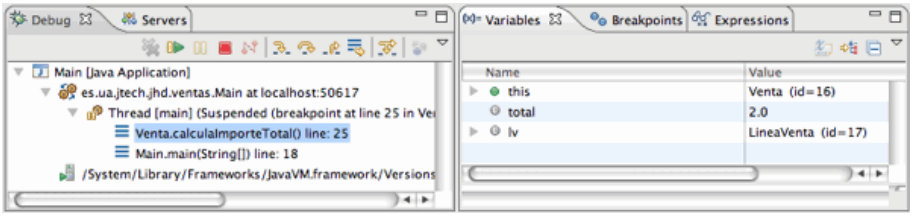
Una de las operaciones más habituales a la hora de depurar es establecer *breakpoints*, puntos en los que la ejecución del programa se detiene para permitir al programador examinar cuál es el estado del mismo en ese momento.

Para establecer *breakpoints*, vamos en la ventana de código hasta la línea que queramos marcar, y hacemos doble click en el margen izquierdo:



Establecer breakpoints

El *breakpoint* se añadirá a la lista de *breakpoints* de la pestaña superior derecha. Una vez hecho esto, re-arrancamos el programa desde **Run - Debug**, seleccionando la configuración deseada y pulsando en **Debug**.

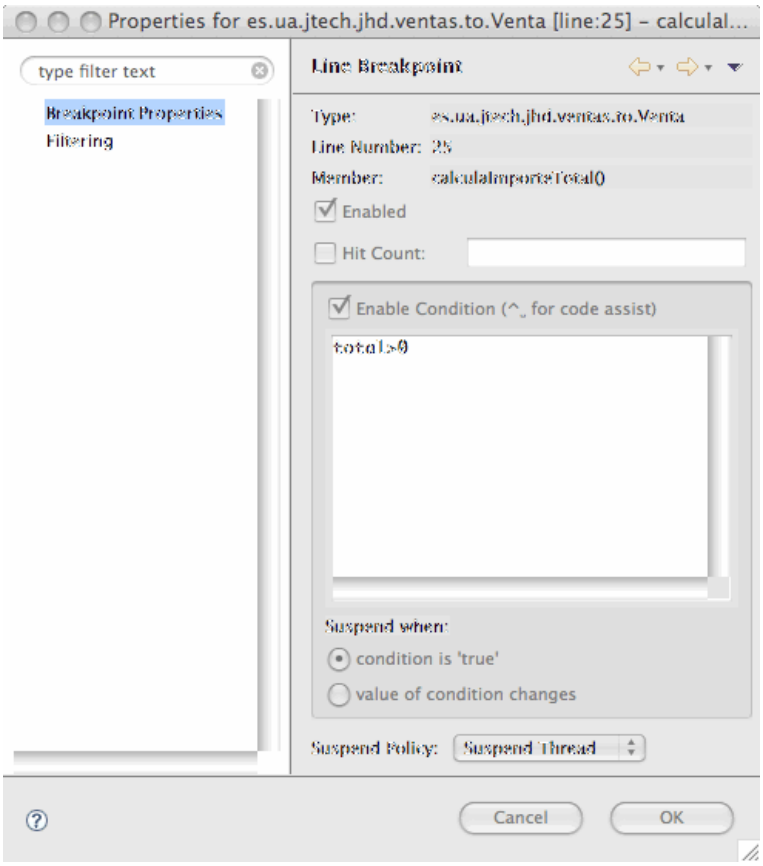


Control del estado del programa

En la parte superior podemos ver el estado de las variables (pestaña *Variables*), y de los hilos de ejecución. Tras cada *breakpoint*, podemos reanudar el programa pulsando el botón de *Resume* (la flecha verde), en la parte superior.

Breakpoints condicionales

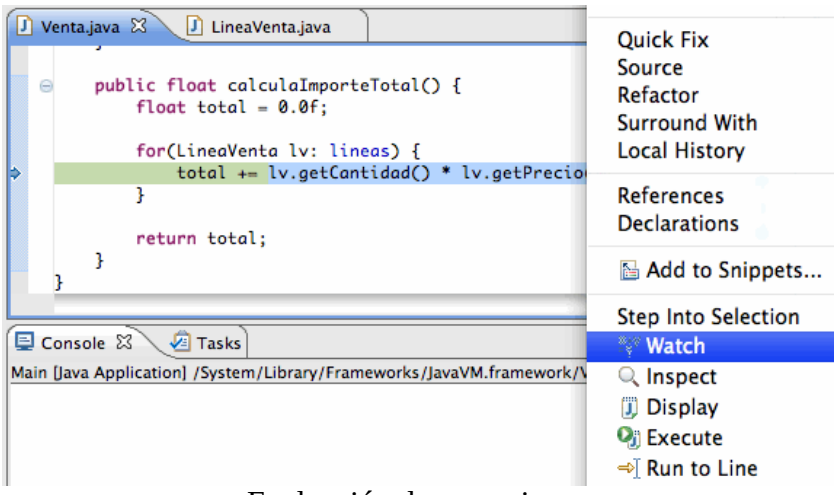
Podemos establecer también *breakpoints* condicionales, que se disparen únicamente cuando el valor de una determinada expresión o variable cambie. Para ello, pulsamos con el botón derecho sobre la marca del *breakpoint* en el código, y elegimos **Breakpoint Properties**. Allí veremos una casilla que indica **Enable Condition**. Basta con marcarla y poner la expresión que queremos verificar. Podremos hacer que se dispare el *breakpoint* cuando la condición sea cierta, o cuando el valor de esa condición cambie:



Breakpoints condicionales

11.1.3. Evaluar expresiones

Podemos evaluar una expresión del código si, durante la depuración, seleccionamos la expresión a evaluar, y con el botón derecho elegimos **Inspect**. Si queremos hacer un seguimiento del valor de la expresión durante la ejecución del programa, seleccionaremos **Watch** en lugar de **Inspect**. De esta forma el valor de la expresión se irá actualizando conforme ésta cambie.

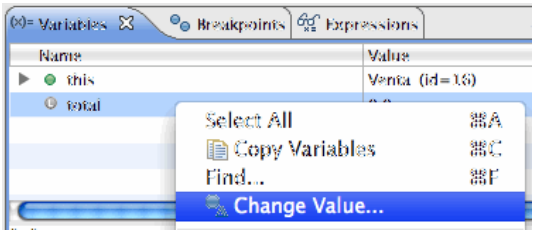


Evaluación de expresiones

11.1.4. Explorar variables

Como hemos dicho, en la parte superior derecha, en el cuadro **Variables** podemos ver el valor que tienen las variables en cada momento. Una vez que la depuración alcanza un *breakpoint*, podemos desde el menú **Run** ir a la opción **Step Over** (o pulsar **F6**), para ir ejecutando paso a paso a partir del *breakpoint*, y ver en cada paso qué variables cambian (se ponen en rojo), y qué valores toman.

También podemos, en el cuadro de variables, pulsar con el botón derecho sobre una, y elegir **Change Value**, para cambiar a mano su valor y ver cómo se comporta el programa.



Exploración de variables

11.1.5. Cambiar código "en caliente"

Si utilizamos Java 1.4 o superior, desde el depurador de Eclipse podemos, durante la depuración, cambiar el código de nuestra aplicación y seguir depurando. Basta con modificar el código durante una parada por un *breakpoint* o algo similar, y después pulsar en *Resume* para seguir ejecutando.

Esto se debe a que Java 1.4 es compatible con la JPDA (*Java Platform Debugger Architecture)*, que permite modificar código en una aplicación en ejecución. Esto es útil

cuando es muy pesado re-arrancar la aplicación, o llegar al punto donde falla.

11.2. Gestión de logs con Log4Java

Log4Java (log4j) es una librería *open source* que permite a los desarrolladores de software controlar la salida de los mensajes que generen sus aplicaciones, y hacia dónde direccionarlos, con una cierta granularidad. Es configurable en tiempo de ejecución, lo que permite establecer el tipo de mensajes que queremos mostrar y dónde mostrarlos, sin tener que detener ni recompilar nuestra aplicación.

Para encontrar más información acerca de Log4J, así como últimas versiones que vayan saliendo, consultad la página del proyecto Jakarta: **http://jakarta.apache.org/log4j/**.

11.2.1. Estructura de Log4J

El funcionamiento de Log4J se basa en 3 elementos básicos: *loggers*, *appenders* y *layouts*.

Los **loggers** son entidades asociadas a nombres (de paquetes o clases), que recogen los mensajes de log que emiten dichos paquetes o clases. Se dice que un *logger* es un **antecesor** de otro si su nombre, seguido de un punto, es un prefijo del nombre del otro log. Un *logger* es el **padre** de otro, si es su antecesor más cercano. Por ejemplo: el logger `com.foo` es padre del logger `com.foo.Bar`, y antecesor de `com.foo.otroPaquete.UnaClase`.

Existe siempre un logger raíz (*root logger*), a partir del cual se genera toda la jerarquía de nuestra aplicación.

Una vez controlado el mensaje de log, se debe mostrar por alguna parte. Ese *dónde* se envía el mensaje de log es lo que llamamos **appender**. Podremos tener como *appenders* la propia pantalla, un fichero de texto, o una base de datos, entre otros, como veremos a continuación.

Finalmente, podemos dar un formato a los mensajes de log, para que se muestren con la apariencia que queramos. Por ejemplo, mostrar la fecha actual, tipo de mensaje, y texto del mensaje. Todo esto se consigue definiendo el correspondiente **layout** de los logs.

Veremos cada uno de estos elementos con más detalle a continuación.

11.2.2. Niveles de log (loggers)

Una ventaja que debe tener una librería de logging sobre los tradicionales `System.out.println` o `printf` que nos ayudaban a ver por dónde iba el programa y qué pasaba, es que deben permitir omitir estos mensajes de forma cómoda, cuando no necesitemos que se muestren, sin tener que quitarlos a mano del código y recompilar la aplicación.

Los mensajes de log que maneja log4j se categorizan en 5 niveles de prioridad:

- `DEBUG`: para mensajes de depuración, mientras se está desarrollando la aplicación, para ver cómo se comporta.
- `INFO`: para mensajes que muestren información del programa durante su ejecución (por ejemplo, versión que se esté ejecutando, inicios o fines de procesos que se lancen, etc)
- `WARN`: para mensajes de alerta sobre situaciones anómalas que se produzcan, pero que no afecten el funcionamiento correcto del programa.
- `ERROR`: para guardar constancia de errores del programa que, aunque le permitan seguir funcionando, afecten al funcionamiento del mismo. Por ejemplo, que un parámetro de configuración tenga un valor incorrecto, o que no se encuentre un fichero no crítico.
- `FATAL`: se utiliza para mensajes críticos, por errores que hacen que el programa generalmente aborte su ejecución.

Se podrá habilitar que se muestren los mensajes pertenecientes a ciertas categorías de las anteriores, en cualquier momento, según la información que nos interese mantener.

Para mostrar mensajes de log en una clase, debemos crear un objeto de tipo `org.apache.log4j.Logger`, asignándole el nombre asociado (normalmente nos basamos en el nombre de la clase que lo usa), y luego tenemos en ella métodos para generar mensajes de cada uno de los 5 niveles:

```
debug (String mensaje)
info  (String mensaje)
warn  (String mensaje)
error (String mensaje)
fatal (String mensaje)
```

Por ejemplo:

```
import org.apache.log4j.*;

public class MiClase
{
        static Logger logger = Logger.getLogger(MiClase.class);
        ...
        public static void main(String[] args)
        {
                logger.info("Entrando en la aplicacion");
                ...
                logger.warn("Esto es una advertencia");
                ...
                logger.fatal("Error fatal");
        }
}
```

Además, existen dos niveles extras que sólo se utilizan en el fichero de configuración:

- `ALL`: el nivel más bajo posible, habilita todos los niveles anteriores
- `OFF`: el nivel más alto, deshabilita todos los niveles anteriores.

Veremos más adelante cómo configurar la habilitación o deshabilitación de los niveles

que interesen.

11.2.3. Salidas de log (appenders)

Además, log4j permite que un nivel de log se muestre por una o varias salidas establecidas. Cada una de las salidas por las que puede mostrarse un log se llama *appender*. Algunos de los disponibles (cuyas clases están en el paquete `org.apache.log4j`) son:

- `ConsoleAppender`: muestra el log en la pantalla. Tiene 3 opciones configurables:
  - `Threshold=WARN`: indica que no muestre ningún mensaje con nivel menor que el indicado
  - `ImmediateFlush=true`: si está a `true`, indica que los mensajes no se almacenan en un buffer, sino que se envían directamente al destino
  - `Target=System.err`: por defecto envía los mensajes a `System.out`. Establece la salida a utilizar
- `FileAppender`: vuelca el log a un fichero. Tiene 4 opciones configurables:
  - `Threshold=WARN`: sirve para lo mismo que en `ConsoleAppender`
  - `ImmediateFlush=true`: sirve para lo mismo que en `ConsoleAppender`
  - `File=logs.txt`: nombre del archivo donde guardar los logs
  - `Append=false`: a `true` indica que los nuevos mensajes se añadan al final del fichero, y a `false`, cuando se reinicie la aplicación el archivo se sobreescribirá.
- `RollingFileAppender`: como `FileAppender`, pero permite definir políticas de rotación en los ficheros, para que no crezcan indefinidamente. Tiene 6 opciones configurables:
  - `Threshold=WARN`: sirve para lo mismo que en `ConsoleAppender`
  - `ImmediateFlush=true`: sirve para lo mismo que en `ConsoleAppender`
  - `File=logs.txt`: sirve para lo mismo que en `FileAppender`
  - `Append=false`: sirve para lo mismo que en `FileAppender`
  - `MaxFileSize=100KB`: los sufijos pueden ser KB, MB o GB. Rota el archivo de log (copia su contenido en otro fichero auxiliar para vaciarse) cuando el log alcanza el tamaño indicado.
  - `MaxBackupIndex=2`: mantiene los archivos de respaldo auxiliares que se indiquen como máximo. Los archivos de respaldo sirven para no borrar los logs antiguos cuando lleguen nuevos, sino guardarlos en otros ficheros mientras quepan. Sin esta opción, el número de archivos de respaldo sería ilimitado.

  NOTA: existe una variante de este *appender*, que es `DailyRollingFileAppender`, que permite definir políticas de rotación basadas en fechas (políticas diarias, semanales, mensuales... etc).
- `JDBCAppender`: redirecciona los mensajes hacia una base de datos. Se espera que este appender sea modificado en un futuro. Sus opciones son:

  - `Threshold=WARN`: sirve para lo mismo que en `ConsoleAppender`
  - `ImmediateFlush=true`: sirve para lo mismo que en `ConsoleAppender`
  - `Driver=mm.mysql.Driver`: define el driver de conexión por JDBC
  - `URL=jdbc:mysql://localhost/LOG4JDemo`:indica la URL de la base de datos en el servidor de base de datos
  - `user=default`: login del usuario
  - `password=default`: password del usuario
  - `sql=INSERT INTO Logs(mensaje) VALUES ('%d - %c - %p - %m')`: sentencia SQL que se utiliza para dar de alta el mensaje en la BD.
- Otros *appenders*:
  - `SocketAppender`: redirecciona los mensajes hacia un servidor remoto
  - `SMTPAppender`: envía un email con los mensajes de log
  - `SyslogAppender`: redirecciona los mensajes hacia el demonio *syslog* de Unix
  - ... etc.

Los *appenders* se establecen en la configuración que se proporcione, que se verá más adelante.

11.2.4. Formato del log (layout)

Los *layouts* son los responsables de dar formato de salida a los mensajes de log, de acuerdo a las especificaciones que indique el desarrollador de software. Algunos de los tipos disponibles son:

- `SimpleLayout`: consiste en la prioridad del mensaje, seguida de un guión, y el mensaje:

```
DEBUG - Hola, esto es una prueba
```

- `PatternLayout`: establece el formato de salida del mensaje de acuerdo a unos patrones similares al comando `printf` de C. Algunos de los patrones disponibles son:
  - `c`: para desplegar la categoría del evento de log. Puede tener una precisión, dada entre llaves, que indique qué parte de la categoría mostrar. Por ejemplo: `%c`, `%c{2}`.
  - `C`: para desplegar el nombre completo de la clase que generó el evento de log. También puede tener una precisión entre llaves. Por ejemplo, si la clase en cuestión es `org.apache.MiClase`, si ponemos `%C` mostrará "`org.apache.MiClase`", pero si ponemos `%C{1}` mostrará "`MiClase`".
  - `d`: muestra la fecha en que se genera el evento de log. Entre llaves podemos indicar el formato de la misma. Por ejemplo: `d{dd/MM/yyyy HH:mm:ss}`
  - `L`: muestra el número de línea donde se ha generado el evento de log
  - `m`: muestra el mensaje del evento de log
  - `M`: muestra el nombre del método que generó el evento de log
  - `n`: genera un salto de línea en el formato
  - `p`: muestra la prioridad del evento de log

  - `r`: muestra los milisegundos desde que se inició la aplicación hasta que se produjo el error
  - ... existen otros muchos modificadores que no se comentan aquí, por simplificar.
- `HTMLLayout`: establece como salida de log una tabla HTML. Tiene como atributos:
  - `LocationInfo=true`: a `true` muestra la clase Java y el número de línea donde se produjo el mensaje
  - `Title=Título de mi aplicación`: indica el valor a incluir en el tag `<title>` de HTML
- `XMLLayout`: la salida será un archivo XML, que cumple con la DTD `log4j.dtd`. Tiene como atributo:
  - `LocationInfo=true`: a `true` muestra la clase Java y el número de línea donde se produjo el mensaje
- `TTCCLayout`: consiste en la fecha, hilo, categoría y NDC (Nested Diagnostic Context) del evento.

El formato de log también se establece en los ficheros de configuración, como se verá más adelante.

11.2.5. Configurar Log4J

Log4J permite configurarse mediante archivos, bien sean XML, o archivos de propiedades de java (donde cada línea tiene el aspecto `clave=valor`).

Vemos un ejemplo de cómo incluir logs en una aplicación. Los pasos a seguir son:

- Incluir los paquetes (`org.apache.log4j.*`)
- Definir una variable de tipo `Logger` con el nombre de la clase que queremos loggear
- Crear una configuración (por ejemplo, con `BasicConfigurator.configure(...)`)

11.2.5.1. Configuración básica de Log4J

Veamos cómo sería la configuración más sencilla de Log4J con un ejemplo:

```
import org.apache.log4j.*;

public class MiClase
{
        static Logger logger = Logger.getLogger(MiClase.class);
        ...
        public static void main(String[] args)
        {
                BasicConfigurator.configure();
                logger.info("Entrando en la aplicacion");
                MiOtraClase mc = new MiOtraClase();
                mc.hola();
                logger.info("Finalizando aplicacion");
        }
}
```

El log se configura en la clase principal de la aplicación (la que la arranca), y después todas las subclases ya parten de esa configuración para mostrar sus logs. Por ejemplo, en el caso anterior, si quisiéramos mostrar información de log desde la clase `MiOtraClase`, sólo tendríamos que declarar el `Logger` y mostrar los mensajes que queramos:

```
import org.apache.log4j.*;

public class MiOtraClase
{
        static Logger logger = Logger.getLogger(MiOtraClase.class);
        ...
        public MiOtraClase()
        {
                logger.info("Constructor de MiOtraClase");
                ...
        }

        public void hola()
        {
                ...
        }
}
```

En este caso se ha tomado una configuración por defecto, que es la que establece el método `BasicConfigurator.configure()`. Tenemos también opción de pasar como configuración de log un fichero de `Properties` (mediante un `PropertyConfigurator`), o un fichero XML (con un `DomConfigurator`), llamando a sus métodos `configure()` respectivos.

11.2.5.2. Configuración mediante ficheros de propiedades

Por ejemplo, si queremos pasarle a la clase `MiClase` un fichero de `Properties` para configurar el logging, haríamos algo como:

```
import org.apache.log4j.*;

public class MiClase
{
        static Logger logger = Logger.getLogger(MiClase.class);
        ...
        public static void main(String[] args)
        {
                PropertyConfigurator.configure(args[0]);
                logger.info("Entrando en la aplicacion");
                MiOtraClase mc = new MiOtraClase();
                mc.hola();
                logger.info("Finalizando aplicacion");
        }
}
```

Un ejemplo de archivo de configuración es:

```
# Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG
hacia arriba)
# Añade dos appenders, llamados A1 y A2
log4j.rootLogger=DEBUG, A1, A2

# A1 se redirige a la consola
```

```
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 utiliza PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%r [%t] %-5p %c %x %m

# A2  se redirige a un fichero
log4j.appender.A2=org.apache.log4j.RollingFileAppender

# A2 solo muestra mensajes de tipo WARN o superior, en el fichero
logs.txt, hasta 1 MB
log4j.appender.A2.Threshold=WARN
log4j.appender.A2.File=logs.txt
log4j.appender.A2.MaxFileSize=1MB

# En el paquete mipaq.otropaq solo muestra mensajes de tipo WARN o
superior
log4j.logger.mipaq.otropaq=WARN
```

Si no indicamos ningún fichero de configuración (si no hay ninguna línea `configure`), por defecto Log4J buscará en el CLASSPATH un fichero que se llame `log4j.properties` y tomará la configuración de él (si lo encuentra, si no aplicará una configuración por defecto). Así que una alternativa es crearnos ese fichero con la configuración que queramos, en lugar de llamar a otro distinto.

11.2.5.3. Configuración mediante un fichero XML

Si queremos hacer la misma configuración anterior desde un fichero XML, haríamos algo como:

```
import org.apache.log4j.*;

public class MiClase
{
        static Logger logger = Logger.getLogger(MiClase.class);
        ...
        public static void main(String[] args)
        {
                DOMConfigurator.configure(args[0]);
                logger.info("Entrando en la aplicacion");
                MiOtraClase mc = new MiOtraClase();
                mc.hola();
                logger.info("Finalizando aplicacion");
        }
}
```

y como parámetro le pasaríamos el fichero XML de configuración. Un ejemplo de archivo de configuración es:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration>

  <!-- Definimos appender A1 de tipo ConsoleAppender -->
  <appender name="A1" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%r [%t] %-5p %c %x %m"/>
    </layout>
```

```
  </appender>

  <!-- Definimos appender A2 de tipo FileAppender -->
  <appender name="A1" class="org.apache.log4j.FileAppender">
    <param name="File" value="logs.txt"/>
    <param name="Threshold" value="warn"/>
    <param name="MaxFileSize" value="1MB"/>
    <layout class="org.apache.log4j.SimpleLayout">
  </appender>

  <!-- Configuramos el root logger -->
  <root>
    <priority value ="debug" />
    <appender-ref ref="A1"/>
    <appender-ref ref="A2"/>
  </root>

</log4j:configuration>
```

11.2.5.4. Cambios "en caliente"

Adicionalmente, se tienen los métodos `PropertyConfigurator.configureAndWatch(...)` y `DomConfigurator.configureAndWatch(...)`, que establecen la configuración, como `configure()`, pero revisan en tiempo de ejecución el fichero de configuración para aplicar los cambios que sobre él podamos hacer.

11.2.5.5. Sumario de configuraciones

En resumen, tenemos 4 formas de establecer la configuración de Log4J:

- Utilizar `BasicConfigurator.configure()`, en cuyo caso usará la configuración por defecto que tiene Log4J.
- Utilizar `PropertyConfigurator.configure("fichero")` o `PropertyConfigurator.configureAndWatch("fichero")`, en cuyo caso utilizará el fichero de propiedades indicado. En el segundo caso, recogerá automáticamente los cambios que hagamos en el fichero de propiedades
- Utilizar `DOMConfigurator.configure("fichero")` o `DOMConfigurator.configureAndWatch("fichero")`, en cuyo caso utilizará el fichero XML de configuración indicado. En el segundo caso, recogerá automáticamente los cambios que hagamos en dicho fichero
- No poner nada, en cuyo caso buscará un archivo de propiedades `log4j.properties` en el CLASSPATH, con la configuración que hayamos indicado en él

11.3. La librería commons-logging

Aunque Log4J es una librería muy potente para logging, puede que no se adapte a nuestras necesidades, o a las de nuestra empresa. En ese caso, tenemos una alternativa más flexible: utilizar las librerías **commons-logging** de Jakarta: http://jakarta.apache.org/commons/logging/.

Esta librería no proporciona herramientas de logging propiamente dichas, sino una envoltura (*wrapper*) con la que colocar los mensajes de log en nuestras aplicaciones, y bajo la cual podemos poner la librería de logs que queramos, como puedan ser Log4Java, la librería de logging de Java (`SimpleLog`), u otras, configurando adecuadamente el puente entre el wrapper de `commons-logging` y nuestra librería de log.

11.3.1. Definir mensajes de logs en nuestras aplicaciones con commons-logging

Para colocar los mensajes de log en nuestros programas utilizando `commons-logging`, basta con tener cargada la librería JAR correspondiente (normalmente, el fichero `commons-logging-1.1.jar` o similar que venga en la distribución que nos descarguemos). Después, en cada clase sobre la que queramos generar logs, creamos un objeto de tipo `org.apache.commons.logging.Log`, como veremos en el siguiente ejemplo, y llamamos a sus métodos `info()`, `debug()`, `error()`, `warn()`, `fatal()`, como hacíamos con Log4J:

```
import org.apache.commons.logging.*;

public class ClaseAuxiliar
{
  String mensaje = null;
  static Log logger = LogFactory.getLog(ClaseAuxiliar.class);

  public ClaseAuxiliar()
  {
    logger.info("Clase auxiliar inicializada");
  }

  public String mensaje() {
    if (mensaje == null) {
      logger.warn("Atencion, el mensaje es null");
      return "";
    }
    return mensaje;
  }
}
```

11.3.2. Enlazar con nuestra librería: Log4J

Ya tenemos colocados nuestros mensajes de log en la aplicación, pero... ¿cómo le decimos al programa que queremos utilizar Log4J, y un fichero de configuración de Log4J?

Simplemente tenemos que dejar en el CLASSPATH de nuestra aplicación dos ficheros de propiedades: Uno llamado `commons-logging.properties`, en cuyo contenido indicamos, con la siguiente línea, que vamos a utilizar Log4J como logger interno:

```
org.apache.commons.logging.Log=
    org.apache.commons.logging.impl.Log4JLogger
```

El segundo fichero debe llamarse `log4j.properties`, que contendrá la configuración que le queremos dar a Log4Java, como hemos visto antes. Un ejemplo de este fichero

sería:

```
# Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG
hacia arriba)
# Añade appender A1
log4j.rootLogger=DEBUG, A1

# A1 se redirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.Threshold=INFO

# A1 utiliza PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss} - %p -
%m %n
```

Recordemos que Log4J, si no se especifica ningún fichero de configuración, buscará el fichero `log4j.properties` en el CLASSPATH, por eso tenemos que llamarlo así.

11.3.3. Enlazar con otras librerías

Si en lugar de trabajar con Log4J queremos trabajar con otras librerías, tendremos que consultar la documentación de `commons-logging` para ver cómo se hace. Por ejemplo, podemos utilizar la librería de logging que viene con JDK. Para eso, simplemente necesitamos, como antes, dos ficheros de propiedades en el classpath: uno llamado `commons-logging.properties`, como antes, en cuyo contenido indicamos, con la siguiente línea, que vamos a utilizar la librería de JDK como logger interno:

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

El segundo fichero debe llamarse `simplelog.properties`, y contendrá la configuración específica para este tipo de loggers. Un ejemplo sencillo de fichero sería:

```
# Nivel de log general
# ("trace", "debug", "info", "warn", "error", o "fatal").
# Por defecto es "info"
org.apache.commons.logging.simplelog.defaultlog=warn

# A true si queremos que el nombre del Log aparezca en cada mensaje en la
salida
org.apache.commons.logging.simplelog.showlogname=false

# A true si queremos que el nombre corto del Log (su último término)
aparezca en cada mensaje en la salida
org.apache.commons.logging.simplelog.showShortLogname=true

# A true si queremos poner fecha y hora actuales en cada mensaje en la
salida
org.apache.commons.logging.simplelog.showdatetime=true
```

11.3.4. Conclusiones

Observad que la principal ventaja de utilizar `commons-logging` es que, una vez colocados los mensajes de log en nuestra aplicación, ya no tenemos que tocar nada del código fuente para configurarla, ni para cambiar de librería de log si queremos. Basta con colocar en el

CLASSPATH los ficheros de propiedades correspondientes, y reajustar en ellos la configuración de logging que deseamos.

12. Ejercicios de depuración y logging

12.1. Depuración de código (1.5 puntos)

Vamos a depurar la función para el cálculo del importe total de una venta, que se encuentra en el proyecto `lja-debug` de las plantillas de las sesión. La función `calculaImporteTotal` (de la clase `Venta`) toma como entrada los datos de la venta que comprenden una serie de líneas de venta, con el precio unitario de cada producto, el tipo del producto, y el número de unidades vendidas, y devuelve un valor numérico con el importe total de la misma. Según el tipo de producto que tengamos se le aplicará un IVA distinto en el precio. Se pide:

a) Añadir un breakpoint al comienzo de la función. Ejecutar paso a paso y monitorizar el valor de la variable `total` y de las expresiones `lv.getCantidad() * lv.getPrecio()` (precio antes de aplicar el IVA) y `lv.getCantidad() * lv.getPrecio() * (lv.getTipo().getIva())` (precio con el IVA incluido). Comprobar el valor que toman en cada iteración y detectar posibles errores en el código. Hacer las correcciones oportunas.

b) Añadir un breakpoint condicional que sólo se ejecute cuando la cantidad del producto actual sea mayor que uno.

c) Probar a cambiar en tiempo de ejecución la cantidad de uno de los productos.

12.2. Logs al leer ficheros (1.5 puntos)

Vamos a añadir logs a nuestra aplicación de filmoteca. Se pide:

- Añadir mensajes de depuración (DEBUG) en `FilePeliculaDAO` que indiquen qué va haciendo el programa en cada paso, y mensajes de error (ERROR) cuando ocurra un error (en los casos en los que se lance una excepción).
- Modificar la configuración del logging, mediante un fichero de propiedades, para que:
  - Que todos los mensajes vayan a la consola, con el formato simple.
  - Además, que la salida de mensajes de tipo ERROR se vuelque a un fichero *errores.log*, con el siguiente formato:

```
[dia/mes/año hora:minuto:segundo] - tipo error - mensaje
```

Utilizaremos la librería *commons logging* para gestionar los *logs*, y dentro de ella indicaremos que se utilice *log4j*.

Nota

Para crear los ficheros de configuración necesarios podemos basarnos en los ejemplos que se encuentran en el proyecto `lja-debug` de las plantillas de la sesión (utilizar los ficheros de

configuración por defecto que tenemos en el directorio `resources`). También podemos coger de este proyecto las librerías de *logs* necesarias.

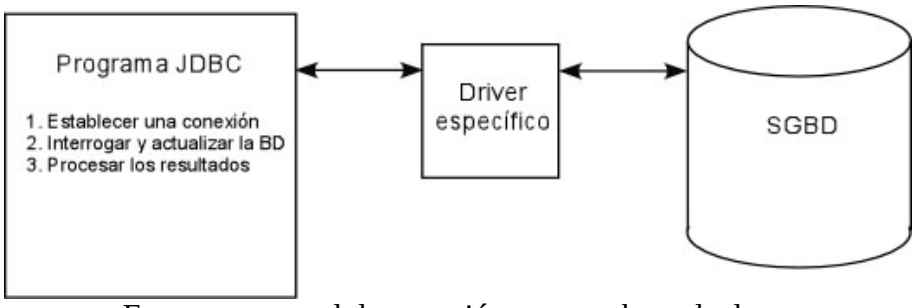
- Probad el programa creando, eliminando y consultando la lista de películas. Forzar a que se produzcan errores para que se muestren los *logs* de error.
- Mediante el fichero de configuración, haced que el fichero *errores.log* no pueda exceder de 1 KB (deberéis utilizar un *RollingFileAppender* en lugar de un *FileAppender*) y aseguraos de que no se sobreescriba su contenido tras cada nueva ejecución. Probad a ejecutar el programa varias veces, provocando errores que se guarden en dicho fichero, y comprobad cómo se rotan los mensajes.
- Probad a que se guarden 2 copias de respaldo del fichero *errores.log*, y generad mensajes suficientes en él para que sobrepase el tamaño de un fichero. Observad cómo se generan los ficheros de respaldo, y qué nombres reciben.

13. Java Database Connectivity

13.1. Introducción a JDBC

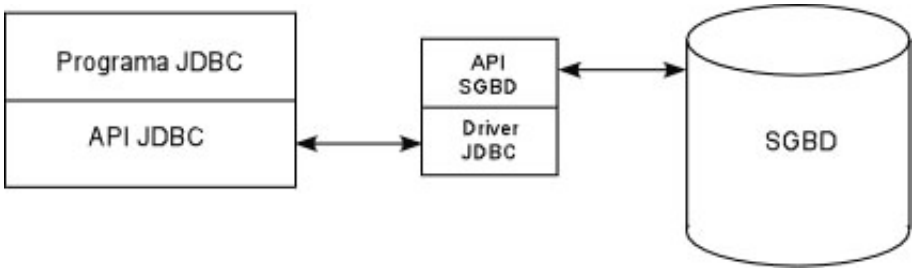
En la mayoría de las aplicaciones que nos vamos a encontrar, aparecerá una base de datos como fuente de información. JDBC nos va a permitir acceder a bases de datos (BD) desde Java. Con JDBC no es necesario escribir distintos programas para distintas BD, sino que un único programa sirve para acceder a BD de distinta naturaleza. Incluso, podemos acceder a más de una BD de distinta fuente (Oracle, Access, MySql, etc.) en la misma aplicación. Podemos pensar en JDBC como el puente entre una base de datos y nuestro programa Java. Un ejemplo sencillo puede ser un applet que muestra dinámicamente información contenida en una base de datos. El applet utilizará JDBC para obtener dichos datos.

El esquema a seguir en un programa que use JDBC es el siguiente:



Esquema general de conexión con una base de datos

Un programa Java que utilice JDBC primero deberá establecer una conexión con el SGBD. Para realizar dicha conexión haremos uso de un driver específico para cada SGBD que estemos utilizando. Una vez establecida la conexión ya podemos interrogar la BD con cualquier comando SQL (select, update, create, etc.). El resultado de un comando *select* es un objeto de la clase ResultSet, que contiene los datos que devuelve la consulta. Disponemos de métodos en *ResultSet* para manejar los datos devueltos. También podemos realizar cualquier operación en SQL (creación de tablas, gestión de usuarios, etc.).



Conexión a través del API y un driver de JDBC

Para realizar estas operaciones necesitaremos contar con un SGBD (sistema gestor de bases de datos) además de un driver específico para poder acceder a este SGBD. Vamos a utilizar dos SGBD: MySQL (disponible para Windows y Linux, de libre distribución) y PostGres (sólo para Linux, también de libre distribución).

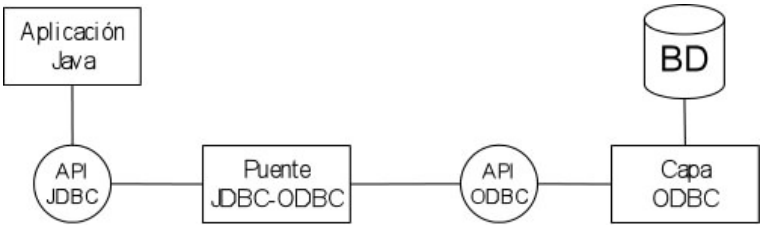
13.1.1. Drivers de acceso

Los drivers para poder acceder a cada SGBD no forman parte de la distribución de Java por lo que deberemos obtenerlos por separado. ¿Por qué hacer uso de un driver?. El principal problema que se nos puede plantear es que cada SGBD dispone de su propio API (la mayoría propietario), por lo que un cambio en el SGBD implica una modificación de nuestro código. Si colocamos una capa intermedia, podemos abstraer la conectividad, de tal forma que nosotros utilizamos un objeto para la conexión, y el driver se encarga de traducir la llamada al API. El driver lo suelen distribuir las propias empresas que fabrican el SGBD.

13.1.1.1. Tipos de drivers

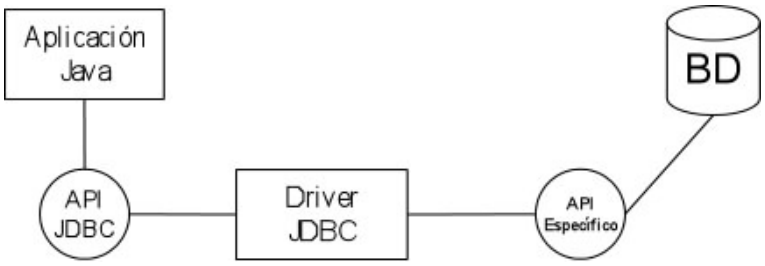
Existe un estándar establecido que divide los drivers en cuatro grupos:

- **Tipo 1: Puente JDBC-ODBC**. ODBC (Open Database Connectivity) fue creado para proporcionar una conexión a bases de datos en Microsoft Windows. ODBC permite acceso a bases de datos desde diferentes lenguajes de programación, tales como C y Cobol. El puente JDBC-ODBC permite enlazar Java con cualquier base de datos disponible en ODBC. No se aconseja el uso de este tipo de driver cuando tengamos que acceder a bases de datos de alto rendimiento, pues las funcionalidades están limitadas a las que marca ODBC. Cada cliente debe tener instalado el driver. J2SE incluye este driver en su versión Windows y Solaris.



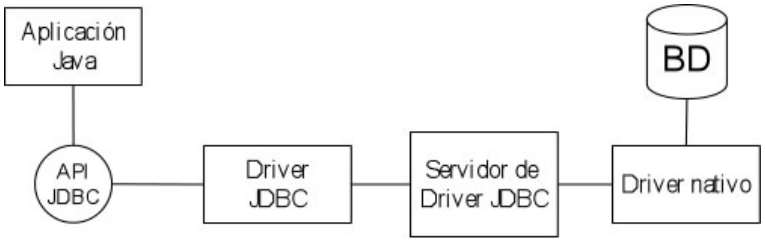
Configuración de un driver de tipo 1

- **Tipo 2: Parte Java, parte driver nativo**. Es una combinación de implementación Java y API nativo para el acceso a la base de datos. Este tipo de driver es más rápido que el anterior, pues no se realiza el paso por la capa ODBC. Las llamadas JDBC se traducen en llamadas específicas del API de la base de datos. Cada cliente debe tener instalado el driver. Tiene menor rendimiento que los dos siguientes y no se pueden usar en Internet, ya que necesita el API de forma local.



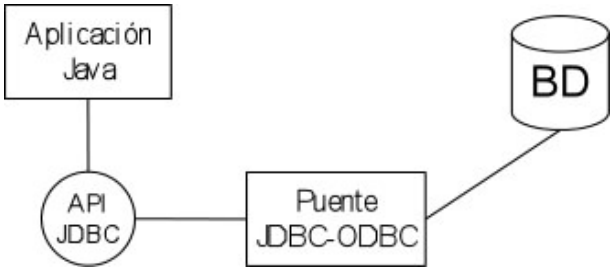
Configuración de un driver de tipo 2

- **Tipo 3: Servidor intermediario de acceso a base de datos**. Este tipo de driver proporciona una abstracción de la conexión. El cliente se conecta a los SGBD mediante un componente servidor intermedio, que actúa como una puerta para múltiples servidores. La ventaja de este tipo de driver es el nivel de abstracción. El servidor de aplicaciones WebLogic incorpora este tipo de driver.



Configuración de un driver de tipo 3

- **Tipo 4: Drivers Java**. Este es el más directo. La llamada JDBC se traduce directamente en una llamada de red a la base de datos, sin intermediarios. Proporcionan mejor rendimiento. La mayoría de SGBD proporcionan drivers de este tipo.



Configuración de un driver de tipo 4

13.1.1.2. Instalación de drivers

La distribución de JDBC incorpora los drivers para el puente JDBC-ODBC que nos permite acceder a cualquier BD que se gestione con ODBC. Para MySQL, deberemos descargar e instalar el SGBD y el driver, que puede ser obtenido en la dirección http://www.mysql.com/downloads/connector/j/. El driver para PostGres se obtiene en http://jdbc.postgresql.org

Para instalar el driver lo único que deberemos hacer es incluir el fichero JAR que lo contiene en el CLASSPATH. Por ejemplo, para MySQL:

```
export CLASSPATH=$CLASSPATH:
  /directorio-donde-este/mysql-connector-java-3.0.15-ga-bin.jar
```

Con el driver instalado, podremos cargarlo desde nuestra aplicación simplemente cargando dinámicamente la clase correspondiente al driver:

```
Class.forName("com.mysql.jdbc.Driver");
```

El driver JDBC-ODBC se carga como se muestra a continuación:

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

Y de forma similar para PostGres:

```
Class.forName("org.postgresql.Driver");
```

La carga del driver se debería hacer siempre antes de conectar con la BD.

Como hemos visto anteriormente, pueden existir distintos tipos de drivers para la misma base de datos. Por ejemplo, a una BD en MySQL podemos acceder mediante ODBC o mediante su propio driver. Podríamos pensar que la solución más sencilla sería utilizar ODBC para todos las conexiones a SGBD. Sin embargo, dependiendo de la complejidad de la aplicación a desarrollar esto nos podría dar problemas. Determinados SGBD permiten realizar operaciones (transacciones, mejora de rendimiento, escabilidad, etc.) que se ven mermadas al realizar su conexión a través del driver ODBC. Por ello es preferible hacer uso de driver específicos para el SGBD en cuestión.

El ejemplo más claro de problemas en el uso de drivers es con los *Applets*. Cuando utilicemos acceso a bases de datos mediante JDBC desde un *Applet*, deberemos tener en cuenta que el *Applet* se ejecuta en la máquina del cliente, por lo que si la BD está alojada en nuestro servidor tendrá que establecer una conexión remota. Aquí encontramos el problema de que si el *Applet* es visible desde Internet, es muy posible que el puerto en el que escucha el servidor de base de datos puede estar cortado por algún *firewall*, por lo que el acceso desde el exterior no sería posible.

El uso del puente JDBC-ODBC tampoco es recomendable en *Applets*, ya que requiere que cada cliente tenga configurada la fuente de datos ODBC adecuada en su máquina. Esto podemos controlarlo en el caso de una intranet, pero en el caso de Internet será mejor utilizar otros métodos para la conexión.

En cuanto a las excepciones, debemos capturar la excepción *SQLException* en casi todas las operaciones en las que se vea involucrado algún objeto JDBC.

13.1.2. Conexión a la BD

Una vez cargado el driver apropiado para nuestro SGBD deberemos establecer la conexión con la BD. Para ello utilizaremos el siguiente método:

```
Connection con = DriverManager.getConnection(url);
Connection con = DriverManager.getConnection(url, login, password);
```

La conexión a la BD está encapsulada en un objeto `Connection`. Para su creación debemos proporcionar la *url* de la BD y, si la BD está protegida con contraseña, el *login* y *password* para acceder a ella. El formato de la *url* variará según el driver que utilicemos. Sin embargo, todas las *url* tendrán la siguiente forma general: *jdbc*:<*subprotocolo*>:<*nombre*>, con *subprotocolo* indicando el tipo de SGBD y con *nombre* indicando el nombre de la BD y aportando información adicional para la conexión.

Para conectar a una fuente ODBC de nombre *bd*, por ejemplo, utilizaremos la siguiente URL:

```
Connection con = DriverManager.getConnection("jdbc:odbc:bd");
```

En el caso de MySQL, si queremos conectarnos a una BD de nombre *bd* alojada en la máquina local (*localhost*) y con usuario *miguel* y contraseña *m++24*, lo haremos de la siguiente forma:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/bd",
                                "miguel", "m++24");
```

En el caso de PostGres (notar que hemos indicado un puerto de conexión, el 5432):

```
Connection con = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/bd", "miguel", "m++24");
```

Podemos depurar la conexión y determinar qué llamadas está realizando JDBC. Para ello haremos uso de un par de métodos que incorpora `DriverManager`. En el siguiente ejemplo se indica que las operaciones que realice JDBC se mostrarán por la salida estándar:

```
DriverManager.setLogWriter(new PrintWriter(System.out, true));
```

Una vez realizada esta llamada también podemos mostrar mensajes usando:

```
DriverManager.println("Esto es un mensaje");
```

13.2. Consulta a una base de datos con JDBC

13.2.1. Creación y ejecución de sentencias SQL

Una vez obtenida la conexión a la BD, podemos utilizarla para realizar consultas, inserción y/o borrado de datos de dicha BD. Todas estas operaciones se realizarán mediante lenguaje SQL. La clase `Statement` es la que permite realizar todas estas operaciones. La instanciación de esta clase se realiza haciendo uso del siguiente método que proporciona el objeto `Connection`:

```
Statement stmt = con.createStatement();
```

Podemos dividir las sentencias SQL en dos grupos: las que actualizan la BD y las que únicamente la consultan. En las siguientes secciones veremos cómo podemos realizar estas dos acciones.

13.2.2. Sentencias de consulta

Para obtener datos almacenados en la BD podemos realizar una consulta SQL (*query*). Podemos ejecutar la consulta utilizando el objeto `Statement`, pero ahora haciendo uso del método `executeQuery` al que le pasaremos una cadena con la consulta SQL. Los datos resultantes nos los devolverá como un objeto `ResultSet`.

```
ResultSet result = stmt.executeQuery(query);
```

La consulta SQL nos devolverá una tabla, que tendrá una serie de campos y un conjunto de registros, cada uno de los cuales consistirá en una tupla de valores correspondientes a los campos de la tabla.

Los campos que tenga la tabla resultante dependerán de la consulta que hagamos, de los datos que solicitemos que nos devuelva. Por ejemplo, podemos solicitar que una consulta nos devuelva los campos *expediente* y *nombre* de los alumnos o bien que nos devuelva todos los campos de la tabla *alumnos*.

Veamos el funcionamiento de las consultas SQL mediante un ejemplo:

```
String query = "SELECT * FROM ALUMNOS WHERE sexo = 'M'";
ResultSet result = stmt.executeQuery(query);
```

En esta consulta estamos solicitando todos los registros de la tabla ALUMNOS en los que el sexo sea *mujer* (M), pidiendo que nos devuelva todos los campos (indicado con *) de dicha tabla. Nos devolverá una tabla como la siguiente:

exp	nombre	sexo
1286	Amparo	M
1287	Manuela	M
1288	Lucrecia	M

Estos datos nos los devolverá como un objeto `ResultSet`. A continuación veremos cómo podemos acceder a los valores de este objeto y cómo podemos movernos por los distintos registros.

El objeto `ResultSet` dispone de un *cursor* que estará situado en el registro que podemos consultar en cada momento. Este *cursor* en un principio estará situado en una posición anterior al primer registro de la tabla. Podemos mover el cursor al siguiente registro con el método `next` del `ResultSet`. La llamada a este método nos devolverá `true` mientras pueda pasar al siguiente registro, y `false` en el caso de que ya estuviéramos en el último registro de la tabla. Para la consulta de todos los registros obtenidos utilizaremos

normalmente un bucle como el siguiente:

```
while(result.next()) {
      // Leer registro
}
```

Ahora necesitamos obtener los datos del registro que marca el *cursor*, para lo cual podremos acceder a los campos de dicho registro. Esto lo haremos utilizando los métodos `getXXXX(campo)` donde `XXXX` será el tipo de datos de Java en el que queremos que nos devuelva el valor del campo. Hemos de tener en cuenta que el tipo del campo en la tabla debe ser convertible al tipo de datos Java solicitado. Para especificar el campo que queremos leer podremos utilizar bien su nombre en forma de cadena, o bien su índice que dependerá de la ordenación de los campos que devuelve la consulta. También debemos tener en cuenta que no podemos acceder al mismo campo dos veces seguidas en el mismo registro. Si lo hacemos nos dará una excepción.

Los tipos principales que podemos obtener son los siguientes:

`getInt`	Datos enteros
`getDouble`	Datos reales
`getBoolean`	Campos booleanos (si/no)
`getString`	Campos de texto
`getDate`	Tipo fecha (Devuelve `Date`)
`getTime`	Tipo hora (Devuelve `Time`)

Si queremos imprimir todos los datos obtenidos de nuestra tabla ALUMNOS del ejemplo podremos hacer lo siguiente:

```
int exp;
String nombre;
String sexo;

while(result.next()){
    exp = result.getInt("exp");
    nombre = result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" + nombre + "\t" + sexo);
}
```

Cuando un campo de un registro de una tabla no tiene asignado ningún valor, la consulta de ese valor devuelve NULL. Esta situación puede dar problemas al intentar manejar ese dato. La clase `ResultSet` dispone de un método `wasNull` que llamado después de acceder a un registro nos dice si el valor devuelto fue NULL. Esto no sucede así para los datos numéricos, ya que devuelve el valor 0. Comprobemos qué sucede en el siguiente código:

```
String sexo;

while(result.next()){
    exp = result.getInt("exp");
```

```
    nombre = result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" + nombre.trim() + "\t" + sexo);
}
```

La llamada al método `trim` devolverá una excepción si el objeto `nombre` es NULL. Por ello podemos realizar la siguiente modificación:

```
String sexo;

while(result.next()){
    exp = result.getInt("exp");
    System.out.print(exp + "\t");
    nombre = result.getString("nombre");
    if (result.wasNull()) {
        System.out.print("Sin nombre asignado");
    else
        System.out.print(nombre.trim());
    sexo = result.getString("sexo");
    System.out.println("\t" + sexo);
}
```

13.3. Restricciones y movimientos en el ResultSet

Cuando realizamos llamadas a BD de gran tamaño el resultado de la consulta puede ser demasiado grande y no deseable en términos de eficiencia y memoria. JDBC permite restringir el número de filas que se devolverán en el `ResultSet`. La clase `Statement` incorpora dos métodos, `getMaxRows` y `setMaxRows`, que permiten obtener e imponer dicha restricción. Por defecto, el límite es cero, indicando que no se impone la restricción. Si, por ejemplo, antes de ejecutar la consulta imponemos un límite de 30 usando el método `setMaxRows(30)`, el resultado devuelto sólo contendrá las 30 primeras filas que cumplan con los criterios de la consulta.

Hasta ahora, el manejo de los datos devueltos en una consulta se realizaba con el método `next` de `ResultSet`. Podemos manejar otros métodos para realizar un movimiento no lineal por el `ResultSet`. Es lo que se conoce como `ResultSet` arrastable. Para que esto sea posible debemos utilizar el siguiente método en la creación del `Statement`:

```
Statement createStatement (int resultSetType,
                                   int resultSetConcurrency)
```

Los posibles valores que puede tener `resultSetType` son: `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_SENSITIVE`. El primer valor es el funcionamiento por defecto: el `ResultSet` sólo se mueve hacia adelante. Los dos siguientes permiten que el resultado sea arrastable. Una característica importante en los resultados arrastables es que los cambios que se produzcan en la BD se reflejan en el resultado, aunque dichos cambios se hayan producido después de la consulta. *Esto dependerá de si el driver y/o la BD soporta este tipo de comportamiento*. En el caso de `INSENSITIVE`, el resultado no es sensible a dichos cambios y en el caso de `SENSITIVE`, sí. Los métodos que podemos utilizar para movernos por el `ResultSet` son:

`next`	Pasa a la siguiente fila
`previous`	Ídem fila anterior
`last`	Ídem última fila
`first`	Ídem primera fila
`absolute(int fila)`	Pasa a la fila número fila
`relative(int fila)`	Pasa a la fila número fila desde la actual
`getRow`	Devuelve la número de fila actual
`isLast`	Devuelve si la fila actual es la última
`isFirst`	Ídem la primera

El otro parámetro, `resultSetConcurrency`, puede ser uno de estos dos valores: `ResultSet.CONCUR_READ_ONLY` y `ResultSet.CONCUR_UPDATABLE`. El primero es el utilizado por defecto y no permite actualizar el resultado. El segundo permite que los cambios realizados en el `ResultSet` se actualicen en la base de datos. Si queremos modificar los datos obtenidos en una consulta y queremos reflejar esos cambios en la BD debemos crear una sentencia con `TYPE_FORWARD_SENSITIVE` y `CONCUR_UPDATABLE`.

13.3.1. Actualización de datos

Para actualizar un campo disponemos de métodos `updateXXXX`, de la misma forma que teníamos métodos `getXXXX`. Estos métodos reciben dos parámetros: el primero indica el nombre del campo (o número de orden dentro del `ResultSet`); el segundo indica el nuevo valor que tomará el campo del registro actual. Para que los cambios tengan efecto en la BD debemos llamar al método `updateRow`. El siguiente código es un ejemplo de modificación de datos:

```
rs.updateString("nombre","manolito");
rs.updateRow();
```

Si queremos desechar los cambios producidos en la fila actual (antes de llamar a `updateRow`) podemos llamar a `cancelRowUpdates`. Para borrar la fila actual tenemos el método `deleteRow`. La llamada a este método deja una fila vacía en el `ResultSet`. Si intentamos acceder a los datos de esa fila nos dará una excepción. Podemos llamar al método `rowDeleted` el cual devuelve cierto si la fila actual ha sido eliminada (método no implementado en MySQL).

Debemos tener en cuenta varias restricciones a la hora de actualizar un `ResultSet`: la sentencia SELECT que ha generado el `ResultSet` debe:

- Referenciar sólo una tabla.
- No contener una claúsula *join* o *group by*.
- Seleccionar la clave primaria de la tabla.

Existe un registro especial al que no se puede acceder como hemos visto anteriormente, que es el registro de inserción. Este registro se utiliza para insertar nuevos registros en la tabla. Para situarnos en él deberemos llamar al método `moveToInsertRow`. Una vez situados en él deberemos asignar los datos con los métodos `updateXXXX` anteriormente descritos y una vez hecho esto llamar a `insertRow` para que el registro se inserte en la BD. Podemos volver al registro donde nos encontrábamos antes de movernos al registro de inserción llamando a `moveToCurrentRow`.

13.4. Sentencias de actualización

La clase `Statement` dispone de un método llamado `executeUpdate` el cual recibe como parámetro la cadena de caracteres que contiene la sentencia SQL a ejecutar. Este método únicamente permite realizar sentencias de actualización de la BD: creación de tablas (CREATE), inserción (INSERT), actualización (UPDATE) y borrado de datos (DELETE). El método a utilizar es el siguiente:

```
stmt.executeUpdate(sentencia);
```

Vamos a ver a continuación un ejemplo de estas operaciones. Crearemos una tabla ALUMNOS en nuestra base de datos y añadiremos datos a la misma. La sentencia para la creación de la tabla será la siguiente:

```
String st_crea = "CREATE TABLE ALUMNOS (
    exp INTEGER,
    nombre VARCHAR(32),
    sexo CHAR(1),
    PRIMARY KEY (exp)
)";
stmt.executeUpdate(st_crea);
```

Una vez creada la tabla podremos insertar datos en ella como se muestra a continuación:

```
String st_inserta = "INSERT INTO ALUMNOS(exp, nombre)
    VALUES(1285, 'Manu', 'M')";
stmt.executeUpdate(st_inserta);
```

Cuando tengamos datos dentro de la tabla, podremos modificarlos utilizando para ello una sentencia UPDATE:

```
String st_actualiza = "UPDATE FROM ALUMNOS
    SET sexo = 'H' WHERE exp = 1285";
stmt.executeUpdate(st_actualiza);
```

Si queremos eliminar un registro de la tabla utilizaremos una sentencia DELETE como se muestra a continuación:

```
String st_borra = "DELETE FROM ALUMNOS
    WHERE exp = 1285";
stmt.executeUpdate(st_borra);
```

El método `executeUpdate` nos devuelve un entero que nos dice el número de registros a los que ha afectado la operación, en caso de sentencias INSERT, UPDATE y DELETE.

La creación de tablas nos devuelve siempre 0.

13.5. Otras llamadas a la BD

En la interfaz `Statement` podemos observar un tercer método que podemos utilizar para la ejecución de sentencias SQL. Hasta ahora hemos visto como para la ejecución de sentencias que devuelven datos (consultas) debemos usar `executeQuery`, mientras que para las sentencias INSERT, DELETE, UPDATE e instrucciones DDL utilizamos `executeUpdate`. Sin embargo, puede haber ocasiones en las que no conozcamos de antemano el tipo de la sentencia que vamos a utilizar (por ejemplo si la sentencia la introduce el usuario). En este caso podemos usar el método `execute`.

```
boolean hay_result = stmt.execute(sentencia);
```

Podemos ver que el método devuelve un valor *booleano*. Este valor será *true* si la sentencia ha devuelto resultados (uno o varios objetos `ResultSet`), y *false* en el caso de que sólo haya devuelto el número de registros afectados. Tras haber ejecutado la sentencia con el método anterior, para obtener estos datos devueltos proporciona una serie de métodos:

```
int n = stmt.getUpdateCount();
```

El método `getUpdateCount` nos devuelve el número de registros a los que afecta la actualización, inserción o borrado, al igual que el resultado que devolvía `executeUpdate`.

```
ResultSet rs = stmt.getResultSet();
```

El método `getResultSet` nos devolverá el objeto `ResultSet` que haya devuelto en el caso de ser una consulta, al igual que hacía `executeQuery`. Sin embargo, de esta forma nos permitirá además tener múltiples objetos `ResultSet` como resultado de una llamada. Eso puede ser necesario, por ejemplo, en el caso de una llamada a un procedimiento, que nos puede devolver varios resultados como veremos más adelante. Para movernos al siguiente `ResultSet` utilizaremos el siguiente método:

```
boolean hay_mas_results = stmt.getMoreResults();
```

La llamada a este método nos moverá al siguiente `ResultSet` devuelto, devolviéndonos *true* en el caso de que exista, y *false* en el caso de que no haya más resultados. Si existe, una vez nos hayamos movido podremos consultar el nuevo `ResultSet` llamando nuevamente al método `getResultSet`.

Otra llamada disponible es el método `executeBatch`. Este método nos permite enviar varias sentencias SQL a la vez. No puede contener sentencias SELECT. Devuelve un array de enteros que indicará el número de registros afectados por las sentencias SQL. Para añadir sentencias haremos uso del método `addBatch`. Un ejemplo de ejecución es el siguiente:

```
stmt.addBatch("INSERT INTO ALUMNOS(exp, nombre)
    VALUES(1285, 'Manu', 'M')");
```

```
stmt.addBatch("INSERT INTO ALUMNOS(exp, nombre)
    VALUES(1299, 'Miguel', 'M')");

int[] res = stmt.executeBatch();
```

Por último, vamos a comentar el método `getGeneratedKeys`, también del objeto `Statement`. En muchas ocasiones hacemos inserciones en tablas cuyo identificador es un autonumérico. Por lo tanto, este valor no lo especificaremos nosotros manualmente, sino que se asignará de forma automática en la inserción. Sin embargo, muchas veces nos puede interesar conocer cual ha sido dicho identificador, para así por ejemplo poder insertar a continuación un registro de otra tabla que haga referencia al primero. Esto lo podremos hacer con el método `getGeneratedKeys`, que nos devuelve un `ResultSet` que contiene la clave generada:

```
ResultSet res = sentSQL.getGeneratedKeys();
int id = -1;
if(res.next()) {
    id = res.getInt(1);
}
```

13.6. Optimización de sentencias

Cuando ejecutamos una sentencia SQL, esta se compila y se manda al SGBD. Si la vamos a invocar repetidas veces, puede ser conveniente dejar esa sentencia preparada (precompilada) para que pueda ser ejecutada de forma más eficiente. Para hacer esto utilizaremos la interfaz `PreparedStatement`, que podrá obtenerse a partir de la conexión a la BD de la siguiente forma:

```
PreparedStatement ps = con.prepareStatement("UPDATE FROM alumnos
    SET sexo = 'H' WHERE exp>1200 AND exp<1300");
```

Vemos que a este objeto, a diferencia del objeto `Statement` visto anteriormente, le proporcionamos la sentencia SQL en el momento de su creación, por lo que estará preparado y optimizado para la ejecución de dicha sentencia posteriormente.

Sin embargo, lo más común es que necesitemos hacer variaciones sobre la sentencia, ya que normalmente no será necesario ejecutar repetidas veces la misma sentencia exactamente, sino variaciones de ella. Por ello, este objeto nos permite parametrizar la sentencia. Estableceremos las posiciones de los parámetros con el carácter '?' dentro de la cadena de la sentencia, tal como se muestra a continuación:

```
PreparedStatement ps = con.prepareStatement("UPDATE FROM alumnos
    SET sexo = 'H' WHERE exp > ? AND exp < ?");
```

En este caso tenemos dos parámetros, que será el número de expediente mínimo y el máximo del rango que queremos actualizar. Cuando ejecutemos esta sentencia, el sexo de los alumnos desde expediente inferior hasta expediente superior se establecerá a 'H'.

Para dar valor a estos parámetros utilizaremos los métodos `setXXX` donde `XXX` será el tipo

de los datos que asignamos al parámetro (recordad los métodos del `ResultSet`), indicando el número del parámetro (que empieza desde 1) y el valor que le queremos dar. Por ejemplo, para asignar valores enteros a los parámetros de nuestro ejemplo haremos:

```
ps.setInt(1,1200);
ps.setInt(2,1300);
```

Una vez asignados los parámetros, podremos ejecutar la sentencia llamando al método `executeUpdate` (ahora sin parámetros) del objeto `PreparedStatement`:

```
int n = ps.executeUpdate();
```

Igual que en el caso de los objetos `Statement`, podremos utilizar cualquier otro de los métodos para la ejecución de sentencias, `executeQuery` o `execute`, según el tipo de sentencia que vayamos a ejecutar.

Una característica importante es que los parámetros sólo sirven para datos, es decir, no podemos sustituir el nombre de la tabla o de una columna por el signo '?'. Otra cosa a tener en cuenta es que una vez asignados los parámetros, estos no desaparecen, sino que se mantienen hasta que se vuelvan a asignar o se ejecute una llamada al método `clearParameters`.

13.6.1. SQL injection

Un problema de seguridad en la base de datos que se nos puede plantear es el SQL injection. Se trata de insertar código SQL dentro de otro código SQL, para alterar su funcionamiento y conseguir que se ejecute alguna sentencia maliciosa. Imaginad que tenemos el siguiente código en una página .jsp o en una clase Java:

```
String s="SELECT * FROM usuarios WHERE nombre='"+nombre+"';";
```

La variable `nombre` es una cadena cuyo valor viene de un campo que es introducido por el usuario. Al introducir el usuario un nombre cualquiera, el código SQL se ejecuta y nada extraño pasa. Pero esta opción nos permite añadir código propio que nos permita dañar o incluso permitirnos tomar el control de la BD. Imaginad que el usuario ha introducido el siguiente código: `Miguel'; drop table usuarios; select * from usuarios; grant all privileges ...`. Entonces el código que se ejecutaría sería:

```
SELECT * FROM usuarios WHERE nombre='Miguel'; drop table usuarios;
select * from usuarios; grant all privileges ...
```

Para evitar la inyección de SQL se recurre, en Java, a usar una sentencia preparada. De esta manera tendríamos:

```
PreparedStatement ps = con.prepareStatement(
                  "SELECT * FROM usuarios WHERE nombre=?");
ps.setString(nombre);
```

Y la sentencia SQL que se ejecutaría, con la misma entrada que antes, es:

```
SELECT * FROM usuarios WHERE nombre="Miguel'; drop table usuarios;
select * from usuarios; grant all privileges ...";
```

13.7. Transacciones

Muchas veces, cuando tengamos que realizar una serie de acciones, queremos que todas se hayan realizado correctamente, o bien que no se realice ninguna de ellas, pero no que se realicen algunas y otras no.

Podemos ver esto mediante un ejemplo, en el que se va a hacer una reserva de vuelos para ir desde Alicante a Osaka. Para hacer esto tendremos que hacer trasbordo en dos aeropuertos, por lo que tenemos que reservar un vuelo Alicante-Madrid, un vuelo Madrid-Amsterdam y un vuelo Amsterdam-Osaka. Si cualquiera de estos tres vuelos estuviese lleno y no pudiésemos reservar, no queremos reservar ninguno de los otros dos porque no nos serviría de nada. Por lo tanto, sólo nos interesa que la reserva se lleve a cabo si podemos reservar los tres vuelos.

Una transacción es un conjunto de sentencias que deben ser ejecutadas como una unidad, de forma que si una de ellas no puede realizarse, no se llevará a cabo ninguna. Dicho de otra manera, las transacciones hacen que la BD pase de un estado consistente al siguiente.

Pero para hacer esto encontramos un problema. Pensemos en nuestro ejemplo de la reserva de vuelos, en la que necesitaremos realizar las siguientes inserciones (reservas):

```
try {
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
        + "destino) VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
        + "destino) VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
        + "destino) VALUES('Paquito', 'Amsterdam', 'Osaka')");
}
catch(SQLException e) {
    // ¿Dónde ha fallado? ¿Qué hacemos ahora?
}
```

En este caso, vemos que si falla la reserva de uno de los tres vuelos obtendremos una excepción, pero en ese caso, ¿cómo podremos saber dónde se ha producido el fallo y hasta qué acción debemos deshacer? Con la excepción lo único que sabemos es que algo ha fallado, pero no sabremos dónde ha sido, por lo que de esta forma no podremos saber hasta qué acción debemos deshacer.

Para hacer esto de una forma limpia asegurando la consistencia de los datos, utilizaremos las operaciones de *commit* y *rollback.*

Cuando realicemos cambios en la base de datos, estos cambios se harán efectivos en ella de forma persistente cuando realicemos la operación *commit*. En el modo de operación que hemos visto hasta ahora, por defecto tenemos activado el modo *auto-commit*, de forma que siempre que ejecutamos alguna sentencia se realiza *commit* automáticamente.

Sin embargo, en el caso de las transacciones con múltiples sentencias, no nos interesará hacer estos cambios persistentes hasta haber comprobado que todos los cambios se pueden hacer de forma correcta. Para ello desactivaremos este modo con:

```
con.setAutoCommit(false);
```

Al desactivar este modo, una vez hayamos hecho las modificaciones de forma correcta, deberemos hacerlas persistentes mediante la operación *commit* llamando de forma explícita a:

```
con.commit();
```

Si por el contrario hemos obtenido algún error, no queremos que esas modificaciones se lleven a cabo finalmente en la BD, por lo que podremos deshacerlas llamando a:

```
con.rollback();
```

Por lo tanto, la operación *rollback* deshará todos los cambios que hayamos realizado para los que todavía no hubiésemos hecho *commit* para hacerlos persistentes, permitiéndonos de esta forma implementar estas transacciones de forma atómica.

Nuestro ejemplo de la reserva de vuelos debería hacerse de la siguiente forma:

```
try {
    con.setAutoCommit(false);
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
           + "destino) VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
           + "destino) VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, " +
           + "destino) VALUES('Paquito', 'Amsterdam', 'Osaka')");
    // Hemos podido reservar los tres vuelos correctamente
    con.commit();
}
catch(SQLException e) {
    // Alguno de los tres ha fallado, deshacemos los cambios
    try {
        con.rollback();
    }
    catch(SQLException e) {};
}
```

Una característica relacionada con las transacciones es la concurrencia en el acceso a la BD. Dicho de otra forma, qué sucede cuando varios usuarios se encuentran accediendo a la vez a los mismos datos y pretenden modificarlos. Un ejemplo sencillo: tenemos una tienda y dos usuarios están accediendo al mismo disco, del cual sólo queda una unidad. El primero de los usuarios consulta el disponible, comprueba que existe una unidad y lo introduce en su cesta de la compra. El otro usuario en ese preciso momento también está consultando el disponible, también le aparece una unidad y también intenta introducirlo en su cesta de la compra. Al segundo usuario el sistema no debería dejarle actualizar los datos que está manejando el primero.

La concurrencia es manejada por los distintos SGBD de manera distinta. Para saber el nivel aislamiento entre diferentes accesos podemos utilizar el siguiente método de la clase

`Connection:`

```
int con.getTransactionIsolation();
```

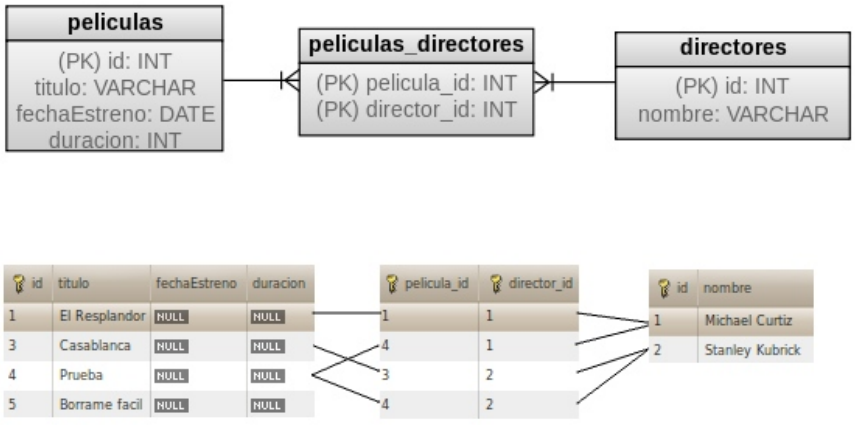
Este método devolverá Connection.`TRANSACTION_NONE` si el SGBD no soporta transacciones. En caso de que si que las soporte, nos dirá el nivel de aislamiento, pudiendo ser éste (ordenado de menor a mayor aislamiento) Connection.`TRANSACTION_READ_UNCOMMITTED`, Connection.`TRANSACTION_READ_COMMITTED`, Connection.`TRANSACTION_REPETEABLE_READ`, Connection.`TRANSACTION_SERIALIZABLE`.

Cuanto mayor sea el nivel de aislamiento más posibles casos de concurrencia se estarán teniendo en cuenta y por lo tanto menos problemas podrán ocurrir, pero esto también producirá un mayor número de bloqueos en los accesos y por lo tanto una menor eficiencia. Un posible problema en las transacciones es el interbloqueo. Un interbloqueo se produce en la siguiente situación: una aplicación tiene que modificar dos registros. Otra aplicación modifica los mismos, pero en orden inverso. Se empiezan a ejecutar las dos aplicaciones a la vez y al haber modificado un registro no dejan que la otra lo modifique. Sin embargo, ninguna de las dos terminan porque están esperando que se desbloquee su registro. En caso de que esto ocurra, el SGBD debería detectar la situación y lanzar una excepción.

14. Ejercicios de Java Database Connectivity

14.1. Filmoteca en MySQL

Vamos a implementar una parte de la base de datos de la filmoteca de los anteriores ejercicios: la tabla de películas y la lista de directores de la película. Dejaremos para otro momento la tabla de actores. Cada película puede tener varios directores, y cada director puede haber dirigido varias películas. Esta es una relación "muchos a muchos" y lo normal es implementar este tipo de relacion con una tabla intermedia, al tratarse de un modelo relacional de base de datos.



Tablas películas y directores con tabla intermedia para la relación 'muchos a muchos'. Arriba: modelo Entidad-Relación. Abajo: datos de ejemplo y sus relaciones de clave ajena.

En MySQL conviene utilizar el motor InnoDB en lugar del que se asigna por defecto a cada tabla, para permitir el uso de claves ajenas. En nuestro caso la tabla intermedia estará compuesta por dos columnas, ambas formando la clave principal (es decir, no se pueden repetir), y a la vez cada una de ellas tendrá una restricción de clave ajena a las tablas de películas y de directores, respectivamente. Esto significa que en la tabla intermedia no puede haber un valor que no esté asociado a un identificador existente en la tabla de películas o de directores. Nótese que estas restricciones imponen un orden a la hora de insertar en la base de datos, así como a la hora de borrar.

La base de datos descrita está incluida en las plantillas de la sesión, en la carpeta `db/`. Inclúyase esta carpeta en el proyecto, ya que contiene un script de ANT que conecta con la base de datos (en él hay que modificar la dirección, el nombre de usuario y la

contraseña) y ejecuta el script `filmoteca.sql` que contiene las sentencias de creación de la base de datos. Si ésta ya existe, los datos son eliminados para introducir los datos de ejemplo de la figura. Ejecútese el script Ant para comprobar a través de algún cliente de MySQL que los datos se han creado con éxito.

Para que el proyecto cuente con las librerías del driver JDBC para MySQL, copiamos a su carpeta `lib` el conector proporcionado en las plantillas, `mysql-connector-java-5.1.5-bin.jar` y lo incluimos en el build path del proyecto.

14.2. Conectar con la base de de datos (0.5 puntos)

En las plantillas de la sesión está incluido un fichero fuente para usarlo como esqueleto para la implementación del DAO JDBC. Se pide implementar en el constructor la comprobación (en tiempo de ejecución) de que la clase `com.mysql.jdbc.Driver` existe y de lo contrario, capturar la excepción y relanzarla como nested, dentro de una nueva `DAOException`. Puede que haga falta alguna modificación en el resto del código fuera del DAO, para manejar la excepción.

Cambiar en el `GestorDAO` la creación del DAO, para que se haga uso del `JDBCPeliculaDAO`. Si ahora ejecutamos el programa, aunque los métodos del DAO todavía no estén implementados, el programa deberá fallar en tiempo de ejecución si no encuentra el Driver de MySQL.

En los tres métodos del DAO se pide establecer la conexión a la base de datos y después cerrarla. Dependiendo de la aplicación puede ser conveniente cerrar las conexiones cada vez que se terminen de usar, ya que los servidores admiten un número limitado de conexiones simultáneas. Sólo conviene mantenerlas abiertas si se sabe que se van a seguir utilizando. Se pide abrir las conexiones en la primera línea de los `try` de cada uno de los tres métodos del DAO. Hará falta la dirección de la base de datos, el usuario y la contraseña, todos ellos vienen como constantes en la clase. También se pide cerrar la conexión en los bloques `finally` de los métodos.

14.3. Consulta sin parámetros (0.5 puntos)

Para el método `JDBCPeliculaDAO.getAllPeliculas()` se pide crear una sentencia no preparada, `Statement` y ejecutar la query (viene como constante en la clase):

```
SELECT * FROM peliculas ORDER BY id;
```

A continuación en el bucle while de la plantilla, recoger los datos (`id`, `titulo`, `duracion`, `fechaEstreno`) del `ResultSet` mientras los haya, introduciéndolos en el objeto `PeliculaTO` y añadiendo el resultado a la lista de películas `List<PeliculaTO> lista`. La consulta de la lista de directores se hará en el apartado siguiente, de momento podemos ejecutar el programa y comprobar que listar las películas funciona.

14.4. Sentencias preparadas (0.5 puntos)

Las sentencias preparadas deben utilizarse cuando la consulta tiene parámetros para evitar que por error o malintencionadamente se inyecten sentencias no deseadas o simplemente ocurra algún error de SQL. En el apartado anterior se consultan todas las películas de la tabla `peliculas` pero no se carga la lista de directores correspondiente a la película. Lo podemos hacer mediante la siguiente sentencia SQL:

```
SELECT nombre FROM directores d, peliculas_directores pd
  WHERE (d.id = pd.director_id) AND pd.pelicula_id = ?;
```

donde el signo de interrogación `"?"` es un parámetro. Introdúzcase en una sentencia preparada la sentencia SQL (viene como constante en la plantilla) y el parámetro `id` de la película. El resultado deberá recorrerse e introducir los nombres de los directores en la lista de directores del objeto `PeliculaTO`. Con esto hemos completado el método `getAllPeliculas()` y pero la interfaz de usuario proporcionada por la clase `Main` no nos permite comprobar que funcione correctamente. Para comprobar que los directores se cargan correctamente se puede recurrir al debugger, al log, o a la salida estándar.

14.5. Sentencias de borrado de registros (0.5 puntos)

Los métodos `delPelicula(int)` y `addPelicula(PeliculaTO)` ya deberían contener el código de establecer y cerrar la conexión JDBC. Ahora se pide:

En `delPelicula(int)` preparar y ejecutar las sentencias:

```
DELETE FROM peliculas_directores WHERE pelicula_id = ?;
DELETE FROM peliculas WHERE id = ?;
```

¿En qué orden deberían ejecutarse, dada la restricción de clave ajena que hay?

En la sentencia de borrado de películas se pide comprobar cuántos registros han sido afectados y si son cero, lanzar una `DAOException`.

Sin introducir transaccionalidad todavía, se pide comprobar que el borrado funciona ejecutando el programa.

Nótese que no eliminamos el director de la tabla de directores. En el supuesto de que hubiera más información asociada a cada director, dejarlos almacenados podría ser una estrategia adecuada. En caso de querer eliminarlos, la clave ajena de la tabla intermedia nos restringiría borrar un director que todavía esté referenciado por otras películas.

14.6. Sentencias de inserción de registros y valores autoincrementados (0.5 puntos)

En `addPelicula(PeliculaTO)` preparar y ejecutar la sentencia:

```
INSERT INTO peliculas(id,titulo,fechaEstreno,duracion) VALUES(?,?,?,?)
```

Si ha afectado a menos de un registro, lanzar una `DAOException`. Cerrar esta sentencia preparada y antes de continuar con el método, probar si funciona.

Para insertar la lista de directores en la base de datos habrá que utilizar varias veces las sentencias

```
SELECT id FROM directores WHERE nombre = ?;
INSERT INTO directores(nombre) VALUES(?);
INSERT INTO peliculas_directores(pelicula_id,director_id) VALUES(?,?);
```

Por tanto las prepararemos antes del bucle que recorre la lista de directores, y cada vez que las vayamos a utilizar limpiaremos sus parámetros con el método `.clearParameters()`.

- La primera de ellas obtiene el identificador del director.
- La segunda se ejecutará sólo si el director no existía. Nótese que la tabla de directores tiene dos columnas pero sólo insertamos el nombre. El identificador de esta tabla está marcado como autoincremental y será generado. Necesitamos obtener el valor del `id` generado con el método `.getGeneratedKeys()`, como está hecho en el código de la plantilla.
- Finalmente con la id obtenida de una de las dos sentencias anteriores, se ejecuta la tercera sentencia que inserta en la tabla intermedia la asociación entre identificador de película e identificador de director.

Una vez finalizado el bucle se pueden cerrar las sentencias preparadas. Se pide completar el código de la plantilla y, sin introducir transaccionalidad, probar que funciona.

14.7. Transacciones (0.5 puntos)

Las transacciones no serían necesarias en el ejemplo anterior puesto que la lógica del programa ya se encarga de mantener la coherencia entre tablas. La transaccionalidad suele ser necesaria en bases de datos en las que varios clientes modifican las tablas de la base de datos y no se debe permitir que un insertado funcione con una tabla pero no funcione con la siguiente. En este caso se deshace la operación entera.

Para los métodos `delPelicula(int)` y `addPelicula(PeliculaTO)` se pide:

- Una vez establecida la conexión, desactivarle el `autocommit`.
- Una vez finalizado el borrado / inserción, respectivamente, forzar el `.commit()`.
- Si algo falla, en el bloque `catch` deshacer la transacción con `.rollback()` y lanzar la excepcion `DAOException`.

15. Pruebas con DBUnit

15.1. Introducción

El correcto acceso a datos es fundamental en cualquier aplicación. La complejidad de algunos modelos de datos crea la necesidad de pruebas sistemáticas de la capa de datos. Por un lado se necesita probar que la capa de acceso a datos genera el estado correcto en la base de datos (BD). Por otro lado también se necesita probar que ante determinado estado de la BD el código Java se comporta de la manera esperada.

Sistematizar estas pruebas requiere una manera sencilla de reestablecer el estado de la base de datos. De otra manera surgirían problemas cada vez que un test falle y deje la BD en un estado inconsistente para los siguientes tests.

15.2. DbUnit

DbUnit es un framework de código abierto que fue creado por Manuel Laflamme. Está basado en JUnit, de hecho sus clases extienden el comportamiento de las clases de JUnit. Eso permite la total integración con el resto de pruebas en JUnit.

DbUnit nos permite gestionar el estado de una BD durante las pruebas unitarias y de integración. Constituye la alternativa a la creación de clases stub y mocks para controlar las dependencias externas.

15.2.1. Ciclo de vida

El ciclo de vida de las pruebas con DbUnit es el siguiente:

1. Eliminar el estado previo de la BD resultante de pruebas anteriores (en lugar de restaurarla tras cada test).

2. Cargar los datos necesarios para las pruebas de la BD (sólo los necesarios para cada test).

3. Utilizar los métodos de la librería DbUnit en las aserciones para realizar el test.

15.2.2. Características de DbUnit

DbUnit nos permite los problemas que surgen si el último caso de prueba ha dejado la base de datos inconsistente.

DbUnit puede trabajar con conjuntos de datos grandes.

Permite verificar que el contenido de la base de datos es igual a determinado conjunto de

datos esperado, a nivel de fichero, a nivel de consulta o bien a nivel de tabla.

Nos proporciona un mecanismo basado en XML para cargar los datos en la BD y para exportarlos desde la BD.

Proporciona una forma de aislar los experimentos en distintos casos de prueba individuales, uno por cada operación.

15.3. Prácticas recomendadas

La documentación de DbUnit establece una serie de pautas como prácticas recomendadas o prácticas adecuadas (best practices).

- Usar una instancia de la BD por cada desarrollador. Así se evitan interferencias entre ellos.
- Programar las pruebas de tal manera que no haya que restaurar el estado de la base de datos tras el test. No pasa nada si la base de datos se queda en un estado diferente tras el test. Dejarlo puede ayudar para encontrar el fallo de determinada prueba. Lo importante es que la base de datos se pone en un estado conocido antes de cada test.
- Usar múltiples conjuntos de datos pequeños en lugar de uno grande. Cada prueba necesita un conjunto de tablas y de registros, no necesariamente toda la base de datos.
- Inicializar los datos comunes sólo una vez para todos los tests. Si hay datos que sólo son de lectura, no tenemos por qué tocarlos si nos aseguramos de que las pruebas no los modifican.

Estrategias de conexión recomendadas.

- Como cliente remoto con la clase `DatabaseTestCase`.
- A través del pool de conexiones de un servidor de aplicaciones, con `IDataBaseConnection o con JndiBasedDBTestCase.`

15.4. Clases e interfaces

DbUnit es un conjunto de clases, algunas de las cuáles heredan de clases de JUnit. DbUnit tiene una serie de dependencias con las siguientes librerías (bibliotecas):

- JUnit: framework base para las pruebas
- Jakarta Commons IO: utilidades de entrada y salida
- Slf4j: frontend para frameworks de logging

Nosotros utilizaremos las versiones JUnit 4, Commons IO 1.4 y Slf4j 1.6. Lógicamente también es necesaria la librería para el acceso a la base de datos, pero esto ya no es una dependencia de DbUnit, sino de nuestro proyecto. En nuestro caso utilizaremos el conector JDBC de MySQL.

`DBTestCase` hereda de la clase `TestCase` de JUnit y proporciona métodos para inicializar y restaurar la BD antes y después de cada test. Utiliza la interfaz `IDatabaseTester` para

conectar con la BD. A partir de la versión 2.2 de DbUnit se ha pasado a la alternativa de utilizar directamente el `IDatabaseTester`, como se verá en los ejemplos.

`IDatabaseTester` devuelve conexiones a la base de datos, del tipo `IDatabaseConnection`. La implementación que nosotros vamos a utilizar es la de JDBC, `JdbcDatabaseTester`. Tiene métodos `onSetUp()`, `setSetUpOperation(op)`, `onTearDown()`, `setTearDownOperation(op)`, `getConnection()`, entre otros.

`IDatabaseConnection` es la interfaz a la conexión con la base de datos y cuenta con métodos para crear un conjunto de datos `createDataSet()`, crear una lista concreta de tablas, `createDataSet(listaTablas)`, crear una tabla extraída de la base de datos, `createTable(tabla)`, crear una tabla con el resultado de una query sobre la BD con `createQueryTable(tabla, sql)`, y otros métodos como `getConnection()`, `getConfig()`, `getRow()`.

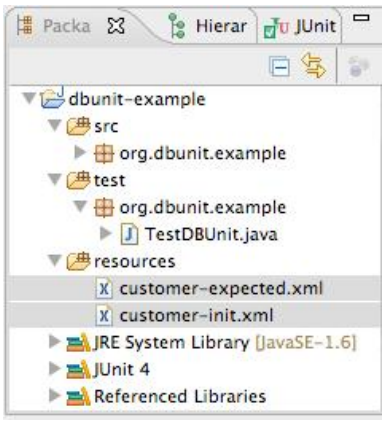
La interfaz `IDataSet` representa una colección de tablas y se utiliza para situar la BD en un estado determinado, así como para comparar el estado actual de la BD con el estado esperado. Dos implementaciones son `QueryDataSet` y `FlatXmlDataSet`. Esta última implementación sirve para importar y exportar conjuntos de datos a XML en un formato como el siguiente:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
<TEST_TABLE COL0="row 0 col 0"
COL1="row 0 col 1"
COL2="row 0 col 2"/>
<TEST_TABLE COL1="row 1 col 1"/>
<SECOND_TABLE COL0="row 0 col 0"
COL1="row 0 col 1" />
<EMPTY_TABLE/>
</dataset>
```

Por último otra clase importante es la `Assertion` que define los métodos estáticos para realizar las comparaciones: `assertEquals(IDataSet, IDataSet)` y `assertEquals(ITable, ITable)`

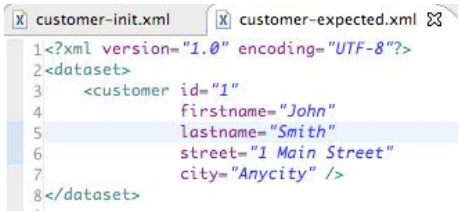
15.5. Estructura en un proyecto

La estructura del proyecto es similar a la que podemos usar con JUnit, aislando las pruebas en una carpeta separada `test`.

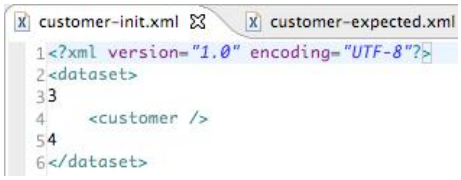


Estructura del proyecto en Eclipse

Con DbUnit también necesitaremos una carpeta de recursos `resources` donde guardar los recursos XML. Éstos consistirán en ficheros independientes para cada prueba, tanto para la inicialización de la BD, como para el estado esperado de la BD tras cada prueba:



XML con el estado de inicialización de la BD



XML con el estado esperado de la BD tras el test

La estructura básica de la clase que contiene un conjunto de pruebas consiste en implementar los métodos que se ejecutan antes de cada test, después, y los tests en sí. También habría que incluir las variables de instancia del DAO y del `IDatabaseTester`:

```
public class TestMiJDBCDAO {
        private JDBCDAO dao;
        private IDatabaseTester databaseTester;

        @Before
        public void setUp() throws Exception {

        }

        @After
        public void tearDown() throws Exception {
```

```
        }

        @Test
        public void test1DelAdd() throws Exception {

        }
        ...
}
```

Obtener la instancia al DAO se podría hacer o bien antes de cada test, o bien antes de todos los tests, con una función anotada como `@BeforeCase`. Antes de cada prueba también hay que configurar el `IDatabaseTester`, en este caso para acceder por JDBC, e inicializar el conjunto de datos a partir del recurso XML apropiado. Finalmente hay que llamar la operación `databaseTester.onSetup()` que llama a la operación por defecto de inicio de test de la clase `DatabaseOperation`. Ésta define el contrato de la interfaz para operaciones realizadas en la base de datos.

```
        @Before
        public void setUp() throws Exception {
                //Obtener instancia del DAO que testeamos
                pdao = new JDBCPeliculaDAO();

                //Acceder a la base de datos
                databaseTester = new
JdbcDatabaseTester("com.mysql.jdbc.Driver",
                "jdbc:mysql://localhost/databasename", "username",
"password");

                //Inicializar el dataset en la BD
                FlatXmlDataSetBuilder builder = new
FlatXmlDataSetBuilder();
            IDataSet dataSet = builder.build(
              this.getClass().getResourceAsStream("/db-init.xml"));
            databaseTester.setDataSet(dataSet);

                //Llamar a la operación por defecto setUpOperation
                databaseTester.onSetup();
        }
```

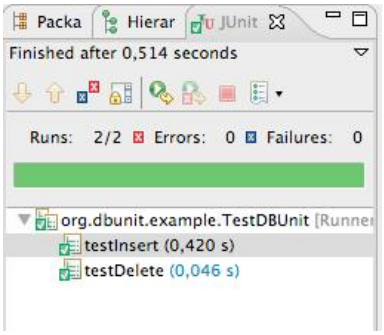
Similarmente llamaremos a `onTearDown()` después de cada prueba:

```
        @After
        public void tearDown() throws Exception {
                databaseTester.onTearDown();
        }
```

La prueba se aísla en un método en el que realizamos la operación a través del DAO, conectamos con la base de datos a través de `IDatabaseConnection`, creamos el dataset de las tablas implicadas en el test, y lo escribimos en un archivo en formato XML. Obtenemos los datos esperados de otro archivo XML y los comparamos. Deben ser idénticos para que la prueba tenga éxito.

```
        @Test
        public void test1() throws Exception {
                //Realizar la operación con el JDBC DAO
                ...

                // Conectar a la base de datos MySQL
                IDatabaseConnection connection =
databaseTester.getConnection();
                DatabaseConfig dbconfig = connection.getConfig();
                dbconfig.setProperty(
"http://www.dbunit.org/properties/datatypeFactory",
                                new MySqlDataTypeFactory());

                // Crear el DataSet de la base de datos MySQL
                QueryDataSet partialDataSet = new
QueryDataSet(connection);
                partialDataSet.addTable("tabla1");
                partialDataSet.addTable("tabla2");
                partialDataSet.addTable("tabla3");

                // Escribir el DataSet en XML para después compararlo con
el esperado
                File outputFile = new File("db-output.xml");
                FlatXmlDataSet.write(partialDataSet, new
FileOutputStream(outputFile));

                // Obtener los datos esperados del XML
                URL url =
IDatabaseTester.class.getResource("/db-expected1.xml");
                Assert.assertNotNull(url);
                File inputFile = new File(url.getPath());

                // Comparar los ficheros XML
                Assert.assertEquals(FileUtils.readFileToString(inputFile,
"UTF8"),
                                FileUtils.readFileToString(outputFile,
"UTF8"));
        }
```

Por último, las pruebas se ejecutan a través de JUnit:



Resultado de la ejecución de las pruebas.

Se puede probar el siguiente ejemplo de proyecto con DbUnit en Eclipse: DBUnitExample.zip.

16. Ejercicios de DbUnit

16.1. Generar el XML de DbUnit (0.5 puntos)

Vamos añadir tests de DbUnit al proyecto de la filmoteca para probar los métodos del `JDBCPeliculaDAO`. Copiamos los .jar proporcionados en las plantillas de la sesión a la carpeta `lib` del proyecto y los importamos en el proyecto. Para comprobar que funcionan añadimos al proyecto el archivo `TestDataExtractor.java` y lo ejecutamos. Debe generar un archivo XML como el siguiente:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <peliculas id="1" titulo="El Resplandor"/>
  <peliculas id="3" titulo="Casablanca"/>
  <peliculas id="5" titulo="Borrame facil"/>
  <directores id="1" nombre="Michael Curtiz"/>
  <directores id="2" nombre="Stanley Kubrick"/>
  <peliculas_directores pelicula_id="1" director_id="1"/>
  <peliculas_directores pelicula_id="3" director_id="2"/>
</dataset>
```

Podemos guardar este archivo como `resources/db-init.xml` para utilizarlo después como punto de partida de los tests de DBUnit.

16.2. Clase para los test de DbUnit (1 punto)

En la carpeta de fuentes `test`, en el paquete `es.ua.jtech.lja.filmoteca.dao` creamos una nueva clase llamada `TestJDBCPeliculaDAO`. En sus métodos utilizaremos el `JDBCPeliculaDAO` y un objeto `IDatabaseTester` así que los declaramos como campos privados de la clase.

Inicializaremos dichos campos en el método `@Before public void setUp()`. En el mismo método inicializaremos el dataset de la base de datos para empezar siempre los tests desde el mismo estado:

```
FlatXmlDataSetBuilder builder = new FlatXmlDataSetBuilder();
IDataSet dataSet =
builder.build(this.getClass().getResourceAsStream("/db-init.xml"));
databaseTester.setDataSet(dataSet);
```

La última operación del método `setUp()` será la llamada a `.onSetup()` del objeto de tipo `IDatabaseTester`.

A su vez, el método `@After public void tearDown()` llamará a `.onTearDown()` del objeto de tipo `IDatabaseTester`.

16.3. Test de borrado y añadido (0.5 puntos)

Los test deben llevar la anotacion `@Test`. Crear el primero de ellos llamado `test1DelAdd()`. Este test deberá utilizar el DAO obtener todas las películas en una `List`, después recorrerlas en ese mismo orden eliminándolas de la base de datos y finalmente insertarlas una a una también en el mismo orden.

Conectar a la base de datos con el `IDatabaseConnection` para crear a partir de la conexión el `QueryDataSet` con las tablas películas, directores y películas_directores. Una vez creado el set de datos, escribirlo en un nuevo fichero `db-output-data.xml`. Comparar utilizando `Assert.assertEquals( )` el XML generado con el del archivo `db-expected1.xml` que deberá estar en la carpeta de recursos y contener exactamente lo mismo que el `db-init.xml`.

16.4. Test que espera una excepción (0.5 puntos)

Crear el segundo test llamado `test2AddExisting()`. Este test deberá utilizar el DAO para añadir una película existente. Se puede proceder obteniendo la lista de películas del DAO e intentando insertar una de ellas. Según la especificación, deberá saltar una `DAOException`. Utilizar la notación

```
@Test(expected=DAOException.class)
```

para que el test falle si no se produce exactamente esa excepción.

16.5. Test de borrado (0.5 puntos)

Crear el tercer test llamado `test3Del()`. Este test deberá utilizar el DAO para eliminar la película con identificador `id==4`. El XML esperado se llamará `db-expected3.xml` y deberá contener todas las películas iniciales menos la 4.

Al ejecutar los tests como JUnit deberán dar luz verde.