Introducción a Java

Índice

1 El lenguaje Java....3
1.1 Conceptos previos de POO....3
1.2 Componentes de un programa Java....5
1.3 Herencia e interfaces....14
1.4 Clases útiles....17
1.5 Estructuras de datos....23
2 Colecciones de datos....24
2.1 Colecciones....25
2.2 Polimorfismo e interfaces....34
2.3 Tipos de datos básicos en las colecciones....35
3 Excepciones....36
3.1 Introducción....36
3.2 Tipos de excepciones....36
3.3 Captura de excepciones....38
3.4 Lanzamiento de excepciones....39
3.5 Creación de nuevas excepciones....40
3.6 Nested exceptions....40
4 Serialización de datos....41
4.1 Introducción....42
4.2 Flujos de datos de entrada/salida....42
4.3 Entrada, salida y salida de error estándar....43
4.4 Acceso a ficheros....44
4.5 Acceso a los recursos....45
4.6 Acceso a la red....46
4.7 Codificación de datos....46
4.8 Serialización de objetos....47

5 Depuración con Eclipse.....48
5.1 Primeros pasos para depurar un proyecto.....49
5.2 Establecer breakpoints.....49
5.3 Evaluar expresiones.....51
5.4 Explorar variables.....52
5.5 Cambiar código "en caliente".....52
5.6 Gestión de Logs.....53

1. El lenguaje Java

Java es un lenguaje de programación creado por *Sun Microsystems* 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. 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).

1.1.1. 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.

1.1.2. 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.

1.1.3. 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**.

1.1.4. 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.

1.1.5. 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.

1.2. Componentes de un programa Java

En un programa Java podemos distinguir varios elementos:

1.2.1. 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`.

1.2.2. 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`.

1.2.3. 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.

1.2.4. 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...*

1.2.5. 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 `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*.

1.2.6. 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"`).

1.2.7. 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();
```

1.2.8. 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();
```

1.2.9. 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(...)
        }
```

1.2.10. 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");
```

1.2.11. 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.

1.2.12. 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`.

1.3. Herencia e interfaces

1.3.1. 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.

1.3.2. 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.

1.3.3. 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*.

1.4. 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:

1.4.1. 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;
                }
        }
```

1.4.2. 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.

1.4.3. 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

1.4.4. 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.

1.4.5. 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*.

1.4.6. 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.

1.5. 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.

2. 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.

2.1. 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.

2.1.1. 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.

2.1.2. 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)*.

2.1.3. 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.

2.1.4. 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)
```

2.1.5. 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.

2.1.6. 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);
```

2.2. 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.

2.2.1. 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.

2.3. Tipos de datos básicos en las colecciones

2.3.1. 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.

2.3.2. 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;
```

3. Excepciones

3.1. Introducción

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.

3.2. 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:

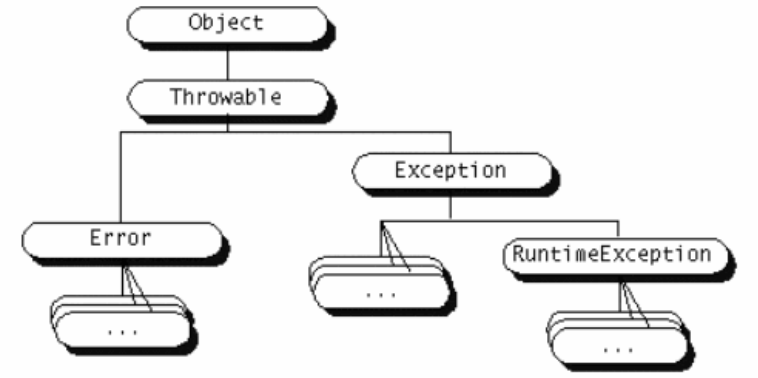
- **`Error`**: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **`Exception`**: Representa errores que no son críticos y por lo tanto pueden ser tratados y

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.

3.3. 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*.

3.4. 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.

3.5. 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.

3.6. 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.

4. Serialización de datos

4.1. 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.

4.2. 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.

4.3. 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*.

4.4. 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();
          }
        }
```

4.5. 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.

4.6. 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.

4.7. 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.

4.8. 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.

5. 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

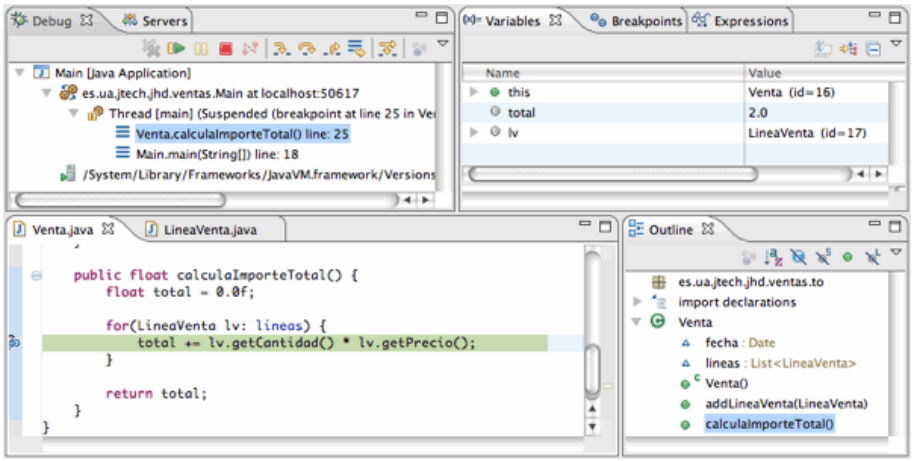
provoque nuestro programa. Este depurador incorpora muchas funcionalidades, como la ejecución paso a paso, establecimiento de *breakpoints*, consulta y establecimiento de valores de variables, parar y reanudar hilos de ejecución, etc. También podremos depurar aplicaciones que se estén ejecutando en máquinas 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.

5.1. 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

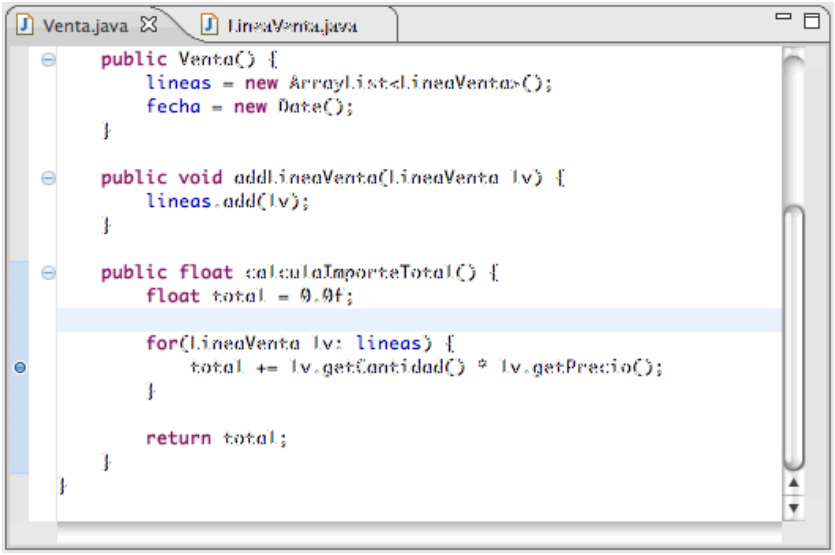
En la parte superior izquierda vemos los hilos que se ejecutan, y su estado. Arriba a la derecha vemos los *breakpoints* que establezcamos, y los valores de las variables que entran en juego. Después tenemos el código fuente, para poder establecer marcas y *breakpoints* en él, y debajo la ventana con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

5.2. Establecer breakpoints

Una de las operaciones más habituales a la hora de depurar es establecer *breakpoints*,

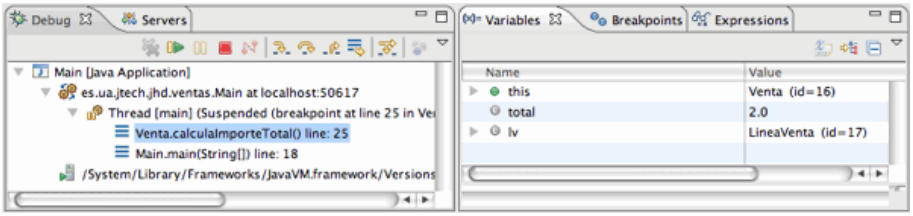
puntos en los que la ejecución del programa se detiene para permitir al programador examinar cuál es el estado del mismo en ese momento.

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**.



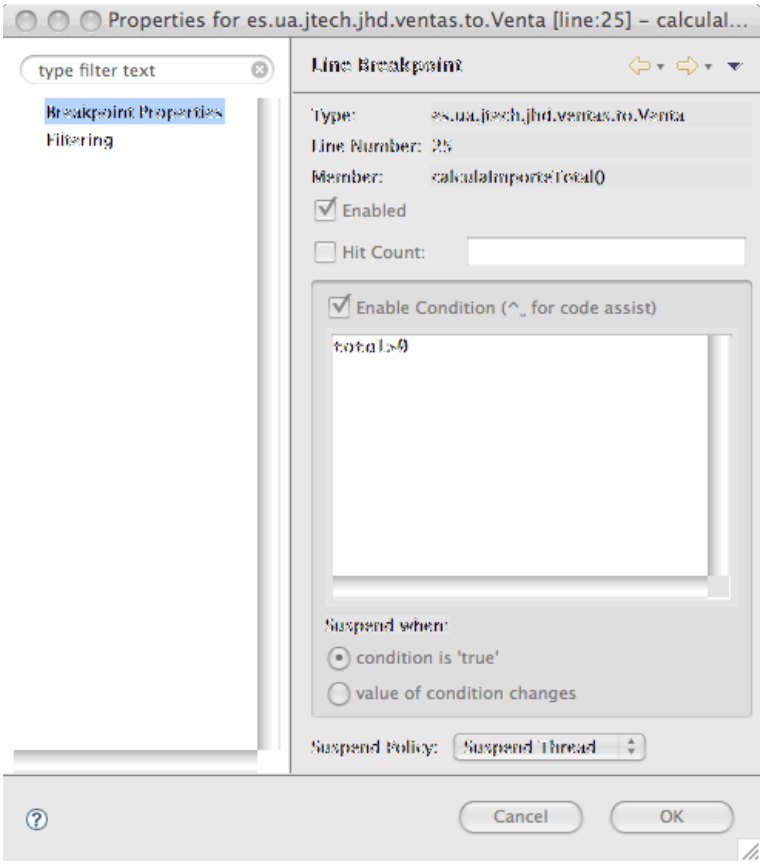
Control del estado del programa

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*

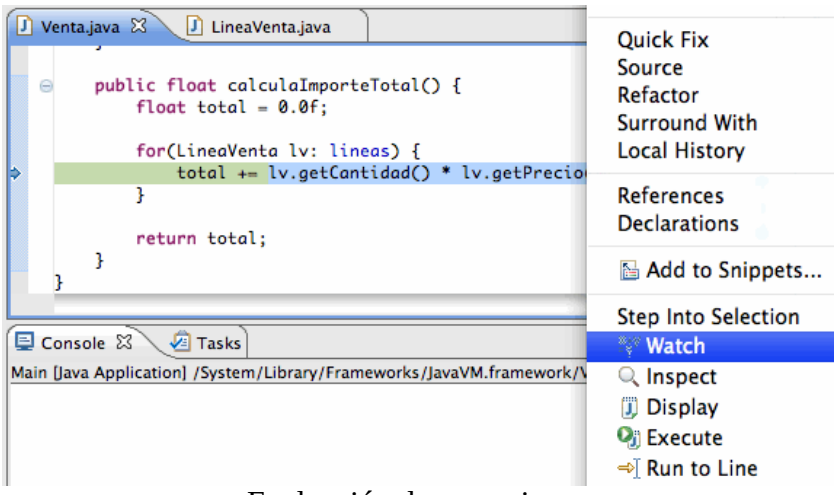
cuando la condición sea cierta, o cuando el valor de esa condición cambie:



Breakpoints condicionales

5.3. 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.

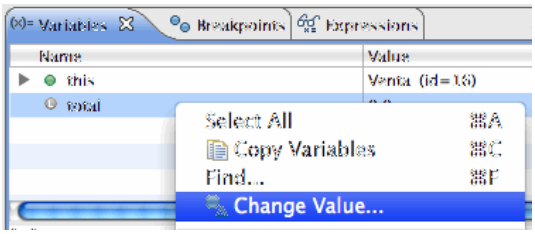


Evaluación de expresiones

5.4. 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

5.5. 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.

5.6. Gestión de Logs

Aunque el debugging con Eclipse parece sencillo y eficaz, no siempre es posible utilizarlo. Algunos casos de difícil depuración son los de aplicaciones distribuidas, aplicaciones multihilo, o aplicaciones en dispositivos embebidos. En cualquier caso utilizar logs para informar de excepciones, advertencias, o simplemente información útil para el programador, es una práctica habitual y recomendable.

Log4Java (log4j) es una librería open source de Jakarta 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.

Log4Java se puede combinar con el envoltorio que proporciona la librería commons-logging, también de Jakarta, para proporcionar todavía mayor flexibilidad.