

Guía de Java

BORRADOR © D.G.S. - UOC

Versión preliminar – Abril 2020 © David García Solórzano

Las versiones definitivas en español y catalán estarán disponibles en las aulas de la UOC en septiembre de 2020.

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea este eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares de los derechos.

Contenido

1.	Introducción	9
1.1.	Breve historia	9
1.2.	Ediciones de Java	9
1.3.	Ciclo de vida de un programa Java	10
1.4.	JRE vs JDK	11
2.	Configuración del entorno	12
2.1.	Instalando JDK	12
2.1.1.	OpenJDK vs Oracle JDK	12
2.1.2.	Oracle JDK para Windows	13
2.1.3.	Oracle JDK para Linux	17
2.1.4.	Oracle JDK para macOS	17
2.2.	Compilando y ejecutando un programa por línea de comandos	17
2.3.	Eclipse IDE	18
2.3.1.	Instalación de Eclipse IDE	18
2.3.2.	Vistas	21
	Explorador de paquetes (package explorer)	21
	Editor	22
	Outline	22
	Problems	23
	Console	23
2.3.3.	Menús	23
	Menú File	23
	Menú Project	25
	Menú Run	25
	Otros menús	26
3.	Fundamentos de Java (como lenguaje imperativo)	27
3.1.	Comentarios dentro del código	27
3.2.	Estructura básica de un programa	27
3.3.	Entrada por teclado y salida por pantalla (I/O)	30
3.3.1.	Métodos de salida	30
	Métodos print y println	30
	Método printf	31
3.3.2.	Métodos de entrada	31

3.4. Variables.....	32
3.4.1. Nombre.....	32
3.4.2. Tipos.....	33
Primitivos.....	33
Referencias.....	37
3.5. Operadores.....	42
3.5.1. Operadores de asignación, aritméticos y unarios	43
Asignación simple.....	43
Operadores aritméticos	44
Asignaciones aumentadas o compuestas	46
Operadores unarios.....	46
3.5.2. Operadores de igualdad, relacionales y condicionales	47
Operadores de igualdad y relacionales.....	47
Operadores condicionales.....	48
Operador instanceof	49
3.5.3. Operadores bitwise y shiftado.....	49
Operadores bitwise	49
Operadores de shiftado	50
3.6. Bloques de flujo de ejecución.....	50
3.6.1. Bloque secuencial	51
3.6.2. Bloque condicional	51
Bloque if.....	51
Bloque if-else.....	51
Bloque if anidado o else-if	52
Operador ternario	52
Bloque switch	52
3.6.3. Bloque iterativo	53
Bloque while.....	53
Bloque do-while	54
Bloque for	54
Bloque enhanced for.....	55
3.6.4. Instrucciones de control de flujo.....	56
Instrucción break.....	56
Instrucción continue.....	57

Instrucción <code>return</code>	57
3.7. <code>String</code>	58
3.7.1. <i>Escape sequences</i>	58
3.7.2. Concatenar con <code>+</code>	59
3.7.3. Algunos métodos interesantes.....	59
3.7.4. Conversiones	60
De tipo primitivo a <code>String</code>	60
De tipo <code>String</code> a tipo primitivo	61
3.8. <code>Arrays</code>	61
3.8.1. Declaración y creación en memoria	62
3.8.2. Tamaño de un <i>array</i>	63
3.8.3. Copiar el contenido de un <i>array</i> a otro <i>array</i>	64
Método <code>clone</code>	64
Método <code>arraycopy</code>	65
Método <code>copyOf</code>	65
Método <code>copyOfRange</code>	65
3.8.4. Ordenar un <i>array</i>	65
3.9. Métodos (funciones)	66
3.9.1. Definición de un método	66
3.9.2. Sobrecarga de un método	67
3.9.3. Usando un método	68
3.9.4. Paso por valor y por referencia	69
3.9.5. Número variable de parámetros	69
3.9.6. <i>Cast</i> implícito en los parámetros	71
3.10. Ámbito de las variables	71
4. Orientación a objetos en Java	74
4.1. Definiendo una clase	74
4.1.1. Estructura mínima	74
4.1.2. Miembros de la clase.....	74
Atributos.....	75
Métodos	76
4.1.3. Constructor	78
Constructor por defecto.....	78
Constructor con argumentos	79

Método especial <code>this</code>	81
Bloque de inicialización de instancia.....	82
4.2. Objetos	83
4.2.1. Instanciar (creando objetos).....	83
4.2.2. Usando un objeto (mensajes).....	84
4.3. Modificadores de acceso.....	84
4.4. Destructor de una clase.....	89
4.5. Packages	90
4.5.1. ¿Qué es un <i>package</i> y cuál es su utilidad?	90
4.5.2. Creando un <i>package</i>	92
4.5.3. Usando los elementos incluidos dentro de un <i>package</i>	94
Importar el elemento a usar	94
Importar el package entero.....	94
Usando el fully qualified name del elemento	96
4.5.4. <i>Packages</i> importados automáticamente por Java	96
Importación implícita del propio package	96
Importación implícita del package <code>java.lang</code>	97
4.6. Modificador <code>static</code>	97
4.6.1. Atributo estático	97
4.6.2. Bloque de inicialización estático.....	97
4.6.3. Método estático	99
4.6.4. Clase estática	99
4.7. Modificador <code>abstract</code>	100
4.7.1. Clase abstracta.....	100
4.7.2. Método abstracto	100
4.8. Modificador <code>final</code>	100
4.8.1. Clase final.....	100
4.8.2. Método final	101
4.8.3. Atributo final.....	101
4.9. Herencia	102
4.9.1. Concepto y sintaxis	102
4.9.2. La clase <code>Object</code>	102
4.9.3. ¿Qué hereda una subclase?.....	102
4.9.4. El modificador <code>abstract</code> y la herencia	107

4.9.5. El modificador <code>static</code> y la herencia	107
4.9.6. El modificador <code>final</code> y la herencia.....	108
4.9.7. Ocultación de atributos	108
4.10. Interfaces.....	108
4.10.1. Concepto y sintaxis	108
4.10.2. Uso de una interfaz.....	109
4.10.3. ¿Qué elementos puede contener una interfaz?	109
4.11. Polimorfismo	112
4.12. Excepciones	115
4.12.1. Concepto.....	115
4.12.2. Declarando y lanzando una excepción	116
4.12.3. Flujo que sigue una excepción.....	118
4.12.4. Tratando/capturando una excepción	119
4.12.5. <i>Checked</i> y <i>unchecked exceptions</i>	121
4.12.6. Declarando, lanzando y capturando más de una excepción	122
4.12.7. Bloque <code>finally</code>	124
4.12.8. Métodos del exception object.....	125
<code>getMessage</code>	125
<code>toString()</code>	125
<code>printStackTrace</code>	125
4.12.9. Sentencia <code>try-with-resources</code>	125
4.13. Enumeraciones.....	126
4.13.1. Concepto y sintaxis	126
4.13.2. Diferencia entre usar constantes y un <i>enumeration</i>	126
4.13.3. Compatibilidad con <code>switch</code>	127
4.13.4. Implementación interna de las constantes (de los valores).....	128
4.13.5. Herencia de la clase <code>Enum</code>	128
4.13.6. Constructor y miembros	128
4.13.7. Métodos abstractos.....	129
4.14. Generics.....	130
4.14.1. Concepto.....	130
4.14.2. Clase genérica	130
4.14.3. Ventajas de <i>generics</i>	131
4.14.4. Interfaz genérica	132

4.14.5. Método genérico	133
4.14.6. Parámetros de tipo	134
4.14.7. Subtipos	134
4.14.8. Usando tipos parametrizados como parámetro de tipo	135
4.14.9. Uso del <i>wildcard</i> ?	135
4.14.10. Limitación de tipos	135
4.14.11. Restricciones en el uso de <i>generics</i>	138
4.15. Colecciones.....	139
4.15.1. Interfaz Set.....	140
4.15.2. Interfaz List	140
4.15.3. Interfaz Map.....	141
4.15.4. Resumen	143
5. Avanzado	145
5.1. La clase String	145
5.1.1. StringBuilder vs StringBuffer	147
5.1.2. Resumen	147
5.2. Programación funcional (expresiones lambda y Stream API)	147
5.2.1. Expresiones lambda.....	147
5.2.2. Interfaz funcional.....	149
5.3. Módulos.....	150
5.4. Clases anidadas	152
5.4.1. Static nested class.....	152
5.4.2. Inner class	152
5.5. Anotaciones.....	153
5.5.1. Creación de una anotación personalizada.....	155
5.5.2. Uso de una anotación personalizada.....	156
5.6. Método requireNonNull	156
6. Extras	158
6.1. Javadoc	158
6.1.1. Generar javadoc por línea de comandos	159
6.1.2. Generar javadoc con Eclipse	159
6.2. JavaFX	162
6.2.1. Introducción.....	162
6.2.2. Configuración de JavaFX en Eclipse IDE.....	162

6.2.3.	Modelo en JavaFX	165
6.2.4.	Vistas en JavaFX (parte visual)	165
6.2.5.	Vistas en JavaFX (parte interactiva)	167
6.3.	JUnit 5	170
6.3.1.	Test unitario	170
6.3.2.	Configuración del <i>plugin</i> JUnit	170
6.3.3.	Usando JUnit	172
7.	Bibliografía	179

BORRADOR © D.G.S. - UOC

1. Introducción

1.1. Breve historia

El lenguaje de programación Java apareció en 1995 de la mano de James Gosling de la empresa Sun Microsystems, que fue adquirida por Oracle en 2010. Dicen que originalmente iba a llamarse Oak, debido a un árbol de roble (en inglés, *oak*) que había en el jardín al cual daba el despacho de Gosling. Más tarde fue bautizado como Green, para finalmente llamarse Java. Dicen que el nombre de Java fue inspirado por el café de tipo Java que servían en la cafetería a la que solía ir Gosling con sus compañeros. De ahí que el logo de Java sea una taza humeante de café.

1.2. Ediciones de Java

Java es tanto un lenguaje de programación como una plataforma. Por plataforma entendemos:

Definición de plataforma Java

Un entorno concreto en el que se ejecutan programas desarrollados en lenguaje Java.

En la actualidad existen cuatro plataformas o ediciones de Java que están destinadas a construir diferentes tipos de aplicaciones. Cada plataforma Java está compuesta por una máquina virtual, llamada *Java Virtual Machine* (JVM), y un conjunto de librerías o API (*Application Programming Interface*).

- Java Virtual Machine (JVM): es un programa, para un sistema operativo concreto, que ejecuta programas desarrollados en Java.
- API: es una colección de librerías que un programador puede usar para desarrollar un nuevo software.

Así pues, un programa es desarrollado para una plataforma concreta de Java. Las cuatro plataformas o ediciones de Java que existen en la actualidad son:

- Standard Edition (SE): esta edición es el núcleo del lenguaje y la parte que veremos en esta guía. Su API incluye todo lo relacionado con los tipos básicos, todo lo relacionado con orientación a objetos, clases para redes, seguridad, acceso a bases de datos, parseo de XML, etc.
- Enterprise Edition (EE): esta plataforma es un superconjunto de la Standard Edition (SE), es decir, incluye la SE proporcionando librerías adicionales. Es usada para desarrollar aplicaciones de gran tamaño, distribuidos, escalables, fiables y seguras.
- Micro Edition (ME): a diferencia de la Enterprise Edition, esta edición es un subconjunto de la Standard Edition y está diseñada para desarrollar aplicaciones en dispositivos móviles. Por este motivo, incluye librerías especiales útiles para el desarrollo de aplicaciones en dispositivos móviles y su máquina virtual está adaptada a las capacidades de *smartphones*, *smartwatches*, etc.

- Java Card: esta edición es un subconjunto de la Standard Edition y es usada para ejecutar aplicaciones de manera segura en tarjetas inteligentes (en inglés, *smart cards*) o dispositivos con pequeña capacidad de almacenamiento. Esta tecnología se utiliza por ejemplo en tarjetas monedero.

1.3. Ciclo de vida de un programa Java

Un programador escribe un programa en Java. Para simplificar, imaginemos que su programa sólo utiliza un fichero llamado `Program`. Éste fichero tendrá código escrito en Java y su extensión será `.java`, así pues el nombre completo del fichero será `Program.java`. Una vez escrito el fichero `Program.java`, el programador lo debe compilar. Para ello llama al **Java Compiler (javac)** de la siguiente manera mediante la línea de comandos:

```
javac Program.java
```

Si el programa estuviera formado por más de un fichero `.java`, entonces podría haber compilado todos los archivos a la vez con el siguiente comando:

```
javac *.java
```

Con este paso se consigue convertir el código escrito en Java a código escrito en un lenguaje intermedio llamado **bytecode**. De hecho, Java Compiler crea un fichero nuevo llamado igual que el fichero `.java` original, pero con extensión `.class`. Gráficamente sería:



Se podría decir que el código **bytecode** es similar al **código máquina** que los compiladores de otros lenguajes de programación, como C/C++, generan. La diferencia radica en que el código **bytecode** no está destinado para ejecutarse en un arquitectura de hardware concreta (p.ej. Intel es diferente a Macintosh), sino que está pensado para ejecutarse en una máquina virtual Java concreta (Java Virtual Machine, JVM). **Java Virtual Machine (JVM)** no es más que un intérprete de **bytecode**. Esto conlleva a que durante la fase de interpretación exista una ralentización en el rendimiento del programa respecto al mismo programa escrito en lenguaje máquina que generan los compiladores de lenguajes como C/C++.

<<Nota al margen>>

Bytecode

El bytecode no es entendido por ninguna arquitectura hardware, sino por la JVM. Es menos complejo (y más rápido) para la JVM interpretar bytecode que directamente código Java.

La JVM sólo es dependiente del sistema operativo, pero no de la máquina/ordenador en que se esté ejecutando. Así pues, lo único que se debe hacer es tener instalada la JVM del sistema operativo correspondiente. Gracias a esto, se desarrolla una vez en cualquier dispositivo (ficheros `.java`), en dicho dispositivo se compila/transforma el código Java en un código

estándar e intermedio llamado *bytecode* (ficheros .class) y, finalmente, este código se ejecuta en cualquier dispositivo equipado con la máquina virtual (JVM). De ahí que el eslogan de Java sea: <<Write once, run anywhere>> (en español, <<Escribe una vez, ejecuta en cualquier parte>>). Gráficamente este proceso sería:



Así pues, si se está ejecutando el archivo .class en Windows, el JVM interpretará el *bytecode* del fichero .class y utilizará código nativo que Windows entiende para finalmente ejecutar el programa correctamente.

<<Nota al margen>>

¿Java es compilado o interpretado?

Siempre hay la discusión de si Java es compilado o interpretado, ya que existen las dos fases. En términos generales se dice que es interpretado por la manera en que la JVM ejecuta el código bytecode.

Para que la JVM ejecute el *bytecode* de un programa, hay que llamar al fichero .class cuyo homónimo .java tuviera el método main. Esta llamada se hace de la siguiente manera:

`java Program`

<<Nota al margen>>

Método main

Más adelante veremos qué es el método especial main de un programa.

1.4. JRE vs JDK

Independientemente de la plataforma de Java, existen dos paquetes diferentes: JRE y JDK. ¿Cuál es la diferencia?

El JRE (*Java Runtime Environment*) es un software que proporciona lo mínimo indispensable para poder ejecutar una aplicación Java en el dispositivo en el que se instala. Está formado por la JVM y un conjunto de componentes (p.ej. librerías de integración, toolkits de interfaz de usuario, etc.). El JRE debe ser instalado por todo aquél que quiere ejecutar programas en Java, p.ej. un usuario final.

Por su parte, el JDK (*Java Development Kit*) es un software de desarrollo que permite crear y ejecutar programas en Java. Éste incluye dos elementos:

- Un conjunto de herramientas necesarias para el desarrollo
- El JRE, el cual permite ejecutar un programa Java.

El JDK debe ser instalado por todo aquél que quiera desarrollar programas en Java.

2. Configuración del entorno

2.1. Instalando JDK

Como hemos visto, cuando queremos programar en Java debemos instalar el JDK en nuestro ordenador. En el momento de escribir esta guía, la versión actual era la 13. Que la versión que exista cuando estés leyendo estas líneas sea una más avanzada, no debería suponer un problema, ya que desde hace tiempo la manera de instalar el JDK sigue siendo la misma.

Antes de ver los pasos generales necesarios para instalar el JDK para Windows, Linux y macOS, cabe enfatizar que desde la versión 11 del JDK, la licencia de uso cambió. Si antes era completamente gratuito, desde JDK 11 su utilización sólo es gratuita para uso personal, pero no para desarrollar aplicaciones para terceros. Asimismo, el soporte por parte de Oracle – como pueden ser actualizaciones –, tampoco es gratuita. Es por esto que desde la versión 11, muchas personas prefieren instalarse la versión libre llamada OpenJDK.

2.1.1. OpenJDK vs Oracle JDK

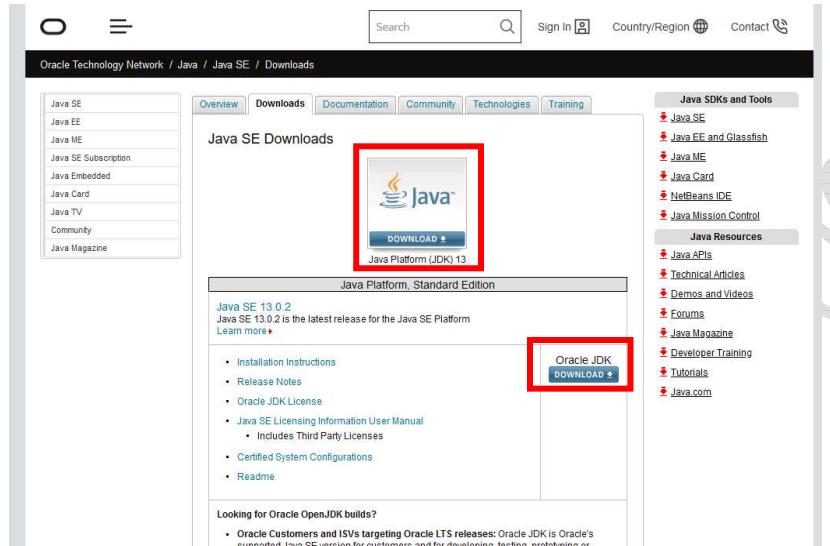
A continuación destacaremos las similitudes y diferencias entre OpenJDK y Oracle JDK:

	OpenJDK	Oracle JDK
Oficialidad	Es una implementación de la especificación oficial de Java. Desde Java SE 7 es la implementación de referencia usada como base para desarrollar Oracle JDK.	Es la versión oficial <i>de facto</i> .
Lanzamiento de nueva release (versión)	Cada 6 meses	Cada 3 años se lanza una versión LTS (<i>Long Term Support</i>), p.ej. JDK 8 y JDK 11, que añade grandes cambios y recibirá soporte (i.e. actualizaciones, corrección de bugs, etc.) durante 8 años. Cada 6 meses aparece una versión no-LTS que añade mejoras sobre la última versión LTS, pero estas mejoras no impactan en el core. Cada versión no-LTS deja de tener soporte a los 6 meses, es decir, cuando aparece una nueva versión.
Licencia	Libre / gratuita	Gratuita para uso personal (i.e. no comercial). De pago para uso comercial (para terceros).
Actualizaciones (p.ej. corrección de bugs)	Cada 6 meses	Inmediato si se paga una licencia. En caso contrario, cada 6 meses.
Encargado del mantenimiento	Comunidad Java, Oracle, Red Hat, IBM, etc.	Oracle Corporation
Instalación	Basado en un .tar.gz o .zip.	Tiene instaladores para diferentes sistemas operativos (msi, rpm, deb, ...)
Rendimiento	No hay diferencias significativas.	

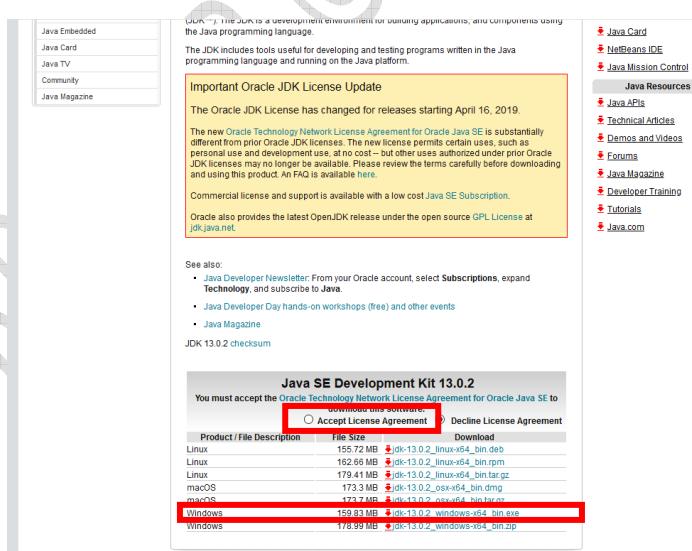
2.1.2. Oracle JDK para Windows

A continuación vamos a explicar cómo instalar JDK 13 en Windows 10:

1. Descárgate la versión de JDK 13.0.2 o superior. Sobre todo, fíjate que se trata del JDK y no del JRE. Accede a través del navegador a la siguiente dirección: <http://java.sun.com/javase/downloads/index.jsp>.



Haz click en uno de los dos botones de *download* (remarcados en rojo en la imagen). A continuación acepta la licencia y escoge la opción de Windows que deseas. Te recomendamos que escojas la opción ejecutable (*.exe).

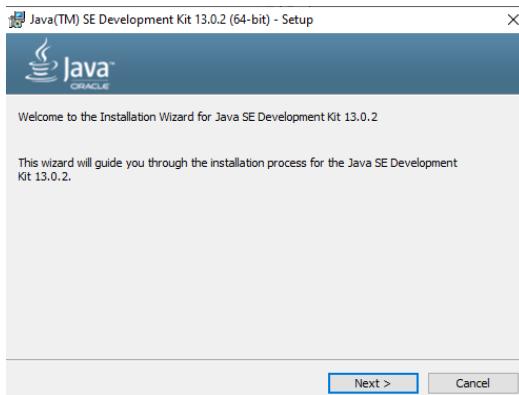


Nota: Si tienes Windows XP o un ordenador con CPU de 32 bits, debes descargarte la versión 8 (8u221), que tiene opción de 32 bits (x86): <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

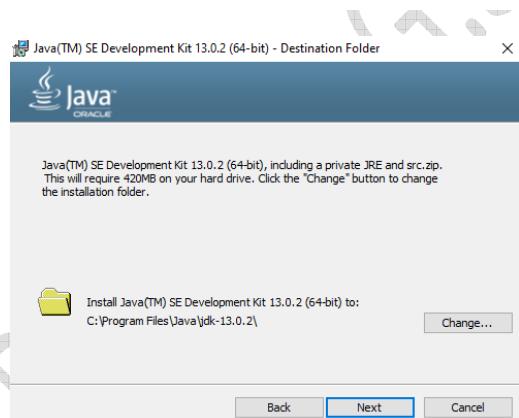
En principio podrás realizar todas las actividades iniciales, pero es importante que consigas un ordenador con CPU de 64 bits y con un sistema operativo Windows 7 o superior.

2. Una vez descargado el programa de instalación, **ejecútalo en modo administrador (i.e. botón derecho del ratón → Ejecutar como administrador)**. El proceso de instalación acaba de comenzar. Una pantalla de seguridad de control aparece para los usuarios de Windows 7 o superior. Si aparece, acepta.

3. A continuación te debe aparecer la pantalla de bienvenida a la instalación, haz *Next*.



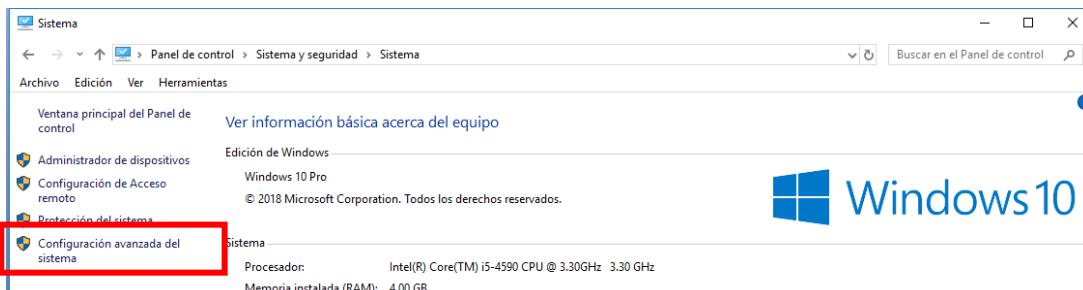
4. Aparecerá una pantalla donde se pide la carpeta de instalación. Deja la que te recomienda el propio instalador. Haz *Next*.



5. Una vez instalado el JDK te aparecerá la siguiente pantalla, donde hay dos botones. Si pulsas en *Next Steps*, entonces se abrirá la página web <https://docs.oracle.com/en/java/javase/13/> donde encontrarás una gran cantidad de documentación. Si pulsas en *Close*, se cerrará el Wizard de instalación.

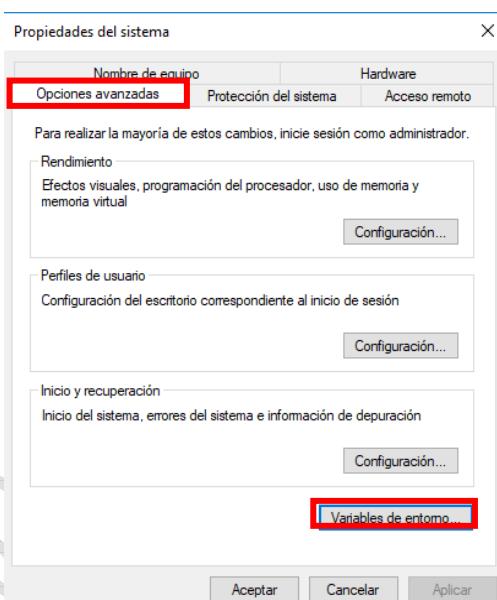


6. Tras haber finalizado la instalación, debes configurar el PATH. Para ello, en Windows (cualquier versión), es necesario que vayas al Panel de control y escojas la opción *Sistema y seguridad*. A continuación escoge *Sistema* y dentro de esta pantalla elige *Configuración avanzada del sistema* (está en el lado izquierdo; ver recuadro rojo en la imagen).

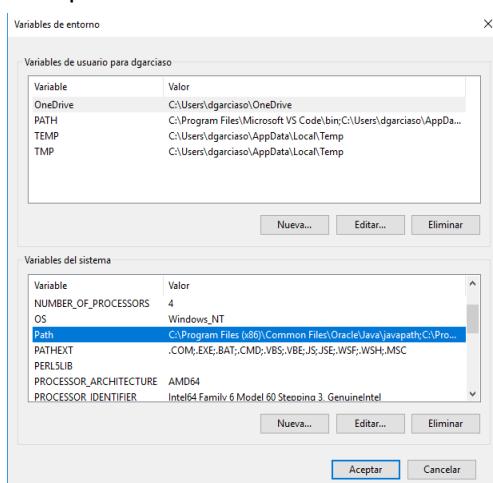


Windows 10

7. Dentro de esta ventana de configuración escoge la pestaña *Opciones avanzadas*, donde encontrarás un botón que dice *Variables de entorno*.



8. Si haces click en *Variables de entorno*, te aparecerá una ventana para modificar las diferentes variables definidas en el sistema. En la lista de abajo busca y selecciona la variable *Path* (o PATH), y a continuación pulsa el botón *Editar...*

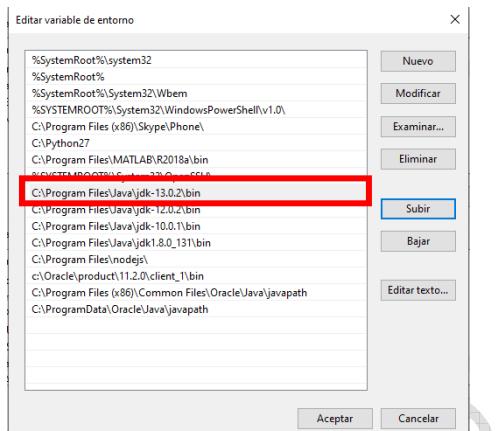


9. Se te abrirá una ventana para modificar el valor de la variable *Path*. Selecciona la primera fila vacía que haya y escribe:

JAVADIR\bin

donde JAVADIR es el directorio donde has instalado el JDK, por ejemplo, si lo has instalado en 'C:\Program Files\Java\jdk-13.0.2', entonces tienes que añadir:

C:\Program Files\Java\jdk-13.0.2\bin



Nota: Recuerda que el nombre de la carpeta dependerá de la versión instalada y la ruta de instalación. Si tienes varias versiones de JDK instaladas, es importante que la versión que acabas de instalar sea la primera del listado. Para conseguir que aparezca antes que cualquier otra versión, utiliza los botones *Subir* y *Bajar* del lado derecho.

10. Finalmente, para comprobar que JDK está correctamente instalado, puedes consultar (por línea de comandos, i.e. cmd) las versiones de java y javac mediante las instrucciones:

```
$ java -version
$ javac -version
```

Estas instrucciones te deberían devolver respectivamente la versión del java (JRE) y javac (compilador) que se están usando en el ordenador. Debe coincidir con la que has instalado. Si todo es correcto, debes ver una pantalla similar a la que se muestra en la siguiente imagen.

```
Microsoft Windows [Versión 10.0.18362.592]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\ldgarcia>java -version
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)

C:\Users\ldgarcia>javac -version
javac 13.0.2

C:\Users\ldgarcia>
```

Si aun así no logras tener instalada la versión 12 de JDK, sigue los pasos que aparecen en el siguiente manual: https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_TOPIC_TOC_PROCESSING_d127e1987

2.1.3. Oracle JDK para Linux

Tenemos dos maneras de instalar JDK. La primera, a través de los repositorios de paquetes de Linux y, la segunda, desde la web de Oracle. Sigue las instrucciones que encontrarás en la siguiente guía: https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_TOPIC_TOC_PROCESSING_d127e1148

2.1.4. Oracle JDK para macOS

El sistema operativo macOS ya trae por defecto la versión completa del JDK instalada y preparada para usarla. En el caso de que se tenga una versión vieja del JDK y se quiera instalar una más nueva, se puede conseguir a través de las actualizaciones de software y el website de soporte de Apple. Si aun así no logras tener instalada la última versión de JDK, sigue los pasos de la siguiente guía: https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_TOPIC_TOC_PROCESSING_d127e1628

2.2. Compilando y ejecutando un programa por línea de comandos

En primer lugar vamos a ver cómo se compila un programa Java por línea de comandos. Es decir, vamos a ejecutar el compilador del JDK para que traduzca de lenguaje Java a *bytecode*. O, lo que es lo mismo, pasaremos de un fichero .java a uno .class. Para ello ves al Ejercicio 1 de la PAC/PEC 1 que te proporcionamos con esta asignatura.

En el directorio PAC_ex1 encontrarás un fichero llamado Ex1.java. Abre una consola de comandos y ves hasta el directorio PAC_ex1 y escribe:

```
javac Ex1.java
```

Entonces verás que no aparece nada. Esto quiere decir que no ha habido ningún error. Sin embargo, si abres el directorio PAC_ex1 verás que ha aparecido un nuevo fichero .class. De hecho es la versión *bytecode* del fichero Ex1.java.

Ábrelo con un editor de texto como puede ser Notepad, Notepad++, Vim, gedit, Kate, Leafpad, etc. Ahora no nos detendremos a ver qué hace este código, sólo queremos que veas que con un editor sencillo podrías escribir código Java. A continuación, iremos de nuevo a la consola de comandos y escribiremos la siguiente instrucción:

```
javac Ex1
```

Esto lo que hará será ejecutar el programa Ex1.class. La salida debería ser: "Number of arguments is not correct! Write: java Ex1 <double>+". Este programa solicita una serie de argumentos para poderse ejecutar correctamente. Podemos escribir lo siguiente:

```
java Ex1 4.25 6 5
```

La anterior ejecución imprimirá por pantalla el texto un número. No obstante, si se escribe una entrada diferente al patrón java Ex1 <double>+, es decir, que los argumentos no son un conjunto de double, entonces el programa imprime un mensaje de error.

2.3. Eclipse IDE

Como hemos visto podemos escribir código con un editor de texto sencillo –p.ej. Notepad– así como compilar y ejecutar programas Java por línea de comandos (i.e. javac y java). Sin embargo, cuando los programas se hacen más complejos, este entorno de desarrollo no será práctico. Es por eso que se crearon los IDE (*Integrated Development Environment*, i.e. entorno integrado de desarrollo), que facilitan el desarrollo de software. IDE hay muchos. Los más utilizados en el ámbito del lenguaje Java son Eclipse, NetBeans e IntelliJ. Son todos muy similares y usar uno u otro dependerá de las preferencias de cada desarrollador. En esta guía vamos a optar por Eclipse.

En primer lugar debemos instalar Eclipse IDE. Para ello, debemos seguir los pasos detallados en el siguiente apartado.

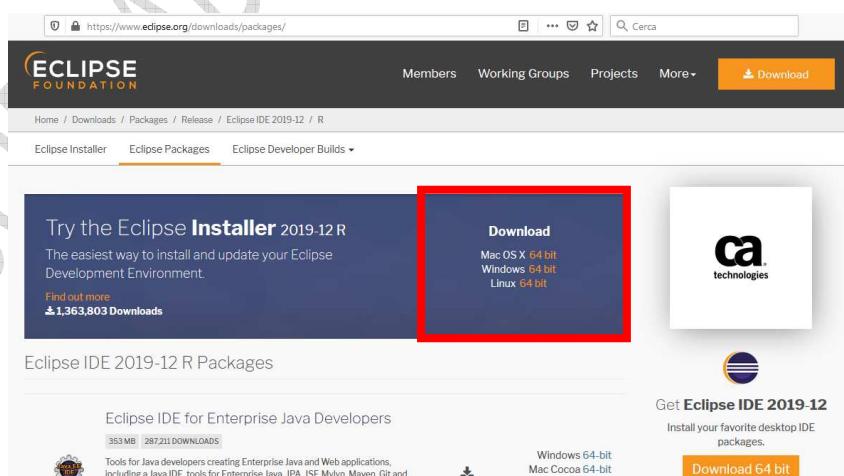
2.3.1. Instalación de Eclipse IDE

La instalación de Eclipse es bastante sencilla, sólo debes seguir los pasos que se detallan a continuación (para sistemas operativos diferentes a Windows, quizás es más sencillo bajarse directamente el fichero comprimido de la modalidad de *Eclipse IDE for Java Developers* y descomprimirlo en el ordenador). Algunos aspectos o nombres pueden cambiar entre lo que se explica en este apartado y lo que te puedes encontrar en tu entorno de trabajo. En tal caso, debes ser capaz de hacer las analogías correspondientes.

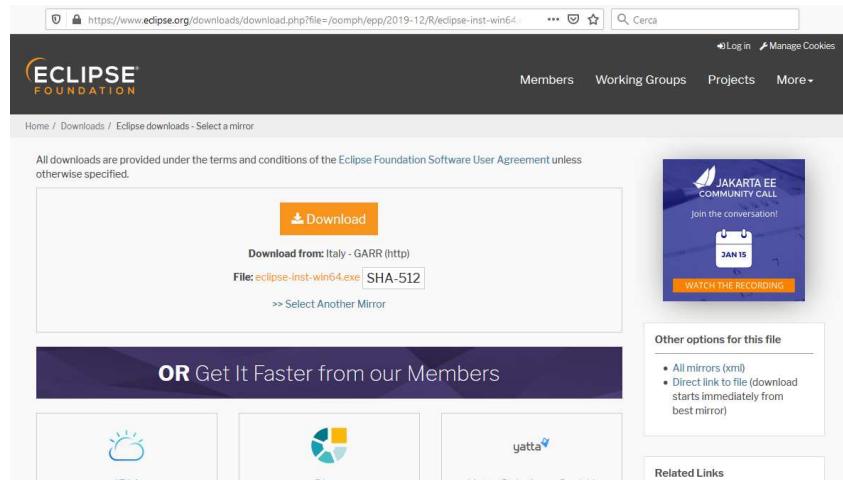
1. En primer lugar, debes tener instalada la versión JDK más actual posible.
2. Ir a la web de descargas de Eclipse:

<https://www.eclipse.org/downloads/eclipse-packages/>

3. Una vez en ella, usaremos la manera más sencilla de instalar Eclipse. Es decir, mediante el instalador Eclipse, en inglés, *Eclipse Installer* (ver recuadro rojo de la Figura 1). Hacemos click en el enlace de nuestro sistema operativo.



4. La web nos llevará a otra página con un botón en grande que dice *Download*. Hacemos clic en él.



The screenshot shows the Eclipse Foundation's download page. At the top, there's a navigation bar with links for Members, Working Groups, Projects, and More. Below the navigation, there's a banner for the 'JAKARTA EE COMMUNITY CALL' with a link to 'WATCH THE RECORDING'. The main content area displays a download button for 'eclipse-inst-win64.exe' from 'Italy - GARR (http)'. It also shows other download options from members like IQMAX, Nhan, and Yatta Solutions GmbH. A sidebar on the right provides 'Other options for this file' including 'All mirrors (xml)' and 'Direct link to file (download starts immediately from best mirror)'. There's also a section for 'Related Links'.

5. Guardamos el ejecutable que nos aparecerá.

6. Una vez tenemos el ejecutable en nuestro ordenador, hacemos doble clic para ejecutarlo. Acuérdate de ejecutarlo como administrador (es decir, en Windows, el usuario con el que estás autenticado en el sistema operativo debe tener permisos de administrador, si no, debes hacer botón derecho sobre el ejecutable y escoger la opción “*Ejecutar como administrador*”).

7. Nos aparecerá una ventana con diferentes modalidades de Eclipse. Antes de nada es interesante (e importante) actualizar el instalador, si es que hay que hacerlo. Para saber si necesitamos actualizarlo, miramos el ícono con dibujo de menú (tres rayas) que hay en la parte superior derecha. Si tiene una exclamación, lo más seguro es que al clicar, nos aparezca la opción *Update* con exclamación. Si es así, hacemos clic en *Update* y esperamos a que se actualice el instalador (seguramente se reiniciará).



8. Cuando tengamos el instalador actualizado y estemos en la pantalla de selección de la modalidad de Eclipse a instalar, escogemos *Eclipse IDE for Java Developers*. También nos serviría *Eclipse IDE for Java EE Developers*, la cual es una versión más completa destinada para el desarrollo de aplicaciones basados en Java EE. Sin embargo, la versión Java EE ocupa más en el disco duro.

<<Comentario al margen>>

Añadir funcionalidades a Eclipse

Si necesitas añadir alguna funcionalidad de Java EE a tu *Eclipse IDE for Java Developers*, puedes hacerlo instalando el *plugin* correspondiente.

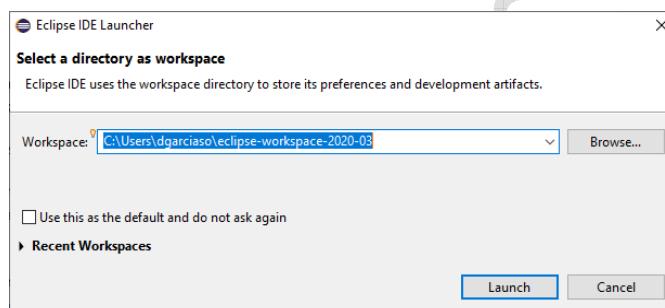
9. Una vez escogida la modalidad, nos aparecerá una ventana que nos pedirá dónde queremos guardar Eclipse, si queremos una entrada en el menú de inicio y un acceso directo en el escritorio. Además no detecta la versión de Java que tenemos por defecto en el sistema

operativo. En el campo *Product Version* escoge la versión *Lastest*. Cuando tengamos nuestras preferencias indicadas, decimos *Install*.

10. Durante la instalación nos aparecerá una ventana para aceptar la licencia de uso. Hacemos clic en *Accept*. La instalación continuará. También es posible que nos pida aceptar unos certificados. Los aceptamos y la instalación seguirá su curso.

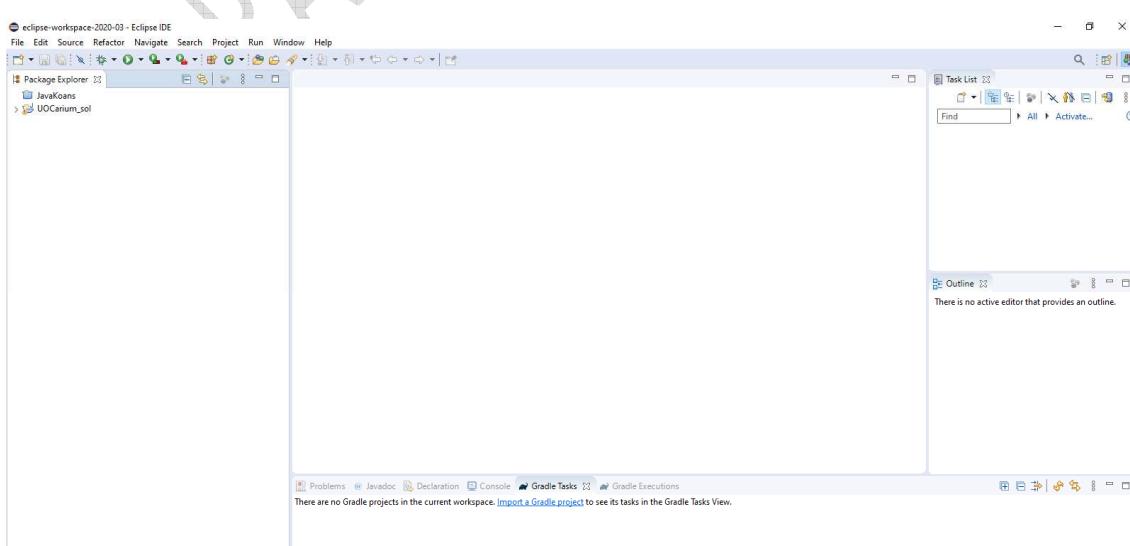
11. Si todo ha ido bien, nos aparecerá en la pantalla el botón *Launch* en verde. Hacemos clic en el botón *Launch* para ejecutar Eclipse.

12. Tras cargar librerías, nos aparecerá una ventana en la que nos pide la ubicación (i.e. carpeta) que Eclipse debe usar como directorio de trabajo o *workspace*. Es decir, allí donde guardará y buscará los proyectos (i.e. los programas que creamos), además de guardar nuestras preferencias. Podemos dejar la ubicación que nos sugiere o escoger otra. También podemos decirle que la recuerde y no nos pregunte más (recomendamos no marcar esta opción). Una vez tengamos claro dónde ubicar nuestro directorio de trabajo, hacemos clic en el botón *OK*.



13. Eclipse seguirá cargándose hasta que, por fin, se abra.

14. La primera vez nos aparecerá una ventana de bienvenida. A continuación cerramos esta ventana mediante la "X" que aparece arriba a la izquierda junto al nombre *Welcome* y nos aparecerá la perspectiva Java.

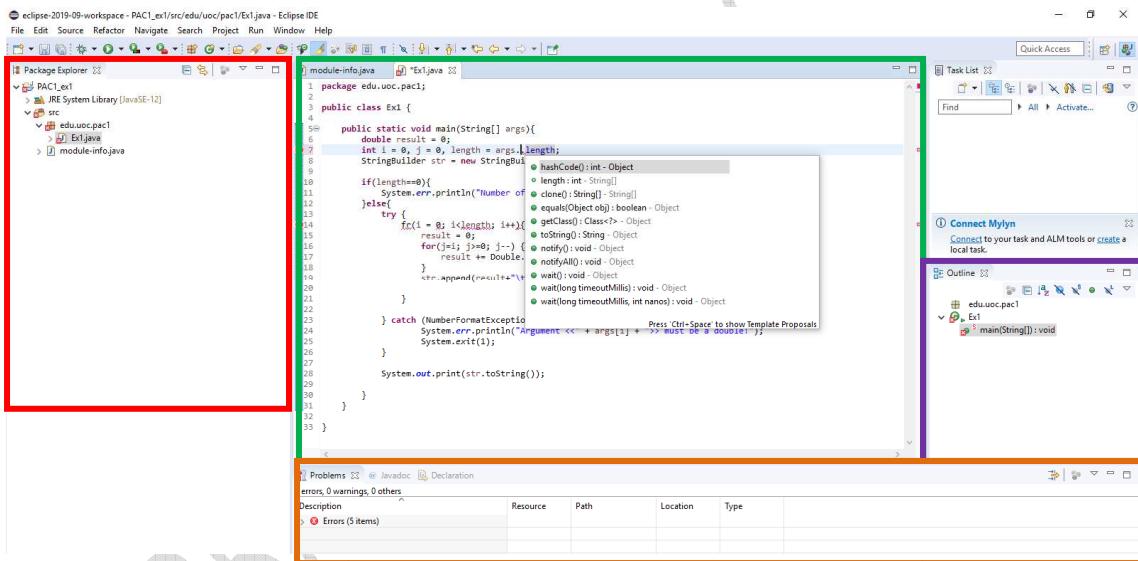


15. Si cerramos Eclipse, éste guardará nuestra configuración. De este modo, cuando lo volvamos a abrir, Eclipse se abrirá con las mismas ventanas y perspectivas que había justo antes de cerrarlo. Esto será especialmente útil cuando estemos desarrollando programas de un cierto tamaño y tengamos abiertos diferentes ficheros y perspectivas.

A continuación vamos a describir muy brevemente los elementos más importantes de Eclipse.

2.3.2. Vistas

Cada una de las ventanas/trozos que se ven en la pantalla de Eclipse es una vista. Cada vista está asociada a una perspectiva, por lo tanto, según en la perspectiva en que nos encontramos, habrá vistas que no estarán disponibles. Todas las vistas tienen unos botones en la parte derecha para maximizar o minimizar la ventana. Haciendo doble clic sobre la ventana también se puede maximizar. Cuando maximizamos una vista, todas las otras vistas se minimizan. Para volver a la configuración normal podemos volver a hacer doble clic en la barra superior de la vista o bien clicar el botón situado a la parte más derecha de la barra superior de la vista, que tiene la forma de dos ventanas sobreuestas. En la perspectiva de Java, cuando abrimos por primera vez un nuevo proyecto, nos encontramos con las siguientes vistas.



Explorador de paquetes (package explorer)

Es la ventana que se encuentra en la parte más izquierda (ver recuadro rojo de la imagen anterior). Se ven todos los proyectos que tenemos en nuestro directorio de trabajo. Para cada proyecto se ve en forma de árbol todos los ficheros que contiene. Los pequeños triángulos que aparecen junto a algunos de los ítems, como por ejemplo directorios, paquetes, ficheros, indican que este elemento se puede expandir. Clicando sobre el triángulo, el ítem se expande revelando los elementos que contiene. En la perspectiva de Java se remarcian los subdirectorios que forman un paquete y los ficheros .java que contiene cada uno de los paquetes. Los ficheros Java se pueden expandir mostrándonos las clases, métodos y atributos que contienen. Si hacemos doble clic sobre algún fichero, éste se nos abrirá en el editor, es decir, en la parte central del aplicativo.

Editor

El editor es la ventana/vista que se encuentra a la parte central de Eclipse (ver recuadro verde de la imagen anterior). Todos los ficheros que se abren, ya sea desde el Explorador de Paquetes o desde el menú Fichero, se muestran en el editor. La parte superior de la vista es una barra de pestañas donde se nos muestran los nombres de los ficheros abiertos recientemente. La pestaña activada corresponde al fichero que se está visualizando en este momento en el editor. En caso de que abriéramos muchos ficheros, en la parte derecha de la barra de pestañas aparecería un símbolo >> con el número de ficheros abiertos debajo. Si clicamos, se nos desplegarán los otros ficheros que están abiertos.

La vista del editor es la que más usaremos durante la implementación de código, por lo tanto, suele ser una buena idea maximizar la ventana del editor mientras trabajamos.

Cuando nos encontramos en la perspectiva de Java y estamos editando un fichero .java, en el código se nos resaltan las palabras claves en color lila. Otros elementos como los nombres de los atributos o cadenas de texto se nos muestran en color azul. Los métodos se pueden ocultar clicando en el símbolo menos (-) que encontramos en la parte izquierda de la línea de declaración del método. Si queremos volver a ver su contenido, sólo hay que clicar en el símbolo más (+) que aparecerá junto a la definición del método.

El editor también tiene la función de ir marcando los errores de compilación que encuentre mientras vamos escribiendo. Los errores salen marcados en la parte derecha como cuadrados rojos, en la parte izquierda como cruces y en la parte de código incorrecta subrayada con rojo. Si nos situamos encima de alguna marca de error nos saldrá un mensaje explicativo del tipo de error.

Otra funcionalidad del editor es el autocompletado de código. El autocompletado de código nos permite inspeccionar el código que se puede incluir allá donde estemos escribiendo. Por ejemplo, si escribimos el nombre de una instancia de un objeto, podemos inspeccionar los métodos y atributos de este objeto tan sólo con escribir el punto para invocar las llamadas (mensajes). Para ello escribimos el punto y esperamos a que nos salga la lista de posibles métodos y atributos. Se puede llamar al autocompletado de código en cualquier momento pulsando la combinación de teclas *Ctrl* y *espacio*. Estemos donde estemos, se nos mostrará una lista de sugerencias con los posibles objetos o variables que tengamos definidas.

Outline

La vista de *outline* (ver recuadro lila de la imagen anterior) nos permite visualizar los componentes del fichero Java que tengamos abierto en el editor, de la misma forma en que se visualizan en el explorador de paquetes. Es decir, se nos muestran las clases, atributos y métodos que haya en el fichero .java actual. Si se clica en cualquiera de estos elementos en la vista de *outline*, inmediatamente se resalta en la ventana del editor. Los botones que hay encima de la vista nos permiten ocultar o mostrar diferentes elementos, como métodos no públicos, métodos estáticos, atributos, etc. También nos permite ordenar los elementos en la vista por orden alfabético.

Problems

Esta vista (ver recuadro naranja de la imagen anterior) se suele encontrar en la parte inferior del editor y nos muestra los errores y avisos que tengamos en cualquier fichero de los proyectos que tengamos abiertos. Para cada problema nos muestra una descripción y el lugar donde se encuentra (fichero y línea).

Console

Esta vista se suele encontrar también en la parte inferior del editor, en otra pestaña junto a la de problemas (ver recuadro naranja de la imagen anterior). Esta vista se activa automáticamente cuando se ejecuta una aplicación Java y sirve para mostrar la salida del programa, así como para pedir la entrada de datos a la aplicación si hace falta.

2.3.3. Menús

Las funcionalidades de Eclipse se encuentran disponibles a través de la barra de menú general de la aplicación situada en la parte superior. El menú se encuentra dividido en diferentes submenús: *File*, *Edit*, *Source*, *Refactor*, *Navigate*, *Search*, *Project*, *Run*, *Window* y *Help*. Según la perspectiva, estos menús, así como su contenido, pueden variar.

También hay disponible un menú secundario accesible pulsando el botón derecho del ratón encima de cualquier selección que hagamos sobre las vistas de explorador de paquetes o editor. Este menú contextual muestra las opciones de menú más comunes y adecuadas a la selección que estamos haciendo.

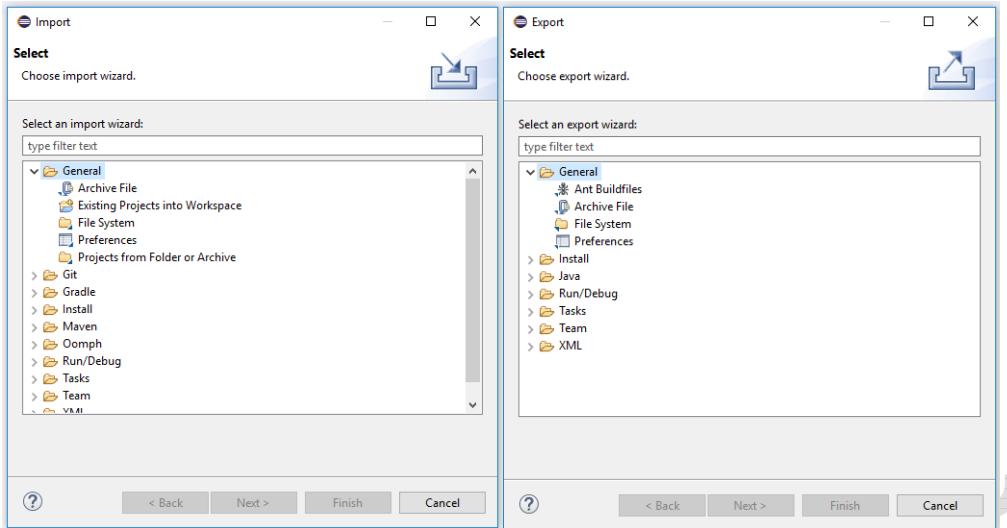
Menú File

Desde este menú podemos acceder a los asistentes de creación de nuevas clases, proyectos, paquetes, interfaces y cualquiera otro elemento relacionado con Java. Para ello sólo hay que abrir el submenú *New* y desde allí seleccionar el asistente que se deseé.

Desde el menú *File* tenemos acceso a todas las operaciones sobre los ficheros de nuestro proyecto. Si estamos situados sobre algún fichero en la vista de explorador de paquetes, podremos mover el fichero o renombrarlo. También existe la opción de refrescar (**Refresh**) las carpetas del explorador. Esta funcionalidad se puede realizar también usando el atajo de teclado F5. Tanto si estamos en esta vista o sobre el editor, podremos cerrar el fichero seleccionado o guardarlo con otro nombre.

De este menú hay que destacar también la posibilidad de cambiar de directorio de trabajo, seleccionando *Switch Workspace*, de este modo podemos tener diferentes directorios de trabajo donde guardar nuestros proyectos. El último directorio de trabajo seleccionado será con el que se abra Eclipse la próxima vez que lo pongamos en marcha.

Finalmente, especial atención merecen los apartados de importar y exportar del menú *File*. Estos asistentes son extremadamente importantes para poder llevarnos nuestros proyectos Eclipse a otras máquinas. Cuando abrimos estos asistentes se nos muestra una lista de posibles ubicaciones de dónde importar o adónde exportar.



En el apartado *General* encontramos las herramientas más utilizadas de los dos asistentes. En el apartado del asistente de importación destacamos:

- La opción *Existing Projects into Workspace* nos permite crear un proyecto en nuestro directorio de trabajo a partir de un fichero comprimido que contenga un proyecto Eclipse o desde otro directorio de trabajo. El proyecto se nos abrirá dentro de nuestro directorio de trabajo y, si queremos, podremos marcar la opción *Copy projects into workspace* para guardarlo.
- La opción *Archive File* nos permite guardar ficheros desde un fichero comprimido dentro de un proyecto Eclipse que ya exista en nuestro directorio de trabajo. Estos ficheros pasarán a formar parte del proyecto seleccionado.
- La opción *File System* nos permite guardar ficheros que tengamos en alguna carpeta de nuestro sistema dentro de un proyecto Eclipse que ya exista en nuestro directorio de trabajo. Estos ficheros pasarán a formar parte del proyecto seleccionado.

En el apartado *General* del asistente de exportación destacamos las siguientes opciones:

- La opción *Archive File* exporta un proyecto de nuestro directorio de trabajo a un fichero comprimido.
- La opción *File System* exporta un proyecto desde nuestro directorio de trabajo a otro directorio local de nuestra máquina.

Dentro del asistente de exportación hay que destacar también el apartado *Java*. Tenemos tres opciones:

- La opción *JAR File* permite generar un fichero JAR de cualquiera de nuestros proyectos. Un fichero JAR es un contenedor de clases Java que permite tener clases Java comprimidas y agrupadas en un único fichero. Esto facilita la exportación y transmisión de programas/APIs entre programadores.

- La opción *Javadoc* que permite generar la documentación *Javadoc* de cualquiera de nuestros proyectos, especificando las opciones que deseamos y el directorio donde guarda la documentación generada.
- La opción *Runnable JAR file* crea un fichero JAR autoejecutable, es decir, como si fuera un *.exe*.

Menú Project

El menú *Project* nos muestra opciones que nos permiten especificar cómo queremos que se lleve a cabo la compilación del proyecto. La última opción de todas es la de propiedades del proyecto (*Properties*). Desde *Properties* se puede configurar cualquier aspecto del proyecto. Las opciones más destacadas son:

- Java Build Path: desde este formulario podemos especificar librerías (p.ej. ficheros JAR) y código que queremos añadir a nuestro proyecto. Este formulario es el mismo que se puede acceder desde el asistente de proyecto nuevo. El *Build Path* es la manera que tiene Eclipse de configurar el *classpath* de la máquina virtual de Java, es, por lo tanto, muy importante que el *Build Path* esté muy bien configurado, puesto que si no, tendremos errores de compilación.

En el apartado de librerías siempre sale por defecto como mínimo las librerías de sistema de la máquina virtual que estemos usando. Si se edita esta librería tendremos la opción de cambiarla por cualquier otra de las máquinas virtuales de Java que tengamos en nuestra máquina. Se pueden añadir otras librerías que hagan falta en nuestro proyecto desde ficheros JAR o desde directorios. Además hay las opciones de editar y borrar librerías ya incluidas. En el apartado *Source* sale por defecto el directorio donde se encuentra el código del proyecto.

- Java Compiler: desde este formulario podemos escoger el grado de compatibilidad de nuestro código. El grado de compatibilidad (*JDK Compliance*) significa hasta qué máquina virtual este código tiene que ser válido. Por ejemplo, si ponemos compatibilidad hasta la máquina virtual 1.4, todos los elementos sintácticos propios de la máquina virtual 1.5, como los genéricos o enumeraciones, que haya en nuestro código darán error. Por defecto se cogen las propiedades generales especificadas en nuestro directorio de trabajo. Se puede cambiar la configuración para un proyecto en concreto si activamos la casilla *Enable project specific settings*.

Menú Run

Desde este menú podemos ejecutar nuestro código, gestionar diferentes configuraciones de ejecución del proyecto y llamar al *debugger*.

La opción *Run* ejecutará la clase que tengamos seleccionada en el editor o en el explorador de paquetes. Buscará el método *main* y lo ejecutará. Los posibles resultados de salida por consola se verán en la vista *Console*. En caso de no haber ningún método *main*, saltará un aviso informándonos de que la selección no contiene ningún método que se pueda ejecutar.

Otros menús

En el menú *Window* encontramos opciones para configurarnos el entorno de ventanas a nuestro gusto. Podemos abrir vistas, cambiar de perspectiva y configurarla. También tenemos la opción de restaurar una perspectiva en su estado por defecto, guardarla o cerrarla.

En el menú *Search* se encuentran las opciones para buscar cadenas de texto dentro de nuestro código. Además, se pueden hacer otros tipos de búsquedas, como encontrar las referencias o declaraciones de una variable. Sólo hay que seleccionar en el editor la variable que se desee buscar y elegir la opción adecuada en el menú de búsquedas.

En los menús *Source* y *Refactor* hay toda una serie de opciones con tal de automatizar ciertas tareas repetitivas a la hora de desarrollar una aplicación. En el menú *Source* podemos encontrar utilidades como añadir bloques de comentarios, generar *setters* y *getters* automáticamente, añadir de forma automática los *imports* que hagan falta en el fichero, crear la identación del código de forma automática, etc. En el menú *Refactor* tenemos utilidades para propagar cambios en el código sobre todos los ficheros .java implicados, como por ejemplo cambiar el nombre de una clase o método y cambiar todas las referencias y declaraciones que haya en el código. Desde aquí también podemos mover una clase de un paquete a otro y propagar los cambios que esto produzca en las demás clases del proyecto.

3. Fundamentos de Java (como lenguaje imperativo)

3.1. Comentarios dentro del código

Los comentarios se utilizan para documentar y explicar el código que realizamos. Los comentarios son ignorados por el compilador, sin embargo son importantes para explicar la lógica del programa (y algunas decisiones tomadas) a terceros (y a uno mismo en el futuro). Decirte que en Java hay dos maneras de escribir comentarios en el código:

- a) `//Comentario de una línea`
- b) `/* Comentario
de múltiples
líneas
*/`

Además de para documentar, los comentarios también sirven para `<<capar>>` o `<<inhabilitar>>` parte de tu código. Así pues, en vez de borrar un trozo de código para siempre, puedes comentarlo y, si fuera necesario, recuperarlo más tarde. Una vez sepas que ese código es descartable, puedes borrarlo para siempre. Esto es útil por ejemplo cuando estamos probando cosas. También es útil si estamos encallados escribiendo un algoritmo complejo, pero tenemos un código base creado que sabemos que funciona. Ese código base podría ser comentado para recuperarlo más tarde si al escribir el algoritmo llegamos a un callejón sin salida y debemos empezar de nuevo. De esta manera, el código base sirve como punto de partida desde donde iniciar de nuevo el algoritmo.

3.2. Estructura básica de un programa

Como ya sabemos, Java es un lenguaje pensado alrededor del paradigma de la programación orientada a objetos (POO). Por el momento no entraremos en conceptos de POO, pero irremediablemente, por la naturaleza de Java, aparecerán desde el principio de las explicaciones. Como consecuencia de ello, algunos elementos POO aparecen cuando analizamos el código mínimo para crear un programa en Java.

`ClassName.java`

```
/*
Este código se puede considerar una plantilla para empezar a programar en Java.
Como puedes ver, estamos utilizando un comentario de múltiples líneas para documentar.

*/
public class ClassName{ //Éste es el nombre de la clase, escoge uno representativo.

    //El main es un método/función especial que hace de punto de entrada del programa
    public static void main(String[] args){

        //TODO: Aquí va tu código!!

    }

    //TODO: Aquí podría ir más código en forma de funciones (también llamados métodos)
}
```

Analicemos el ejemplo anterior:

- Debido a que Java está orientado a objetos, los programas en Java están organizados por clases. Cada clase tiene que ir en un fichero cuyo nombre debe ser el mismo que el de la clase. Por una cuestión de convención de nombres, lo más habitual es que la primera letra de cada palabra que compone el nombre de la clase empiece por mayúscula. En el caso del ejemplo, la clase se llama `ClassName` y el fichero `ClassName.java`.
- Un programa de una cierta envergadura estará formado por más de una clase (i.e. más de un fichero `.java`). Esto obliga a que, en como mínimo una de esas clases, haya un método especial llamado `main`. Dicho método sirve de punto de entrada al programa. Es decir, el programa se iniciará llamando a este método. El método `main` tiene una firma concreta:

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

<<comentario al margen>>

Uso de `varargs` en el `main`

Como veremos más adelante, `String[] args` se puede cambiar por `String...args`, gracias a la nueva sintaxis de las `varargs`.

- El método `main` no devuelve nada (por eso su tipo de retorno es `void`) y recibe un `array` de `String` (i.e. cadena de caracteres) por parámetros llamado `args` (el nombre se puede cambiar, pero lo más habitual es denominarlo así, procedente de la abreviatura de `arguments`). Este parámetro `args` almacena los valores pasados por línea de comandos (i.e. argumentos) cuando el programa se ejecuta. Como en el ejemplo el método `main` se encuentra en la clase `ClassName`, haremos:

```
java ClassName David 36
```

- En este caso, `args` tendría dos casillas con los valores "David" y "36", los dos serían texto (i.e. `String`). El primer valor estaría en `args[0]` y el segundo en `args[1]`.
- Por ahora olvidémonos de la palabra reservada `static`. Veremos su significado tanto en esta guía como en los materiales teóricos de diseño y programación orientados a objetos. Eso sí, si creamos más funciones dentro del fichero `ClassName`, **por ahora (y sólo por ahora), al no trabajar orientado a objetos (que es para lo que está concebido Java)**, las firmas de todas las funciones deberán tener la palabra `static`.

Cuando pasemos a programar orientado a objetos, veremos que este requisito ya no será necesario.

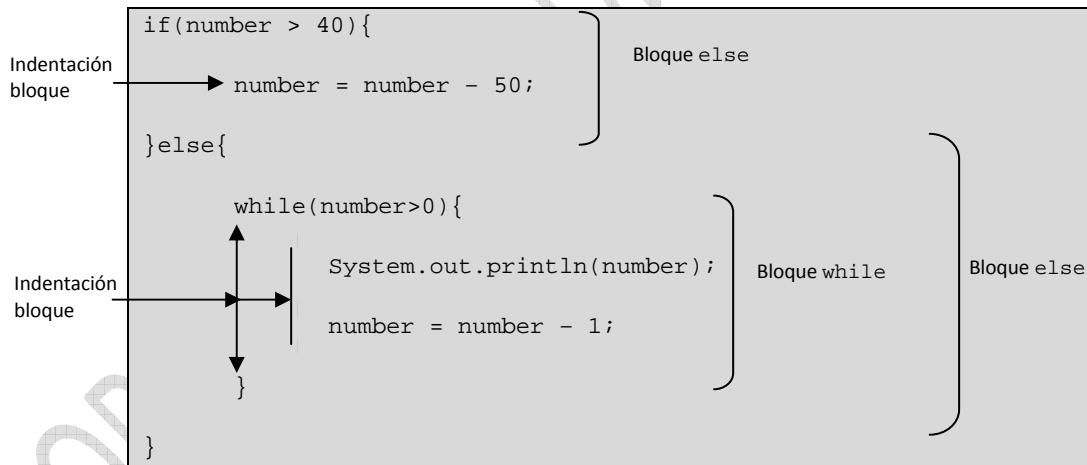
- En los dos comentarios que pone TODO (del inglés, *to do*, i.e. para hacer, pendiente...) podríamos escribir el código de nuestro programa. El código del primer estará formado por instrucciones/sentencias (*statements*) y bloques (*blocks*).
- **Instrucción/sentencia:** es la unidad mínima de programación encargada de realizar una acción. Cada sentencia debe terminar con un punto y coma (:).

```
int number = 50; //declara una variable de tipo int llamada
//number e inicializada con el valor 50

number = number * 3; //multiplica por 3 el valor de number y se
//lo asigna a number

System.out.println(number); //imprime por pantalla el valor de
//number
```

- **Bloque:** es un grupo de sentencias agrupadas entre dos llaves { }. Todas las sentencias ubicadas entre dos llaves son tratadas como una única unidad. Dentro de dos llaves puede no haber ninguna instrucción.



- A la hora de escribir nuestro código debemos tener presente algunas convenciones de estilo de formato:
- No importa añadir muchos espacios, tabuladores, *enters* (i.e. nuevas líneas en blanco) entre dos palabras, puesto que Java los trata como un único espacio/tabulador/nueva línea.
- Es importante indentar (i.e. sangrar) el código con tabuladores y espacios en blanco así como usar apropiadamente los espacios en blanco entre líneas con el fin de facilitar la lectura del código.
- En el caso de las llaves, lo más habitual en Java es poner la primera llave ({}) al final de la primera línea. En el caso de la última llave (}) ésta se pone en una

línea después de la última sentencia y alineada con la primera sentencia del bloque (mira el código anterior).

3.3. Entrada por teclado y salida por pantalla (I/O)

Java tiene tres *streams* (flujos) de entrada y salida:

- Entrada estándar (*standard input stream*): por defecto se refiere al teclado, aunque se puede redirigir a otra fuente de entrada, p.ej. un fichero de texto. Para acceder a este flujo usaremos `System.in`.
- Salida estándar (*standard output stream*): por defecto Java utiliza la pantalla de la consola como salida, pero se puede cambiar, p.ej. un fichero de texto. Para acceder a este flujo usaremos `System.out`.
- Salida de error (*error output stream*): este *stream* se utiliza para mostrar mensajes de error o con información importante. Por defecto utiliza la pantalla de la consola, aunque es muy frecuente redireccionarlo hacia un fichero de *log*. Para acceder a este flujo usaremos `System.err`.

La clase `System` de la API Java nos proporciona acceso a tres atributos públicos y estáticos llamados `in`, `out` y `err`. Tanto `System.in`, como `System.out` y `System.err` son instanciados e inicializados automáticamente por la JVM cuando ésta arranca, así que no hay que hacer nada para tenerlos disponibles.

En este apartado sólo veremos los métodos básicos de la entrada y la salida estándares, obviando la salida de error.

3.3.1. Métodos de salida

La clase `System` de la API Java nos proporciona acceso a un atributo público y estático llamado `out` de tipo `PrintStream` (otra clase de la API de Java). Dicho atributo, que es una referencia a un objeto de la clase `PrintStream`, nos permite acceder a los métodos públicos de dicha clase, la cual se encarga de escribir datos en el flujo de salida que se indique. De entre ellos, destaca: `print`, `println` y `printf`.

Métodos `print` y `println`

Estos dos métodos son los más frecuentes. La diferencia entre ellos es que `println` imprime el contenido que se le pase y añade un salto de línea (`\n`). Es decir, avanza el cursor para escribir en la siguiente línea de la pantalla. Sabiendo esto, el siguiente código:

```
System.out.print(5);  
  
System.out.println(8); //Es equivalente a System.out.print(8 + "\n");  
  
System.out.print(5);
```

Imprimiría:

Como vemos, el 8 se imprimiría en la misma línea que el primer 5, porque el 5 ha sido imprimido con un `print`.

Método `printf`

Los dos métodos anteriores no permiten especificar un formato para controlar aspectos como, por ejemplo, el número de decimales a mostrar para un valor de tipo `double`. El método `printf` recibe, como mínimo, una cadena de texto indicando el texto junto con el formato a mostrar. A continuación, si es necesario, recibe tantos valores (en forma de literales o variables) como se necesiten para llenar el texto. Veamos varios ejemplos para entender su funcionamiento:

```
System.out.printf("Hola! %d %.2f", 53, 89.7); //Hola! 53 89.70 → añade un 0
//para que 87.7 tenga dos decimales

System.out.printf("Hola! %03d", 53); //Hola! 053 → añade un 0 para que 53
//tenga 3 dígitos, sino ponemos el 0 en %03d imprimiría un espacio en blanco
```

La manera de indicar en el texto de salida dónde queremos un formato especial es mediante el uso de un comodín que tiene un determinado formato: `%[flags][ancho]códigoConversión`. Tanto los *flags* como el ancho son dos valores opcionales. Los *flags* sirven para indicar aspectos como la alineación del texto y el elemento a añadir como *padding* (relleno), entre otros. Por su parte, el ancho es un valor que indica cuánto espacio debe ocupar el elemento como mínimo. Finalmente, el código de conversión sirve para indicar el tipo del valor: `d` para entero, `f` para `float` o `double`, `c` para `char` y `s` para `String` (cadena de texto). Veamos algunos ejemplos:

```
System.out.printf("%s|%6d|%.2f", "eps", 53, 89.7); //eps|      53|89.70 → el
//número 53 ocupa un ancho de 6 caracteres.

System.out.printf("Hola|%-4d|", 53); //Hola|53 |→ 53 ocupa 5 posiciones y
//está alineado a la izquierda (lo indicamos con un -).
```

Respecto al salto de línea, este método funciona exactamente igual que `print`, es decir, no avanza el cursor a la siguiente línea. Si queremos hacer un salto, debemos añadir `\n` al final del texto.

3.3.2. Métodos de entrada

Como ya hemos dicho, por defecto, el flujo de entrada viene del teclado. En JDK 5 se introdujo una clase llamada `Scanner` que facilita la lectura con formato del teclado. De este modo, con `Scanner` podremos controlar, de manera sencilla, si queremos leer un `int`, un `double`, etc.

```
Scanner in = new Scanner(System.in); //Objeto Scanner que lee de teclado

int number = in.nextInt(); //Espera recibir un int, en caso contrario lanza
una excepción
```

```

double decimal = in.nextDouble(); //Espera recibir un double, en caso
contrario lanza una //excepción

String word = in.next(); //Lee un texto hasta llegar al primer espacio

String text = in.nextLine(); //Lee un texto hasta llegar al primer salto de
//línea (incluye los espacios en blanco)

in.close(); //Cerramos el input

```

3.4. Variables

Como ya sabes, una variable se define como:

Definición de variable

Unidad básica de almacenamiento. Se puede ver como una etiqueta que permite guardar y recuperar un dato de memoria. Este dato será de un tipo. Así pues una variable tendrá un nombre, un tipo y guardará un valor.

Se llama <>variable<> porque el valor que almacena puede variar (i.e. cambiar) durante la ejecución del programa.

3.4.1. Nombre

Los nombres de las variables deben ser representativos de aquello que almacenan. Imagina que tenemos tres variables con los siguientes tres nombres: n, p1 y vC. ¿Sabes qué almacenan? Ni idea, ¿verdad? Así pues, estos nombres tan crípticos no son adecuados, aunque sí correctos sintácticamente (es decir, Java no dará error). Sin embargo, si cambiamos los nombres de las variables anteriores por estos otros: numberPeople, person1 y viewsCount, ahora sí que podrías saber qué guardan. Como puedes ver, escoger un nombre que sea descriptivo es una buena práctica.

A la hora de asignar un nombre a una variable debemos tener presente que:

- Los nombres de las variables son *case-sensitive*. Es decir, Java distingue entre minúsculas y mayúsculas, por lo que no son numPeople y numpeople no son la misma variable.
- El nombre de una variable debe empezar o con una letra, o con el símbolo \$, o con el símbolo *underscore* _. No obstante, se ha generalizado la buena práctica de que el nombre de las variables empiecen por una letra, no por \$ ni _.
- El nombre de una variable puede tener cualquier longitud y a partir de la primera letra, éste puede estar formado por cualquier combinación de letras Unicode, dígitos, y los símbolos \$ y *underscore* _.
- Para escribir el nombre de una variable en Java se sigue la convención denominada *lower camel case* (o *dromedary case*). Esta convención indica que la primera palabra

se escribe en minúscula y, si el nombre de la variable está compuesto por dos o más palabras, entonces a partir de la segunda palabra la primera letra de cada una de ellas se escribe en mayúscula, p.ej. lastName (está compuesta de last + name), height, resourceNumber, xMin, yTopLeft, etc.

- El nombre que se le asigne a una variable no puede coincidir con uno de los siguientes literales: true, false y null.
- Tampoco se pueden usar como nombres de variable algunas de las siguientes palabras reservada de Java (también llamadas *keywords*):

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while
				yield (JDK 13+)

3.4.2. Tipos

Como hemos comentado, las variables serán de un tipo. En Java existen dos tipos:

- Primitivos (o básicos)
- Referencias

Los primeros sirven para almacenar valores sencillos, mientras que las referencias se utilizan para guardar elementos complejos como son los objetos de una clase.

Primitivos

En Java existen 8 tipos primitivos o básicos. Éstos son:

Tipo	Bytes necesarios para guardar el valor	Intervalo/rango (i.e. valores posibles)	Valor por defecto	Utilidad	Ejemplos
boolean	1 bit	true o false	false		true
byte	1 (8 bits)	[-128, 127]	0	Guardar datos pequeños.	56, 83
short	2 (16 bits)	[-32768, 32767]	0	Mismo uso que	32769, 0xffff

				byte, pero su tamaño es el doble.
char	2 (16 bits)	['\u0000' , '\uffff'] (internamente se guarda como un entero [0, 65535])	'\u0000'	Guardar un símbolo de la codificación Unicode.
int	4 (32 bits)	[-2147483648, 2147483647]	0	Guardar valores enteros.
long	8 (64 bits)	[-2 ⁶³ , 2 ⁶³ -1]	0	Guardar valores enteros que no caben dentro del rango del int.
float	4 (32 bits)	Estándar para coma flotante de 32 bits (IEEE 754).	0.0	Guardar valores decimales con 6-7 decimales significativos.
double	8 (64 bits)	Estándar para coma flotante de 64 bits (IEEE 754).	0.0	Guardar valores decimales con 15 decimales significativos.

<<Comentario al margen>>

Unicode

El código Unicode es un estándar de la informática para representar texto. Éste incluye alfabetos, símbolos (p.ej. \$, &, ?, @) y elementos especiales (p.ej. €). Unicode incluye más de 110.000 caracteres.

Para declarar una variable seguiremos uno de los siguientes patrones (y combinaciones de ellos):

```
type nameVariable;
type nameVariable = initialValue;
type nameVariable1, nameVariable2;
```

Por ejemplo:

```
float grade;
int numberPeople = 120360;
```

```
byte age = 36; //se podría declarar int, pero la edad de una persona no será
//mayor a 127. Con un byte ocupamos menos espacio en memoria (1 byte) que con
//un int (4 bytes).
```

```
int viewsCount = 820_656, distance = 134045, numStudents;
```

Como se puede apreciar en el ejemplo anterior, en el momento de la declaración se puede aprovechar para inicializar la variable con un valor. Si no se indica un valor, entonces el compilador le asigna el valor por defecto del tipo con el que se ha declarado. En el caso de la variable grade, ésta será inicializada automáticamente con el valor 0.0, mientras que la variable numStudents será inicializada con el valor 0. Asimismo, se puede ver cómo podemos inicializar más de una variable de un mismo tiempo a la vez, combinando al mismo tiempo la inicialización o no de dichas variables. Hay algunos programadores que desaconsejan declarar las variables de esta manera y prefieren que cada variable sea declarada individualmente. Es decir, cambiaríamos la última línea del ejemplo anterior por:

```
int viewsCount = 820_656;
int distance = 134045;
int numStudents;
```

Por otro lado, en el caso de datos numéricos, Java permite, para facilitar su lectura, separarlos mediante símbolo _ (*underscore*), p.ej. 123_456, 12_345, 1_3 (sería el número 13), etc. Es justo lo que se ha hecho en la variable viewsCount.

Asimismo, los literales de números enteros se pueden expresar en binario (base 2), octal (base 8), decimal (base 10) o hexadecimal (base 16). Por ejemplo:

```
int number = 0b01001011; //número 75 en decimal. Se debe anteponer 0b o 0B
//(cero b) al número binario.

int number = 01144; //número 612 en decimal. Se debe anteponer un 0 (cero) al
//número en octal.

int number = 1983; //número 1983 en decimal. Es la manera habitual de
//representar los números

int number = 0xFFFF; //número 65535 en decimal. Se debe anteponer 0x o 0X
//(cero X) al número en hexadecimal.

int number = 0xFFFF_F; //con los literales numéricos podemos usar siempre el
//underscore _ independientemente de su formato.
```

En este punto hay que tener presente una apreciación con los tipos long y float. Veamos el siguiente ejemplo:

```
long year = 1983;
float price = 9.99;
```

Las dos declaraciones anteriores lanzarán un error indicando que el valor que se desea asignar a `year` y a `price` es incompatible. ¿Cómo es esto posible si están dentro del rango/intervalo

de valores? Sencillamente porque el compilador de Java considera los valores 1983 y 9.99 como valores de tipo `int` y `double`, respectivamente. Así pues, para forzar que los considere `long` y `float`, respectivamente, debemos añadir la letra `L` en el primer caso, y la letra `F` en el segundo. En ambos casos la letra puede ser mayúscula o minúscula:

```
long year = 1983L;
float price = 9.99f;
```

A continuación, veamos un caso de variable de tipo `char` y otro de tipo `boolean`.

```
char initial = 'D';
boolean open = true;
```

Las variables de tipo `char` guardan un único carácter. Estos valores se delimitan con una comilla simple. En el caso de las variables de tipo `boolean`, los valores que puede aceptar son `true` o `false`.

<<comentario al margen>>

JDK 10 y posterior

Desde JDK 10 es posible no indicar el tipo (ya sea primitivo o referencia) de una variable local utilizando la palabra reservada `var`. Una restricción de usar `var` es que la variable debe ser inicializada para que el compilador pueda inferir el tipo de la variable, p.ej. `var v1 = 5;`

Cuando declaramos una variable de tipo primitivo, lo que hace el compilador es asignarle una dirección de memoria a dicha variable. Nosotros como programadores no tenemos control de en qué dirección de memoria se crea esta variable.

Contenido	Dirección de memoria
36	1000
	1004
	1008
	1012

```
int age = 36;
```

En el ejemplo anterior, la variable `age` se encuentra en la dirección de memoria 1000. A nivel de memoria el nombre de la variable desaparece y pasa a ser la dirección 1000. Es decir, a bajo nivel no existen nombres de variables, sino direcciones de memoria. Por lo tanto, gracias a los lenguajes de alto nivel como es el caso de Java, el programador no trabaja directamente con direcciones de memoria, lo cual facilita el desarrollo de software. Así pues, cuando un programador escribe el siguiente código después de la declaración anterior:

```
age = age + 2;
```

El compilador traduce esta instrucción de manera que va a la dirección de memoria 1000, recupera el valor que hay en dicha posición (i.e. 36), le suma 2 y guarda en la dirección de memoria 1000 el valor 38.

Referencias

Todas las variables que no son declaradas como uno de los 8 tipos primitivos anteriores, son referencias. En el siguiente ejemplo vamos a declarar un *array* de int.

```
int [] grades; //también se puede hacer así: int grades[]; pero es preferible
//la opción int[].
```

La manera de declarar un *array* es usando []. El modo de inicializar una referencia no es asignando un valor como en el caso de los tipos primitivos, sino usando el operador (y palabra reservada) new.

```
int [] numbers = 5; //error: no es un tipo primitivo, sino una referencia

int [] grades = new int[5]; //OK: dentro de [] indicamos el tamaño del array.
//Las 5 casillas del array son inicializadas con el valor por defecto del tipo
//primitivo, en este caso, 0.
```

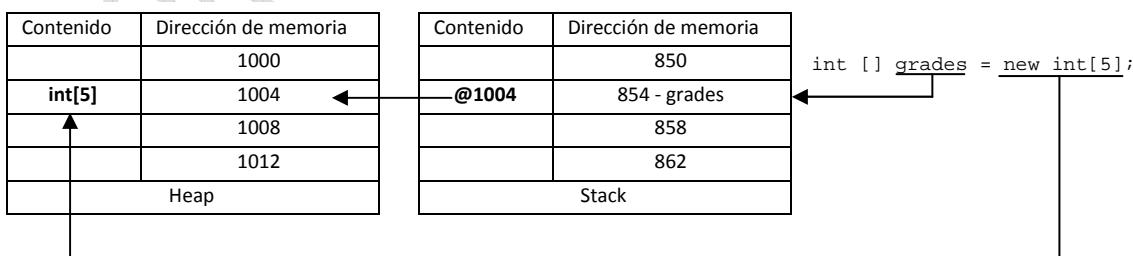
En la línea anterior estamos declarando un *array* llamado grades con 5 casillas de tipo primitivo int.

El operador new crea un objeto en una dirección de memoria concreta del *heap* y esa dirección de memoria se la asigna a la referencia, la cual se guarda en una zona de memoria del *stack*. Así pues, el valor de una referencia será una dirección de memoria.

<<Comentario al margen>>

Stack y heap

Los conceptos de *stack* y *heap* se estudian en profundidad en asignaturas relacionadas con los sistemas operativos.



Así pues, grades es una variable de tipo referencia ubicada en la posición 854 del *stack* y cuyo valor es la dirección de memoria (valor 1004) de un objeto *array* de 5 casillas de tipo int que está en el *heap*.

<<Comentario aclaratorio>>

Stack vs Heap

Cuando un programa se ejecuta, la JVM crea un hilo de ejecución (en inglés, *thread*) con un *stack* privado. En el *stack* se guardan variables locales, variables de referencia, el control de las invocaciones a métodos, los valores de los parámetros y de retorno de un método, resultados parciales, etc. Este espacio de memoria se rige por un comportamiento de pila (*stack*), es decir, LIFO (*Last-In, First-Out*).

En cambio, el *heap* es un espacio de memoria dinámica que la JVM adquiere sólo arrancarse la JVM y es único. Por lo tanto, el *heap* es común a todos los *threads* de ejecución. En el *heap* se guardan los objetos.

Por todo lo anterior, el tamaño del *stack* es mucho menor que el del *heap*.

Así pues, una vez creado el objeto de tipo *array* de *int*, podríamos asignar un valor a una de sus casillas:

```
grades[3] = 10;
```

Hemos asignado el valor 10 a la cuarta casilla del *array* *grades*. De igual manera que ocurre en otros lenguajes (p.je. C/C++), la primera casilla de un *array* en Java se corresponde con el índice 0.

Para imprimir por pantalla debemos usar el comando *System.out.print()* o *System.out.println()*. El segundo añade un salto de línea, es decir, un \n. Dicho esto, analicemos el siguiente código:

```
int number = 8;

int [] grades = new int[5];

grades[3] = 10;

System.out.println(number); //Imprime 8

System.out.println(grades[0]); //Imprime 0 (valor por defecto)

System.out.println(grades[3]); //Imprime 10

System.out.println(grades); //Imprime I@98af3857 --> dirección de memoria
```

Si miramos la última línea del código anterior, vemos que lo que se imprime por pantalla no es ningún valor del *array*, sino un <<texto extraño>>. Ese <<texto extraño>> no es más que la dirección de memoria del *heap* donde está alojado el objeto, que es un *array* de *int*. **A las variables que como *grades* guardan una dirección de memoria, en Java, se les llamas variables de tipo referencia o, simplemente, referencias.**

A diferencia de lenguajes como C/C++, en Java no existe el tipo puntero (o apuntador) como tal:

```
int* agePointer; //Código C/C++
```

El símbolo * nos está diciendo que `agePointer` es una variable que almacena la dirección de memoria –es decir, un puntero– de otra variable que es de tipo `int`. Como `agePointer` es una variable, el compilador le asignará una dirección de memoria:

Contenido	Dirección de memoria
36	1000
@1000	1004
	1008
	1012

```
//Código C/C++  
  
int age = 36;  
  
int* agePointer;  
  
agePointer = &age;
```

Al declarar la variable `agePointer` de tipo puntero de `int`, el compilador le ha asignado la posición de memoria 1004. En la última instrucción del código anterior, le hemos asignado la dirección de memoria de la variable `age`. Como podemos ver, el operador & en C/C++ devuelve la dirección de memoria de una variable.

A partir de aquí, puedo usar tanto `age` como `agePointer` para modificar el valor de `age`, ya que `agePointer` sabe en qué dirección de memoria está guardado el valor de la variable `age`. Así pues:

```
//Código C/C++  
  
age = 80;  
  
*agePointer = 80; //El símbolo * es el Operador de desreferencia o también  
//llamado de indirección
```

Las dos instrucciones anteriores harían exactamente lo mismo, es decir, asignarían el valor 80 a la variable `age`. Así pues, con cualquiera de las dos instrucciones anteriores, la memoria en nuestro ejemplo quedaría así:

Contenido	Dirección de memoria
80	1000
@1000	1004
	1008
	1012

¿Qué ventajas tiene usar punteros? La ventaja reside en poder comunicarse a más bajo nivel con el hardware, haciendo que la ejecución del código sea más rápida. Por ejemplo, trabajar con estructuras de datos complejas es más sencillo utilizando punteros que variables. Así pues, pasar direcciones de memoria a una función es más rápido que pasar una copia entera de la variable (p.ej. una estructura de datos). Además el uso de punteros permite aplicar operaciones aritméticas (p.ej. sumar y restar) sobre direcciones de memoria para llegar a otros datos que están en otras zonas de memoria. Todo esto es útil en ámbitos como la creación de sistemas operativos.

Sin embargo, el uso de punteros también trae consigo problemas, tales como:

- Pérdida de robustez y seguridad al intentar acceder a zonas de memoria prohibidas por el programa.
- Dificulta la liberación automática de memoria.
- Dificulta la defragmentación de memoria.
- Dificulta la programación distribuida de forma clara y eficiente.

Llegados a este punto, ¿y cómo se relacionan las referencias y los punteros en Java? Pues bien, de manera muy informal y superficial, podríamos decir que:

Referencia en Java

Una referencia en Java es como un puntero de C/C++ con la sintaxis de un tipo primitivo y con unas restricciones más fuertes.

A partir de la definición anterior, para declarar una referencia declararemos una variable de la misma manera que hacíamos con los tipos primitivos, pero ahora el tipo será un *array* o una clase (ya sea propia de la API de Java o creada por nosotros).

<<Comentario al margen>>

Punteros en Java

Java no tiene punteros explícitamente, pero sí implícitamente. Es decir, las referencias están programadas internamente como punteros, pero estos punteros no son accesibles para los programadores.

```
double[] grades; //grades es una referencia a un array de double cuyo valor es
//null.

Date birthdate; //birthdate es una referencia a un objeto de tipo Date. Date
//es una clase que existe en la API de Java.

Person david; //david es una referencia a un objeto de tipo Person (clase
//creada por nosotros)

Person [] people = new People[3]; //array de objetos Person, cada casilla es
//una referencia a un objeto Person con valor null. A su vez, people es una
//referencia al array de objetos Person
```

El hecho de que Java no use el término <<dirección de memoria>> o <<puntero>>, sino <<referencia>>, es porque el término <<referencia>> añade un significado más genérico y más estricto. De hecho, propiamente dicho, una variable de referencia almacena la dirección de memoria de un objeto.

Cuando se declara una variable de tipo referencia sin usar el operador new, el valor por defecto es null. Así pues, en el ejemplo anterior, tanto grades como birthdate y

david son tres referencias que no apuntan a ninguna zona de memoria del *heap*, es decir null. En cambio, sí ha creado un objeto de tipo *array* con 3 elementos Person. La dirección de memoria del *heap* de ese objeto se lo ha asignado como valor a la referencia people. Por lo tanto, people es una referencia a una zona de memoria del *heap*, p.ej. I@78ab8876. Sin embargo, cada casilla del *array* people es una referencia que no apunta a ninguna zona del *heap*, es decir, son null. En el caso de las variables grades, birthdate y david, así como para cada casilla de people, aún no se les asignado un objeto con el operador new (o asignándole otra referencia) y apuntan a null. Sin embargo, si hacemos:

```
Date birthdate = new Date(2020,0,11); //birthdate es una referencia a un
//objeto de tipo Date. Date es una clase que //existe en la API de Java.

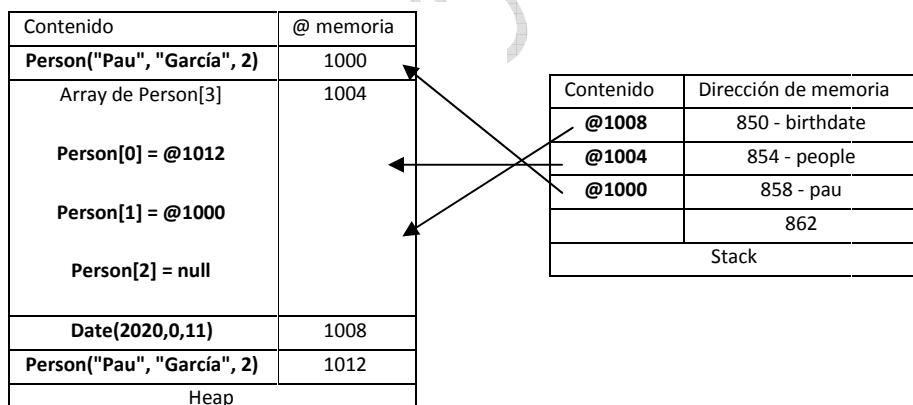
Person pau = new Person("Pau", "García", 2); //el constructor de Person recibe
//tres argumentos: name, surname //age.

Person [] people = new People[3];

people[0] = new Person("Pau", "García", 2); //es una referencia distinta a la
//de pau. Es decir, people[0] y pau son dos objetos aunque se inicializan con
//los mismos valores.

people[1] = pau; //la casilla 1 de people apunta a la misma posición de
//memoria del heap que pau. Es decir, people[1] y pau son la "misma"
//referencia.
```

El código anterior de manera gráfica sería:



Java proporciona al programador referencias y no punteros para evitar los peligros de los punteros, p.ej. aplicar operaciones aritméticas sobre los punteros o dejar memoria reservada dinámicamente (i.e. usando malloc, calloc o realloc) sin liberar (sin haber usado free). Como ya hemos comentado, Java es muy estricto con las referencias en comparación a como C/C++ lo es con los punteros. Algunas de estas restricciones que impone Java a las referencias son:

- Un programador no puede cambiar explícitamente el valor de una variable de tipo referencia. El siguiente código sería incorrecto:

```
Person pau = 10458;
```

- La única manera que una variable de tipo referencia obtiene su valor es mediante la asignación de un objeto. Las siguientes dos instrucciones son correctas:

```
Person pau = new Person("Pau", "García", 2);
Person student = pau;
```

- Como consecuencia de las dos restricciones anteriores, un programador no puede aplicar operadores aritméticos sobre las referencias.
- El único capaz de manipular directamente las direcciones de memoria es el JVM.

3.5. Operadores

Una vez ya sabemos cómo declarar e inicializar variables, es hora de hacer <> con ellas. En este apartado vamos a ver los principales operadores de Java. Pero antes, definamos qué es un operador:

Definición de operador

Igual que en matemáticas, los operadores son símbolos del lenguaje que indican que debe realizarse una operación específica sobre un cierto número de operandos y devolver un resultado fruto de aplicar dicha operación a los operandos. Por su parte, un operando es una entrada de un operador.

Antes de continuar, cabe definir el concepto de *expresión*.

Definición de expresión

Es una combinación de valores, funciones que retornan un valor, operadores y otras expresiones que, utilizando reglas de precedencia de los operadores que las forman, generan un resultado. Dicho resultado es otra expresión.

Así pues, tres ejemplos de expresión podrían ser: $2 + 7 - add(8,2)$, -1 y $(3 * 2) + 8$.

Un aspecto clave cuando se trata el tema de los operadores es la precedencia de los mismos. Cuando las expresiones son complejas es importante saber cuál es la precedencia que tiene cada operador dentro de la expresión –es decir, qué operador se aplica primero– con el objetivo de lograr el resultado esperado. A continuación se muestran los operadores de Java ordenados de mayor a menor precedencia. Es decir, el operador que está más arriba en la tabla se debe aplicar antes que un operador que está más abajo.

Operador	Precedencia					
postfix	expr++	expr--				
unario	++expr	--expr	+expr	-expr	~	!
multiplicativo	*	/	%			
adición	+	-				
shiftado	<<	>>	>>>			

relacional	<	>	<=	>=	instanceof							
igualdad	==	!=										
bitwise AND	&											
bitwise OR exclusiva	^											
biwise OR inclusiva												
AND condicional	&&											
OR condiconal												
ternaria	?	:										
asignación	=	+=	-=	*=	/=	%=	&=	^=	=	<<=	>>=	>>>=

Los operadores que están en la misma fila en la tabla anterior, tienen la misma precedencia. En este caso, la regla que se sigue es que todos los operadores binarios (i.e. dos operandos), excepto los de asignación, son evaluados de izquierda a derecha, mientras que los operadores de asignación (última fila de la tabla), son evaluados de derecha a izquierda. Asimismo los paréntesis, igual que en las matemáticas, tienen la máxima precedencia y se suelen usar para cambiar el orden de evaluación. Veamos algunos ejemplos de expresiones para entender las reglas de precedencia:

Ejemplo	Equivalente	Resultado
1+2-3+4	((1+2)-3)+4	4
1*2%3/4	((1*2)%3)/4	0.5
1+2*3-4/5+6%7	1+(2*3)-(4/5)+(6%7)	12.2

A continuación vamos a ver los diferentes tipos de operadores según la frecuencia de uso.

3.5.1. Operadores de asignación, aritméticos y unarios

En este apartado veremos:

- La asignación simple
- Los operadores aritméticos
- Asignaciones aumentadas o compuestas
- Los operadores unarios

Asignación simple

Este es uno de los operadores más comunes en cualquier programa. Como su propio nombre indica, este operador consiste en asignar una expresión (operando derecho) a una variable (operando izquierdo), ya sea de tipo primitiva o referencia. Algun ejemplo podría ser:

```
int age = getAge()-(2*3); //asignamos el resultado de restarle seis (2*3) al
//valor devuelto por el método/funcióñ getAge().

int speed = 10; //asignamos a la variable "speed" una expresión que es un
//literal numérico, i.e. el entero 10. Después de la asignación "speed"
//almacenará el valor 10.

Person pau = new Person("Pau", "García", 2); //asignamos un objeto, es decir,
// "pau" es una referencia (almacena la dirección de memoria del heap del
//objeto Person creado).
```

Operadores aritméticos

Java proporciona los mismos operadores aritméticos que existen en el lenguaje matemático más uno nuevo, muy común en los lenguajes de programación, llamado <<módulo>> y cuyo símbolo es %. Este nuevo operador lo que hace es dividir de manera entera (es decir, sin decimales) el operando de la izquierda entre el operando de la derecha y devolver el resto de dicha división.

Operador	Descripción	Ejemplo	Resultado	Tipo de resultado
+	Suma	2+3	5	Numérico
-	Resta	5-6.5	-1.5	Numérico
*	Multiplicación	7*3	21	Numérico
/	División	10/2	5	Numérico
%	Módulo	10%7	3	Entero (int)

Los operadores anteriores se pueden aplicar entre diferentes tipos de datos primitivos que sean numéricos, es decir, queda excluido el tipo boolean. Por ejemplo, en la tabla anterior para el operador -, hemos realizado la resta de un int (5) con un float (6.5), siendo el resultado de la operación un valor numérico de tipo float.

Llegados a este punto hay que tener presente el valor que resultante de aplicar un operador. En este caso debemos distinguir varias casuísticas:

- **Operandos del mismo tipo (casos int, long, float o double):** si los dos operandos que intervienen en un operador aritmético son del mismo tipo, el resultado será de ese mismo tipo. Así pues, si miramos la suma: int + int = int, long + long = long, float + float = float y double + double = double. Lo mismo ocurriría con la resta, la multiplicación, la división y el módulo.

La regla anterior afecta a la división entre enteros (int), ya que el resultado real podría ser un número decimal (i.e. float o double), pero el resultado obtenido aplicando la regla anterior es que será un entero (i.e. sin decimales). Así pues, $1/2 = 0$. Sin embargo, $1.0/2.0 = 0.5$. En este caso, si queremos forzar que el resultado sea un float o un double, debemos hacer un *cast explícito*, es decir, una conversión explícita del tipo, p.ej. (float) $(1/2) = 0.5$.

<<Comentario al margen>>

Cast explícito

Consiste en indicar al compilador que queremos que se realice una pérdida de precisión (truncado) al pasar de un tipo a otro más pequeño y que somos completamente conscientes de esta posible pérdida durante la conversión, p.ej. `int a = (int)3.5; //a valdrá 3`. Si no hacemos el cast explícito cuando hay posible pérdida de precisión (p.ej. de double, float y long hacia int), entonces el compilador nos dará un error.

- Operandos del mismo tipo (caso byte, short y chart):** si los dos operandos son byte o short o char, entonces las operaciones aritméticas son realizadas como si los operandos fuesen int (*cast* implícito porque no hay pérdida de precisión) siendo el resultado un int. Por ejemplo, byte + byte = int + int = int. Si se quiere recuperar el tipo original, hay que hacer una conversión explícita (i.e. *cast explícito*) del resultado:

```
byte b1 = 1, b2 = 3, b3;

b3 = (byte)(b1 + b2); //si hubiéramos escrito b3 = b1 + b2;
//hubiéramos obtenido un error porque el resultado de b1 + b2 es un int
//y lo estamos asignando a una variable de tipo byte.
```

En el caso del tipo char, el valor int asignado a un carácter será su correspondiente número Unicode dentro del rango [0, 65535].

Para estos tres tipos –byte, short y chart–, Java siempre hace las operaciones aritméticas como si fuesen int. Es decir, los promociona (les aplica un *casting* implícito) a int.

- Operandos de tipos distintos:** en este caso, el operando de tipo más pequeño es automáticamente promocionado al tipo más grande –es decir, se hace un *casting* implícito–, llevándose la operación en el tipo más grande y siendo el resultado del tipo más grande. El orden de promoción en Java es el siguiente: int → long → float → double. Así pues: 1 / 2.0 → 1.0 / 2.0 = 0.5 (double).

En este tipo de operaciones, debemos tener presente que si queremos asignar a una variable el resultado de una expresión como 5 - 6 . 5, entonces dicha variable debe ser del tipo mayor, en este caso float o double.

Llegados a este punto cabe resaltar la necesidad de ser consciente de que al hacer un operación aritmética sobre dos números enteros (int) muy grandes, el resultado es posible que no se pueda expresar como int (porque se salga del intervalo válido de datos), por lo que el resultado será un long. Incluso puede darse el caso de salirse de los límites inferior o superior de un double.

Finalmente indicar que el operador + puede usarse para concatenar (es decir, unir) cadenas de caracteres, que en Java son objetos de la clase String (lo veremos más adelante). Por ejemplo:

```
String texto1 = "Hola ";
String texto2 = "David!";

String texto1 = texto1 + texto2; //equivalente a texto1+=texto2; después de
//la asignación el valor de texto1 es "Hola David!"
```

Asignaciones aumentadas o compuestas

Por otro lado, los operadores aritméticos de la tabla anterior permiten combinarse con el operador de asignación, creando nuevos operadores denominados **asignaciones aumentadas o compuestas**. Estos operadores lo que hacen es aplicar asignar a la variable que hay en el lado izquierdo el resultado de aplicar el operador aritmético al valor de la variable (operando izquierdo) y a la expresión de la derecha (operando derecho). Veamos en la siguiente tabla estos operadores suponiendo que el valor de la variable number es 10.

Operador aumentado	Ejemplo	Equivalente	Nuevo valor de number
<code>+=</code>	<code>number += 3</code>	<code>number = number + 3</code>	13
<code>-=</code>	<code>number -= 6</code>	<code>number = number - 6</code>	4
<code>*=</code>	<code>number *= 3</code>	<code>number = number * 3</code>	30
<code>/=</code>	<code>number /= 2</code>	<code>number = number / 2</code>	5
<code>%=</code>	<code>number %= 7</code>	<code>number = number % 7</code>	3

<<Comentario al margen>>

Azúcar sintáctico

El azúcar sintáctico (en inglés, *syntactic sugar*) es una nueva sintaxis que se añade a un lenguaje de programación para hacer algunas operaciones ya existentes de manera más sencilla y abreviada. Esta nueva sintaxis hace “más dulce” el lenguaje. Un ejemplo es el operador: `+=`.

Existe una pequeña diferencia entre los operadores simples (`+`, `-`, `*`, `/`, `%`) y los compuestos que acabamos de ver:

```
byte b1 = 1, b2 = 3;

b2 = (b1 + b2); //error: el resultado es un int, hay pérdida de precisión al
//intentarlo asignar a una variable de tipo byte. Hay que hacer casting
//explícito.

b2 += b1; //correcto, la asignación compuesta hace el cast por nosotros
```

Operadores unarios

Este tipo de operadores sólo utilizan un operando. Los operadores unarios de Java son:

Operador	Descripción	Ejemplo	Descripción
<code>+</code>	Signo positivo	<code>number = +3;</code>	El <code>+</code> indica que el número 3 es positivo
<code>-</code>	Signo negativo	<code>number = -3;</code>	El <code>-</code> indica que el número 3 es negativo
<code>++ (sufijo)</code>	Incremento	<code>int number = 3; int a = number++;</code>	Su código equivalente es: <code>int number = 3; a = number; //a vale 3 number = number + 1; //number vale 4</code>

++ (prefijo)	Incremento	int number = 3; int a = ++number;	Su código equivalente es: int number = 3; number = number + 1; //number vale 4 a = number; //a vale 4
-- (sufijo)	Decremento	int number = 3; int a = number--;	Su código equivalente es: int number = 3; a = number; //a vale 3 number = number - 1; // number vale 2
-- (prefijo)	Decremento	int number = 3; int a = --number;	Su código equivalente es: int number = 3; number = number - 1; //number vale 2 a = number; // a vale 2
!	Negación	boolean a = false; boolean b = !a;	La variable b después de la asignación será true (el valor contrario al de a).

Como puedes ver en la tabla anterior, una de las particularidades que tienen los operadores `++` y `--` es que pueden aplicarse como prefijo o sufijo. Veamos un ejemplo más extenso de las diferencias:

```
int a = 0;

int b = a++; //Tras esta sentencia, b vale 0 y a vale 1
int c = ++a; //Tras esta sentencia, c vale 2 y a vale 2

int d = (a += 1); //Tras esta sentencia, d vale 3 y a vale 3
```

3.5.2. Operadores de igualdad, relacionales y condicionales

Los operadores de este apartado los veremos en tres grupos:

- Operadores de igualdad y relacionales.
- Operadores condicionales
- Operador `instanceof` (pertenencia a una clase o interfaz)

Operadores de igualdad y relacionales

Los operadores de estas dos tipologías, también denominados bajo el término <>operadores comparativos>>, permiten comparar dos operandos. El resultado de estos dos operando será un boolean, es decir, `true` o `false`.

Operador	Descripción	Ejemplo	Resultado
<code>==</code>	Igual que	3 == 3.0 true == false	true false
<code>!=</code>	Distinto que	int b = 5; b != 10; b != 5;	true false
<code>></code>	Mayor que	15 > 10 7 > 10	true false
<code>>=</code>	Mayor o igual que	8 >= 7 7 >= 8	true false
<code><</code>	Menor que	-6.5 < 6.5 15 < (13.75+1)	true false
<code><=</code>	Menor o igual que	3.2 < 3.3 10 <= 5	true false

Como se puede ver, se pueden comparar expresiones cuyos resultados sean de tipos numéricos diferentes. El compilador se encarga de realizar las conversiones oportunas antes de hacer la comparación. Asimismo, los operadores == y != aceptan también valores de tipo boolean.

Operadores condicionales

En esta categoría sólo existen dos operadores: el AND condicional y el OR condicional. Estos dos operadores actúan sobre dos expresiones boolean.

En el caso del AND condicional, éste devuelve true si los dos operandos son true. Mientras que el operador OR condicional devuelve true cuando al menos uno de los dos operandos es true. Estos dos operadores se relational con los operadores lógicos · y + del álgebra de Boole, respectivamente. También se puede hacer su equivalencia con las puertas lógicas AND y OR, respectivamente. Por todo lo anterior, una manera sencilla de obtener el resultado de un AND y un OR es convertir mentalmente los operandos false en 0, los true en 1, el AND por una multiplicación y el OR por una suma. Si el resultado de hacer la multiplicación o la suma es 1, entonces el resultado definitivo es true y, en caso contrario, es false. Hay que tener presente que en el álgebra de Boole $1+1 = 1$.

Operador	Descripción	Ejemplo	Resultado
&&	AND condicional	(3 == 3.0) && (50<100) false && true	true false
 	OR condicional	(3 == 3.0) (50<100) false true (3>5) (5<4)	true true false

Los dos operadores anteriores tiene, además, un comportamiento que se denomina de <<cortocircuito>>, puesto que el segundo operando es evaluado sólo si es necesario. ¿Cuándo es necesario evaluar la expresión del segundo operando? Cuando con el resultado del primer operando no se puede determinar con seguridad el resultado de la operación AND u OR. Por ejemplo:

```
(3 == 3.0) || (50<100) // No evaluará (50<100) porque (3==3.0) es true y el
//operador || es true cuando al menos uno de los dos operandos es true. Así
//pues, da igual si el segundo operando, es decir, (50<100) es true o no,
//porque el resultado será true debido a que el primer operando es true.

false && (50<100) // No evaluará (50<100) porque false es false y el operador
//&& sólo es true cuando los dos operandos son true.
```

Quizás te preguntes, ¿y por qué es importante saber que AND y OR se comportan como un cortocircuito? Con el siguiente ejemplo esperemos que lo veas claro:

```
false && doSomething();
```

Imagina que la función/método doSomething() hace una serie de operaciones y devuelve true o false en función de esas operaciones. Además, dentro de esas operaciones que realiza, hay una que es contar las veces que ha sido llamada esta función. Pues bien, en el

ejemplo anterior, dicho contador sería siempre cero porque jamás se llamaría a doSomething().

Operador instanceof

Este operador compara un objeto con un tipo de clase concreta. Es decir, se usa para saber si una instancia/objeto pertenece a una clase concreta (ya sea padre directo o no) o implementa una determinada interfaz.

<<Comentario al margen>>

Clase e interfaz

Estos dos conceptos se verán más adelante tanto en esta guía como en los módulos teóricos de diseño y programación orientados a objetos.

Un ejemplo de uso de este operador podría ser:

```
Person pau = new Person("Pau", "García", 2);

pau instanceof Person; //devolvería true ya que la referencia //pau almacena
//un objeto de la clase Person

pau instanceof Ball; //devolvería false porque pau no es una //referencia a un
//objeto de la clase Ball.
```

3.5.3. Operadores bitwise y shiftado

En este apartado veremos operadores que trabajan directamente con los bits. Estos operadores son mucho menos utilizados que los vistos hasta ahora y se clasifican en:

- Operadores bitwise (también conocidos como <<operadores a nivel de bits>>)
- Operadores de shiftado (u <<operadores de desplazamiento de bits>>)

Operadores bitwise

Estos operadores manipulan bits, es decir, 0 y 1.

Operador	Descripción	Tipo operandos	Ejemplo	Resultado
&	AND de bits	byte, short, int, long, char (tratado como int) y boolean	(3 == 3.0) & (50<100) false & true 60 & 13	true false 12
	OR inclusiva de bits	byte, short, int, long, char (tratado como int) y boolean	(3 == 3.0) (50<100) true false (3>5) (5<4) 60 & 13	true true false 61
^K	XOR (u OR exclusiva) de bits	byte, short, int, long, char (tratado como int) y boolean	5^3 (3 == 3.0) ^ (50<100) true ^ false	6 false true
~	Complemento a 1	byte, short, int, long y char	~5 = 250	65530

Los operadores AND (&) y OR (|) de bits cuando los operandos son boolean actúan como hemos comentado con sus homónimos condicionales, pero sin el comportamiento de

<<cortocircuito>>. Es decir, independientemente del valor de la expresión del primer operando, el segundo operando siempre se evaluará. Así pues:

```
false & doSomething(); //siempre se llamará a doSomething();
```

El operador XOR (u OR exclusiva) lo que hace es devolver 1 cuando dos bits de la misma posición son distintos entre los dos operandos. En caso contrario devuelve 0. De hecho, su equivalente con AND y OR lógicos sería: (A AND !B) OR (!A AND B). Veamos el ejemplo de la tabla:

```
5^3 = (00000101)^ (00000011) = 00000110 = 6
```

Este operador también se puede aplicar sobre operandos cuyo resultado sea un boolean. En este caso, cuando los dos operandos coinciden en valor el resultado es false y, en caso contrario, el resultado es true. Así pues: true^true es false, false^false es false, true^false es true y false^true es true.

El operador complemento a 1 (~) lo que hace es convertir la representación binaria del operando y cambiar los ceros por unos y los unos por cero. En el ejemplo de la tabla los pasos internos son:

```
~5 = ~(00000101) = 11111010 = 250
```

Operadores de shiftado

Estos operadores manipulan bits desplazándolos hacia la derecha o hacia la izquierda.

Operador	Descripción	Ejemplo	Resultado
<<	Desplaza el operando de la izquierda hacia la izquierda tantas veces como indica el operando de la derecha y las posiciones shiftadas son rellenadas con ceros.	60 << 2	240 (11110000)
>>	Desplaza el operando de la izquierda hacia la derecha tantas veces como indica el operando de la derecha.	60 >> 2	15 (1111)
>>>	Desplaza el operando de la izquierda hacia la derecha tantas veces como indica el operando de la derecha y las posiciones shiftadas son rellenadas con ceros.	60 >> 2	15 (00001111)

3.6. Bloques de flujo de ejecución

Como ya comentamos en el apartado 3.2, los bloques son una agrupación de instrucciones que son tratadas como una única unidad. En Java podemos distinguir 3 tipos de bloques que afectan al flujo de ejecución del programa, es decir, determinan qué instrucciones se ejecutan y cuáles no: (1) bloque secuencial, (2) bloque condicional y (3) bloque iterativo.

3.6.1. Bloque secuencial

Un programa, de por sí, es una secuencia de instrucciones. Es decir, las instrucciones se ejecutan en el orden en el que están escritas (siempre de arriba abajo). Es por eso que es el bloque más común.

```
int number = 10; //1ª instrucción en ejecutarse
number += 3; //2ª instrucción en ejecutarse, se ejecuta después de int number = 10;
System.out.println(number); //3ª instrucción y última en ejecutarse
```

3.6.2. Bloque condicional

Un bloque condicional es aquél que sirve para ejecutar o no una serie de instrucción según una condición. Dentro de este tipo de bloque, también llamado de decisión o selección, existen varias alternativas.

Bloque if

Este bloque sirve para denotar un conjunto de instrucciones que se deben ejecutar si se cumple una expresión lógica, es decir, si dicha expresión es true.

Patrón del bloque	Ejemplo
<pre>if(condición){ //Código: se ejecuta si la condición //es true } //Próxima instrucción</pre>	<pre>int number = 10; if(number >= 10){ number += 6; } number -= 5; //resultado 9</pre>

Aunque no es recomendable, si el código que hay dentro de un bloque if es sólo una instrucción, se puede obviar el uso de las llaves. El siguiente código sería equivalente al anterior:

```
int number = 10;

if(number >= 10)

    number += 6;

number -= 5; //resultado 9
```

Bloque if-else

Este bloque sirve para indicar dos bloques de instrucciones que son excluyentes entre sí. El primer bloque (bloque dentro del if) se ejecutará si se cumple una expresión lógica, es decir, si dicha expresión es true. En caso contrario, se ejecutará el segundo bloque (bloque dentro del else).

Patrón del bloque	Ejemplo
<pre>if(condición){ //Código: se ejecuta si la condición //es true }else{ //Código: se ejecuta si la condición //es false } //Próxima instrucción</pre>	<pre>int number = 10; if(number > 10){ number += 6; }else{ number *= 5; } number -= 5; //resultado 45</pre>

Bloque if anidado o else-if

Dentro de un if o un else se puede escribir/anidar otro bloque condicional, pero también existe una opción más corta –llamada if anidado– para el caso de anidar dentro del else.

Patrón del bloque	Ejemplo
<pre>if(condición1){ //Código: se ejecuta si la condición1 //es true }else if(condición2){ //Código: se ejecuta si la condición1 //es false y la condición 2 es true }else{ //Código: se ejecuta si la condición1 //y la condición 2 son false } //Próxima instrucción</pre>	<pre>int number = 8; if(number > 10 && number < 15){ number += 6; }else if(number > 8){ number *= 5; }else{ number += 3; } number -= 5; //resultado 45</pre>

Si nos fijamos, este bloque no deja de ser un if-else en el que el else no tiene la llave {. El último else no es necesario ponerlo si no se requiere por la lógica del programa. Es decir, si no queremos hacer nada cuando todas las condiciones son falsas, entonces no hace falta poner un bloque else vacío (i.e. sin instrucciones).

Operador ternario

Existe un operador condicional llamado ternario que compacta un if-else más una la devolución de un valor en una única instrucción.

Patrón del bloque	Ejemplo
<pre>condición ? exprTrue : exprFalse; El resultado es exprTrue si condición ese true. En caso contrario, el resultado es exprFalse.</pre>	<pre>int number = (8>10) ? 5 : 3; number -= 2; //resultado 1</pre>

Este operador es muy utilizado para asignar valores a una variable en función de una condición.

Bloque switch

Este bloque es una alternativa a la concatenación de bloques if anidados para valores fijos (no rangos) de una expresión que hace de selector. Los valores del selector pueden ser byte, short, int, char o String (i.e. texto/cadena de caracteres; desde JDK 1.7).

Patrón del bloque	Ejemplo
<pre>switch(selector){ case valor1: //Bloque de código; break; case valor2: //Bloque de código; break; ... case valorN: //Bloque de código; break; default: //Bloque de código; } //Próxima instrucción</pre>	<pre>int number = 2, age = 30; switch(number){ case 0: age = 50; break; case 1: case 2: age = 25; break; default: age = 10; } age -= 5; //resultado 20</pre>

Los casos se evalúan de manera secuencial hasta que su valor coincide con el del selector y entonces entra en su bloque. Así pues, no es lo mismo poner el `case 0` primero que ponerlo después del `case 2`, por ejemplo. Una vez se entra en un caso, se irán ejecutando todos los bloques hasta que uno de ellos contenga un `break`. Cuando se encuentra el `break`, el programa sale del `switch`.

Asimismo, el caso `default` no es necesario ponerlo si no se requiere su uso, cuyo código se ejecuta si ningún caso coincide con el valor del selector. Es decir, en el ejemplo anterior, si `number` hubiera sido 1, entonces se ejecutaría el bloque del `case 1`, pero al no haber `break`, se ejecutaría también el bloque del `case 2`, asignándole el valor 25 a la variable `age`. Como el `case 2` tiene un `break`, la ejecución sale del `switch` y pasa a la instrucción `age -= 5`. Así pues, el programa se comporta de manera idéntica tanto si el valor de `number` es 1 como si es 2.

Desde JDK 13 en adelante se puede usar la palabra reservada `yield` para devolver un valor al terminar un caso y, por consiguiente, terminar el bloque `switch`.

```
int numPeople = 0;

int number = switch(numPeople){

    case 0:

        System.out.println("Hola!");

        yield 1;

    case 1:

        age++;

        height = 150;

        yield 15;

    default:

        yield -1;

};

//En este punto number tendrá el valor 1 y se habrá imprimido por pantalla el
//texto "Hola!".
```

3.6.3. Bloque iterativo

Este tipo de bloque, también llamado bloque de bucle, sirve para ejecutar un conjunto de instrucciones repetidamente hasta que deja de cumplirse una condición. Existen varias alternativas.

Bloque `while`

Ejecuta el código que hay dentro de sus llaves mientras su condición sea `true`. Cuando la condición sea `false`, entonces finaliza el bucle y se pasan a ejecutar las instrucciones que hay después del bloque `while`. Cabe resaltar que la condición se evalúa en cada iteración.

Patrón del bloque	Ejemplo
<pre>while(condición){ //Código a repetir } //Próxima instrucción</pre>	<pre>int index = 3, number = 0; boolean end = false; while(index>0){ number++; index--; } number += 2; //resultado 5 while(!end){ number++; if(number>=7){ end = true; } } number += 2; //resultado 9</pre>

Bloque do-while

Este bloque es similar al anterior pero la condición se evalúa por primera vez una vez se han ejecutado las instrucciones que hay dentro del bloque. No obstante, a cada iteración se evalúa la condición.

Patrón del bloque	Ejemplo
<pre>do{ //Código a repetir }while(condición); //Próxima instrucción</pre>	<pre>int index = 0, number = 0; do{ number = 3; }(index>0); number += 2; //resultado 5</pre>

En el ejemplo anterior, se ejecuta una vez el código que hay dentro del bloque do-while, puesto que la condición no se ha evaluado hasta que se pasa por la asignación `number = 3;`. Una vez hecha la asignación se comprueba si la condición es `true`. En este caso se ve que es `false` y se termina el bloque do-while.

A diferencia del bloque while, el código que hay dentro de las llaves del do-while siempre se ejecuta como mínimo una vez, mientras que en el caso del while puede llegar a no ejecutarse nunca.

Por último, es importante percatarse de que después de la condición del do-while debe escribirse un punto y coma (;).

Bloque for

Este bloque proporciona una manera compacta de iterar sobre un intervalo de valores.

Patrón del bloque	Ejemplo
<pre>for(inicialización; condición; incremento){ //Código a repetir } //Próxima instrucción</pre>	<pre>for(int i=1; i<10; i++){ System.out.println(i); } //imprime del 1 al 9</pre>

Cuando se usa este bloque hay que tener en cuenta que:

- La expresión de inicialización se ejecuta sólo la primera vez que se ejecuta el bloque. Es posible declarar una variable dentro de la expresión de inicialización. El alcance (*scope*) o visibilidad de esta variable está limitada al bloque `for` donde ha sido declarada. Es decir, dicha variable sólo existe dentro del bloque `for`. Si la variable que usamos para controlar el `for` no se necesita fuera del `for`, entonces la mejor práctica es declarar en la expresión de inicialización del propio `for` y no fuera de éste. De esta manera se limita el tiempo de vida de dicha variable. Los nombres típicos para estas variables que controlan el bloque `for` suelen ser `i`, `j` y `k`.
- El bloque (es decir, la iteración) termina cuando la condición es `false`.
- La expresión de incremento es invocada después de ejecutar el código que hay dentro de las llaves. Esta expresión puede ser de incremento o de decremento.
- Tanto en la expresión de inicialización como en la de incremento se pueden poner más de una instrucción separándolas con comas.

```
for(int row = 0, col = 0; col < 10; row++, col++){
    //Código a ejecutar
}
```

Bloque enhanced for

El bloque `for` anterior tiene otra sintaxis diseñada para iterar sobre colecciones llamada como *enhanced for*. Por colección nos estamos refiriendo tanto a un *array* como a una estructura de datos como puede ser una pila, una cola, etc. que la propia API de Java proporciona.

Patrón del bloque	Ejemplo
<pre>for(type nameVariable : nameCollection){ //Código a repetir para todos los elementos //que hay dentro de nameCollection } //Próxima instrucción</pre>	<pre>int[] numbers = {1, 2, 3}; for(int value : numbers){ System.out.println(value); } //imprime del 1, 2 y 3 en //líneas separadas</pre>

Si nos fijamos, en la variable `value` se almacena el valor actual del *array* `numbers`. Es decir, de por sí, el *enhanced for* crea un índice interno inicializado a 0 y lo va incrementando hasta llegar a la longitud (o tamaño) máxima de la colección.

El código anterior con un `for` simple sería:

```
int[] numbers = {1, 2, 3};

for(int i = 0; i < numbers.length; i++){
    System.out.println(numbers[i]);
}
```

En este tipo de bucles es interesante y útil utilizar la manera de declarar variables de JDK 10 (`var`) para que sea el propio compilador quien infiera el tipo de los objetos de la colección. Por ejemplo:

```
for(var value : numbers){  
    System.out.println(value);  
}
```

3.6.4. Instrucciones de control de flujo

En este apartado veremos tres instrucciones que permiten cambiar el flujo de ejecución de un programa.

Instrucción break

La instrucción `break` tiene dos modalidades: etiquetada y sin etiqueta. Esta última la hemos visto cuando hablamos del bloque `switch`. Sin embargo, también podemos usarla para terminar un bloque iterativo (i.e. `while`, `do-while` y `for`). Por ejemplo:

```
int[] numbers = {1, 2, 3};  
  
boolean found = false;  
  
for(int i = 0; i < numbers.length; i++){  
    if(numbers[i] == 2){  
        found = true;  
  
        break; //Cuando el valor de la casilla sea 2, terminará el bucle.  
    }  
}  
  
if(found){  
    System.out.println("Number 2 was found");  
}  
else{  
    System.out.println("Number 2 was not found");  
}
```

Por su parte, un `break` etiquetado permite finalizar aquél bloque iterativo que tenga la etiqueta que se indica en el `break`.

```
etiqueta: for(int i = 0; i < 10; i++){  
  
    for(int j = 0; j < 10; j++){  
        if(i == 5 && j == 3){  
            break etiqueta;  
        }  
        System.out.println(i);  
    }  
}
```

En el ejemplo anterior, cuando la variable `i` vale 5 y la variable `j` vale 3, se ejecuta el `break` etiquetado. Lo que hace es finalizar el bucle exterior que ha sido etiquetado con el nombre `<<etiqueta>>`. Si hubiéramos escrito un `break` sin etiqueta, entonces el bucle que inicializa la variable `j` sería la que hubiera terminado, e imprimiría el valor de la `i` en ese momento (i.e. 5) y continuará con el bucle exterior para seguir imprimiendo `i` igual a 6.

Instrucción continue

La instrucción `continue` se utiliza para saltar la iteración actual de un bloque iterativo (i.e. `while`, `do-while` o `for`). De manera homóloga a la instrucción `break`, la instrucción `continue` también tiene su forma etiquetada y no-etiquetada. Un ejemplo de versión no-etiquetada sería:

```
int[] numbers = {1, 2, 3};

for(int i = 0; i < numbers.length; i++){
    if(i == 1){

        continue; //Cuando i sea 2, la ejecución irá al inicio del bucle,
                   //obviando lo que hay a continuación para la iteración i = 2
    }

    System.out.println(i); //Imprimirá 1 y 3 en líneas separadas
}
```

Igual que con `break`, un `continue` etiquetado permite saltar la iteración de un bucle externo.

```
etiqueta: for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
        if(i == 5 && j == 3){

            continue etiqueta;

        }

        System.out.println(i);
    }
}
```

En el ejemplo anterior, sólo se imprimiría 5 tres veces, en vez de diez veces.

Instrucción return

La instrucción `return` se utiliza dentro de funciones/métodos y lo que hace es salir de la función para seguir el flujo de ejecución a partir de la instrucción que hay justo después de la llamada de la función/método. El `return` puede devolver un valor o no. Si devuelve un valor, el tipo del valor devuelto debe coincidir con el indicado en la firma de la función/método.

```
return 10;
return;
```

3.7. String

En este apartado vamos a ver un tipo referencia muy utilizado en Java: `String`. `String` es una clase que proporciona la API de Java y que nos sirve para almacenar cadenas/secuencias de caracteres, es decir, textos. Dado que la manipulación de textos es vital en cualquier programa, la clase `String` se ha convertido en un tipo primitivo de facto (aunque no lo es; es una clase, es decir, un tipo referencia). Cualquier cadena de caracteres está compuesta por cero o más caracteres todos ellos acotados por comillas dobles " .

```
String text = "Hola UOC!"; // "Hola UOC!" es un literal que se asigna a una
// referencia de tipo String

char letter = 'A'; // Los String se escriben entre " y los char entre '

String emptyMsg = ""; // Texto vacío
```

Excepcionalmente, en Java para instanciar un objeto de tipo `String` no es necesario llamar a su constructor, sólo es necesario asignarle una cadena de caracteres (i.e. un texto). Veremos que también se puede llamar a su constructor con la palabra reservada `new`, pero esto lo comentaremos más adelante, cuando veamos la diferencia entre las clases `String`, `StringBuilder` y `StringBuffer`.

3.7.1. Escape sequences

Existen diferentes secuencias de escape (*escape sequences*) que sirven para indicar alguna instrucción de formateo del texto. Por ejemplo:

- `\n` añade un salto de línea al final del texto poniendo el cursor en la siguiente línea.
- `\t` añade un tabulador allá donde aparece.

```
String text = "Hola UOC!\tHola\nEps!";

System.out.print(text);

/* El print imprimirá por pantalla:

Hola UOC!      Hola
Eps!
```

¿Qué ocurre si el texto tiene dentro unas comillas dobles o una contrabarra (o barra invertida)? Pues que el compilador se confundirá porque no sabrá qué comillas dobles son las del texto y cuáles las que limitan la cadena de caracteres. Lo mismo con la barra invertida, no sabrá si se trata de un comando o no. ¿Cómo solucionar este conflicto? Añadiendo una doble invertida delante de las comillas o de la barra invertida que queremos que sea parte del texto.

```
String text = "Hola \"UOC\"!\\"tHola"; System.out.print(text);

/* El print imprimirá por pantalla:

Hola "UOC"!\\tHola
Eps!
```

3.7.2. Concatenar con +

Cuando vimos los operadores aritméticos, dijimos que el operador + tenía un comportamiento especial con los String. Concretamente comentamos que sirve para concatenar una cadena de caracteres – es decir, String–, con otro elemento (i.e. otro String o un tipo primitivo).

```
String text = "My name is ";

System.out.print(text+"David"); //Escribe por pantalla My name is David

System.out.print(text+"David and I'm "+36); //My name is David and I'm 36

int age = 41;

text += "Elena and I'm 41";

System.out.print(text); //My name is Elena and I'm 41
```

En el caso de concatenar un String con cualquier otro tipo de valor, p.ej. un int, este último se convierte en String y ambos String (i.e. el original y el int convertido en String) se concatenan dando como resultado un nuevo String.

3.7.3. Algunos métodos interesantes

La clase String proporciona muchos métodos, los cuales nos facilitan muchas tareas. Puedes ver los métodos disponibles en la documentación oficial: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html>

Firma del Método	Descripción	Ejemplo
<code>char charAt(int index)</code>	Devuelve el carácter que se encuentra en la posición index	String text = "Hola!"; char c = text.charAt(0); System.out.print(c); //H
<code>boolean contains(CharSequence c)</code>	Dice si una cadena de caracteres está dentro del String o no.	String text = "Hola UOC"; boolean b = text.contains("UOC"); System.out.println(b); //true
<code>boolean equals(String another)</code>	Comprueba si un String es igual a otro.	String text = "UOC"; System.out.print(text.equals("UOC")); //true System.out.print(text.equals("uoc")); //false
<code>boolean equalsIgnoreCase(String another)</code>	Comprueba si un String es igual a otro ignorando mayúsculas y minúsculas.	String text = "UOC"; System.out.print(text.equalsIgnoreCase("UOC")); //true System.out.print(text.equalsIgnoreCase("uoc")); //true
<code>int indexOf(String substring)</code>	Devuelve el índice en el que comienza el substring dentro del String. Si no está, devuelve -1.	String text = "Patata"; System.out.print(text.indexOf("t")); //2 System.out.print(text.indexOf("ata")); //1
<code>boolean isEmpty()</code>	Comprueba si el String es vacío ("").	String text = "UOC"; String empty = ""; System.out.print(text.isEmpty()); //false System.out.print(empty.isEmpty()); //true

<code>int length()</code>	Devuelve el número de caracteres que tiene el texto incluyendo espacios	<pre>String text = "Hola!"; int n = text.length(); System.out.print(n); //5</pre>
<code>String replace(char old, char new)</code>	Devuelve el String con todas las apariciones del carácter old reemplazadas por el carácter new	<pre>String text = "Hola patata"; text = text.replace("a", "*"); System.out.print(text); //Hol* p*t*t*</pre>
<code>String[] split(String regex)</code>	Separa el String original en tantos String que estén separados por el String regex.	<pre>String text = "Hola-patata-UOC"; String[] texts = text.split("-"); System.out.print(texts[0]); //Hola System.out.print(texts[1]); //patata System.out.print(texts[2]); //UOC</pre>
<code>String substring(int beginIndex)</code>	Devuelve el subtexto que hay dentro del texto a partir del índice indicado	<pre>String text = "Hola UOC"; text = text.substring(5); System.out.print(text); //UOC</pre>
<code>String substring(int beginIndex, int endIndex)</code>	Devuelve el subtexto que hay dentro del texto desde el índice inicial hasta el índice final (no incluido)	<pre>String text = "Hola UOC"; text = text.substring(5, 6); System.out.println(text); //U</pre>
<code>String toLowerCase()</code>	Devuelve el String escrito todo en minúsculas	<pre>String text = "UOC"; text = text.toLowerCase(); System.out.print(text); //uoc</pre>
<code>String toUpperCase()</code>	Devuelve el String escrito todo en mayúsculas	<pre>String text = "uoc"; text = text.toUpperCase(); System.out.print(text); //UOC</pre>
<code>String trim()</code>	Elimina los espacios iniciales y finales del String	<pre>String text = " Hola patata "; text = text.trim(); System.out.print("_"+text+"_"); //_Hola patata_</pre>

3.7.4. Conversiones

A menudo necesitamos convertir un valor en formato texto (i.e. String) en un tipo primitivo para hacer alguna operación. En otras ocasiones, tendremos un valor en formato primitivo y necesitaremos tenerlo en formato texto. Ambos casos son un tipo de conversión.

De tipo primitivo a String

Una manera sencilla y rápida de convertir un valor primitivo en un texto es concatenándole un texto (aunque sea vacío, i.e. " "). Por ejemplo:

```
int num = 50;

String text = 50+""; //ahora text es "50", este 50 es texto.
```

Otra manera de hacerlo, quizás más explícita y clara a simple vista es usando un método estático de la clase `valueOf` que sirve para todos los primitivos.

```
int num = 50;

String text = String.valueOf(num); //ahora text es "50", este 50 es texto.

text = String.valueOf(50.25); //ahora text es "50.25", este 50.25 es texto.

text = String.valueOf(false); //ahora text es "false", este false es texto.
```

Una última forma de convertir un tipo primitivo en un `String` es usando el método estático `toString` de cada clase *wrapper* de los tipos primitivos.

<<Comentario al margen>>

Clases *wrapper*

Más adelante veremos qué es una clase *wrapper*.

```
int num = 50;

String text = Integer.toString(num); //ahora text es "50", este 50 es texto.

text = Double.toString(50.25); //ahora text es "50.25", este 50.25 es texto.

text = Boolean.toString(false); //ahora text es "false", este false es texto.

text = Character.toString('c'); //ahora text es "c", esta c es texto.
```

De tipo String a tipo primitivo

Para obtener el valor de tipo primitivo de un texto podemos usar los métodos estáticos `parseXXX` que incluyen las clases *wrappers* de los tipos primitivos (excepto para `char`).

```
int i = Integer.parseInt("50"); //ahora i es 50.

double d = Double.parseDouble("50.25"); //ahora d es 50.25.

boolean b = Boolean.parseBoolean("true"); //ahora b es true.
```

En el caso del tipo `char`, debemos usar el método `charAt` de la clase `String` que hemos visto anteriormente.

```
String text = "A";

char c = text.charAt(0); //ahora c es 'A' y no "A".
```

3.8. Arrays

Ya hablamos muy superficialmente de los *arrays* cuando vimos el tipo referencia. En este apartado queremos profundizar en ellos. Podemos definir un *array* de la siguiente manera:

Definición de *array*

Es un tipo de dato estructurado que permite almacenar de manera ordenada un conjunto fijo de elementos (i.e. una colección) homogéneos, es decir, que son del mismo tipo y están

semánticamente relacionados entre sí. El tipo de los elementos almacenados puede ser primitivo o referencia. Cada elemento dentro del *array* tiene asignado un índice numérico (o posición) con el fin de encontrarlo únicamente. La primera posición o casilla del *array* es la número 0. Para acceder a una casilla hay que usar el operador [].

Cuando el *array* tiene una única dimensión se le suele llamar *array*, vector o arreglo. Cuando el *array* tiene dos dimensiones se le llama *array* de dos dimensiones, matriz o tabla. Cuando el número de dimensiones es mayor se le llama *array* de N dimensiones (siendo N el número de dimensiones), *array* de *arrays* o, simplemente, *array* multidimensional. También puede darse el caso de que el número de columnas en un *array* multidimensional sea desigual, dándose lo que en inglés se conoce como *jagged array*.

3.8.1. Declaración y creación en memoria

Un *array* será un objeto que será alojado en el *heap*. Para poder acceder a dicho objeto tendremos una variable de tipo referencia. El nombre de un *array* suele ser un sustantivo en plural.

```
int[] grades; //array de int

short[][] ages; //array de 2 dimensiones, o matriz o table
```

Con la declaración anterior no se crea ningún objeto en el *heap*. Las referencias *grades* y *ages* apuntan a null, es decir, ninguna dirección de memoria del *heap*. Para crear el objeto de tipo *array* hay que usar la palabra reservada *new*:

```
int[] grades = new int[10]; //array de 10 int

short[][] ages = new short[5][6]; //array de 5 filas y 6 columnas (6 casillas
//cada fila).

Person[][] people = new Person[2][]; //jagged array de 5 filas y con un número
//de columnas por determinar
```

Con el paso anterior, las referencias *grades*, *ages* y *people* apuntan a la dirección del *heap* de sus respectivos objetos *array*. Sin embargo, cada casilla de las tres *arrays* son inicializados con el valor por defecto (ver apartado 2.3.2). En el caso de los tipos primitivos, *int* y *short*, el valor por defecto es 0, mientras que en el caso de un tipo referencia como es un objeto de la clase *Person*, el valor por defecto es null.

Por lo que respecta a la referencia *people* aún faltan por inicializar las 2 filas:

```
Person[][] people = new Person[2][];

people[0] = new Person[3]; //accedemos a la fila 0 y le asignamos un array de
//3 casillas (columnas).

people[1] = new Person[4]; //accedemos a la fila 1 y le asignamos un array de
//4 casillas (columnas).
```

La referencia *people* gráficamente sería:



En Java, los *arrays* permiten ser inicializados de una manera especial mediante el uso de llaves {}. Por ejemplo:

```
int [] grades = { 5, 3, 6, 10, 7 };
```

Con la línea anterior hemos creado un *array* de 5 casillas de tipo int, donde el valor de la casilla 0 es 5, el de la casilla 1 es 3, el de la 2 es 6, el de la 3 es 10 y el de la casilla 4 es 7. La línea anterior es equivalente al siguiente código:

```
int [] grades = new int[5];  
  
grades[0] = 5;  
  
grades[1] = 3;  
  
grades[2] = 6;  
  
grades[3] = 10;  
  
grades[4] = 7;
```

Para declarar e inicializar a la vez un matriz podríamos hacer:

```
short[][] ages = {{1,2,3,4,5,6}, {6,5,4,3,2,1}, {1,3,5,7,9,11},  
{2,4,6,8,10,12}, {0,80,1,2,6}}; //El número de elementos de las filas y  
//columnas se deduce con el número de elementos pasados
```

Como hemos dicho varias veces, un *array* en Java es un objeto de una clase interna de Java. Algunos ejemplos de clase interna que Java utiliza para instanciar el objeto para los *arrays* de tipo primitivo sería (el símbolo [indica el número de dimensiones):

Tipo de array	Clase interna que se le asigna
int[]	[I
double[]	[D
double[][]	[[D
short[]	[S
byte[]	[B
boolean[]	[Z

3.8.2. Tamaño de un *array*

Como hemos visto, para inicializar un *array* se debe indicar su tamaño (*length*) para ubicarlo en memoria correctamente. Una vez creado en memoria, el tamaño no puede ser modificado, es decir, es fijo. Éste es uno de los mayores inconvenientes que tienen los *arrays*.

Como un *array* es un objeto de una clase interna de Java, éste pone a disposición de los programadores atributos públicos, como, por ejemplo, *length*. Este atributo de sólo lectura (es *final*) permite saber la longitud del *array*.

```
int [] grades = new int[5]; //Declara e inicializa grades con 5 casillas.  
  
int numGrades = grades.length; //numGrades es 5
```

El valor que puede tomar el índice que se le pasa a un *array* va de 0 hasta el tamaño de `length-1`. En caso de usar un valor incorrecto para acceder a un elemento del *array* –por ejemplo, un entero negativo o uno mayor a `length-1`–, Java lanzará una excepción (i.e. un error) llamado `ArrayIndexOutOfBoundsException`.

Veamos un ejemplo de matriz:

```
int [] grid = new int[10][8]; //Declara e inicializa grid de 10x8 int.

grid.length; //10

grid[0].length; //8

grid[1].length; //8

grid[9].length; //8
```

3.8.3. Copiar el contenido de un *array* a otro *array*

Existen diferentes maneras de copiar el contenido de un *array*. Todas ellas hacen una copia superficial (*shallow copy*) del contenido. ¿Qué quiere decir esto? Pues que se copia el contenido tal cual de los elementos del *array* original. Para los tipos primitivos, esto no es un problema, pero para los de tipo referencia sí, puesto que lo que se copia es la dirección de memoria de los objetos. ¿Qué consecuencias tiene esto? Imaginemos un programa en el que participan dos *arrays* de *Person*. En él, la casilla con índice 2 del *array* `people1` su atributo `name` tiene el valor "Elena". A continuación creamos el *array* `people2` haciendo una copia de `people1` con cualquiera de los métodos que explicaremos en este apartado. Si hacemos lo siguiente:

```
people2[2].name = "David"; //El valor antes de la asignación era "Elena"

people1[2].name; //será "David"; se ha modificado porque tanto people2[2] como
//people1[2] apuntan al mismo objeto, son la misma referencia.
```

En el ejemplo anterior no se ha hecho una copia profunda (*deep copy*) del contenido. Es decir, al hacer la copia no se ha creado un objeto nuevo de tipo *Person* para `people2[2]` con el mismo estado (i.e. mismos valores para todos los atributos) que el objeto de `people1[2]`, sino que directamente se ha copiado la referencia, i.e. la dirección de memoria al objeto.

En este caso, si queremos hacer una copia profunda, deberemos sobrescribir el método `clone` de la clase *Person* (ya lo veremos más adelante).

Método `clone`

Como un *array* en Java es un objeto de una clase interna y, como veremos, toda clase en Java hereda implícitamente de la clase raíz *Object*, entonces todo *array* proporciona el método `clone` que hereda de *Object*, que a su vez lo sobrescribe.

```
int[] grades1 = {1, 2, 3};

int[] grades2 = grades1.clone(); //ahora grades2 es {1, 2, 3}
```

Método arraycopy

La clase propia de Java llamada `System` proporciona un método público llamado `arraycopy` que permite indicar: el *array* de origen, el elemento a partir del cual se hace la copia, el *array* de destino, el índice del *array* de destino a partir del cual se copia y el número de elementos del *array* de origen que queremos copiar.

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8, 9};

int[] grades2 = new int[5];

System.arraycopy(grades1,0,grades2,0,5); //ahora grades2 es {1, 2, 3,4,5}
```

Método copyOf

Java proporciona el método estático `copyOf` dentro de la clase `Arrays`. Éste utiliza internamente el método anterior, `arraycopy`. Este método crea un *array* nuevo. Así pues, podríamos hacer:

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8};

int[] grades2 = Arrays.copyOf(grades1,8); //ahora grades2 es {1,2,3,4,5,6,7,8}

int[] grades3 = Arrays.copyOf(grades1,5); //ahora grades3 es {1,2,3,4,5}

int[] grades4 = Arrays.copyOf(grades1,6); //ahora grades4 es {1,2,3,4,5,0},
//añade un 0
```

Método copyOfRange

La misma clase `Arrays` tiene otro método que crea un nuevo *array* a partir de uno de origen con los elementos ubicados dentro de un rango o intervalo.

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8};

int[] grades2 = Arrays.copyOfRange(grades1,1,3); //ahora grades2 es {2,3} el
//último índice no se incluye, en este caso, el índice 3 que tiene valor 4.
```

3.8.4. Ordenar un array

Para los tipos primitivos, la clase `Arrays` ofrece el método `sort` el cuál ordena los elementos del *array* en orden ascendente, i.e. de menor a mayor.

```
int[] grades = {8, 7, 3, 4, 5, 6, 1, 2};

Arrays.sort(grades); //ahora grades es {1,2,3,4,5,6,7,8}
```

Si el tipo de elemento guardado en el *array* es de tipo referencia (i.e. un objeto de una clase), entonces debemos usar una sobrecarga del método `sort` que tiene como segundo parámetro un objeto de la clase `Comparator` que indica cómo se debe hacer la comparación entre elementos con el fin de ordenarlos correctamente.

```
People[] people = {new Person("David", 36), new Person("Elena",41), new
Person("Marina", 5), new Person("Pau",2)};

Arrays.sort(people, new Comparator<Person>() {
```

```

@Override
public int compare(Person first, Person second) {
    return first.getAge() - second.getAge(); //ordena de menor a mayor edad
}
}); //ahora people es { (Pau,2), (Marina, 5), (David, 36), (Elena, 41) }

```

<<Comentario al margen>>

Comparador es una interfaz que proporciona la API de Java que además es parametrizada. Veremos estos conceptos más adelante.

3.9. Métodos (funciones)

Cuando un programa se hace complejo, es muy útil dividir su código en pequeñas unidades lógicas, de manera que cada una de estas unidades sea capaz de hacer una tarea muy concreta del programa. A cada una de esas unidades contendrá parte del código del programa y las llamaremos <<métodos>> (en lenguajes imperativos son llamadas con el nombre de <<funciones>>). Esta organización del código de un programa en métodos facilita la legibilidad y el mantenimiento. Además, en muchas ocasiones hay trozos de código que deben ser llamados varias veces, por lo que es mejor no repetir estos fragmentos de código, sino ponerlos en métodos y llamar a dichos métodos cuando sea necesario.

3.9.1. Definición de un método

La sintaxis que define un método es:

```

accessModifier modifiers returnType methodName(Type param1, ..., Type paramN){
    //Code
}

```

Como podemos ver, el primer elemento a indicar es el modificador de acceso (veremos más adelante). Éste será `public`, `protected`, `private` o `ninguno` (lo que significará `package-private`).

El segundo conjunto de modificadores será: `final`, `abstract`, `static` o una combinación permitida de los tres (veremos su significado más adelante).

El tercer elemento es el tipo del valor o elemento que devuelve el método. Éste puede ser un tipo primitivo (p.ej. `int`, `char`, etc.), o un tipo referencia (p.ej. un `array`, un objeto de una clase, etc.) o el tipo `void` (que significa que no devuelve nada).

A continuación se escribe el nombre del método, el cual sigue el convenio de ser un verbo o sintagma verbal escrito siguiendo el estilo *lower camel case* (i.e. la primera palabra en minúscula y las siguientes palabras con la inicial en mayúscula).

Después viene el listado de parámetros que recibe (puede ser vacío). Para cada parámetro indicaremos su tipo y su nombre (en *lower camel case*).

Finalmente dentro de las llaves {} se escribe el código del método. Para devolver un valor (ya sea de tipo primitivo o referencia) se debe usar la palabra reservada `return`. Cuando un método encuentra un `return`, la ejecución del método finaliza devolviendo el control al punto donde el método ha sido llamado. En los métodos con tipo de devolución `void`, podemos no escribir `return` o escribir `return;` Lo más habitual es no escribir `return`.

Algunos ejemplos de método son:

```
public int addTwoInt(int number1, int number2){
    return number1 + number2;
}

private final double calculateArea(double width, double height){
    return width * height;
}

public void printText(String text){
    System.out.println(text);
}

protected int[] getArray(){
    return {3,5,6,7,10};
}

Person createDummyPerson(){ //su modificador de acceso es package-private
    new Person("Dummy", 10);
}
```

Estrictamente hablando, en Java sólo al conjunto formado por el nombre del método más los tipos de los parámetros en el orden en que aparecen es conocido como **firma o signatura del método**. Así pues, las firmas de los métodos anteriores serían:

```
addTwoInt(int, int)
calculateArea(double, double)
printText(String)
getArray()
createDummyPerson()
```

3.9.2. Sobrecarga de un método

El lenguaje Java soporta la sobrecarga (*overloading*) de métodos. Esto significa que podemos tener dentro de una misma clase dos métodos llamados exactamente igual siempre y cuando sus firmas no coincidan, i.e. se diferencien en el número y/o en el tipo y/o en el orden de los parámetros.

Es una práctica muy habitual (seguro que alguna vez lo has hecho) utilizar nombres diferentes para acciones similares sobre tipos de parámetros diferentes. Por ejemplo:

```
addTwoInt(int, int)
addTwoFloat(float, float)
addThreeInt(int, int, int)
addArrayInt(int[])
```

La práctica anterior es muy engorrosa. Por eso, lo habitual y más elegante en Java es hacer una sobrecarga. Así pues, el ejemplo anterior con sobrecarga sería:

```
add(int, int)
add(float, float)
add(int, int, int)
add(int[])
```

Es importante darse cuenta de que Java mira las firmas de los métodos, es decir, no considera el tipo de retorno (porque en Java estrictamente no es parte de la firma). Así pues, la siguiente sobrecarga sería incorrecta y el compilador daría un error:

```
void add(int, int)
int add(int, int)
```

Para el compilador de Java los dos métodos anteriores son el mismo método porque tienen la misma firma:

```
add(int, int, int)
```

En cambio, la siguiente sobrecarga es correcta porque el orden de los parámetros es diferente. Así pues, para Java son dos métodos distintos:

```
draw(String, int)
draw(int, String)
```

3.9.3. Usando un método

El proceso para usar un método es el siguiente:

- Desde un punto del programa se hace una llamada al método. Para ello se utiliza el nombre del método y la lista de valores que utilizarán los parámetros definidos en la firma. A estos valores que se hacen en la llamada se les llama **argumentos**.

```
int value = 8;
int sum = add(value, 3*2);
```

- El método es llamado y recibe los argumentos en sus correspondientes parámetros. Así pues, `number1` vale 8 y `number2` vale el resultado de `3*2`, es decir, 6.

- El método ejecuta el código que hay en su cuerpo.
- Devuelve un resultado o no (si es void).
- El valor es devuelto al punto en donde se ha llamado al método. Es decir, es como si en este momento se sustituyera el método por el valor devuelto por éste. En el ejemplo, la variable `sum` vale 14. A partir de ese punto, la ejecución del programa continua.

3.9.4. Paso por valor y por referencia

Cuando a un método se le pasa un argumento de tipo primitivo, entonces se hace una copia de su valor y es esa copia la que es pasada realmente al método. Así pues, el método que ha sido invocado utiliza el valor copiado, lo que implica que no puede modificar el valor original. Este proceso es conocido como **paso por valor**.

<pre>int value = 8; int sum = add(value, 3*2); System.out.print(value); //8</pre>	<pre>public int add(int number1, int number2){ number1++; return number1 + number2; }</pre>
---	---

En el ejemplo anterior el valor de la variable 8 es copiada en el parámetro `number1`. Cuando incrementamos una unidad el valor de `number1`, no estamos modificando el valor de la variable `value`. Así pues, cuando imprimimos por pantalla el valor de `value` en la tercera línea del código de la izquierda, su valor seguirá siendo 8, mientras que el valor de `sum` será 15 (i.e. 9 + 6). Si la variable `value` se hubiera llamado `number1` – es decir, igual que el primer parámetro del método `add` – tampoco se hubiera modificado el valor de la variable `number1` del código de la izquierda.

Así pues, con el paso por valor no podemos modificar, desde el método, el valor del elemento (i.e. variable) que ha sido pasado como argumento al método.

En cambio cuando el argumento pasado a un método es de tipo referencia (i.e. un *array* o un objeto de una clase), entonces el paso se hace por referencia. ¿Qué quiere decir esto? Pues que lo que se le pasa al parámetro del método es la referencia al objeto, es decir, la dirección de memoria del heap de ese objeto. Así pues, el parámetro del método es exactamente una referencia idéntica al objeto que se le ha pasado como argumento, por lo que está realmente accediendo al mismo objeto, lo que significa que puede modificarlo. Este proceso es conocido como **paso por referencia**.

<pre>int[] numbers = {1,2,8}; int sum = add(numbers); System.out.print(numbers[2]); //3</pre>	<pre>public int add(int[] numbers){ numbers[2] = 3; return numbers[0]+numbers[1]+numbers[2]; }</pre>
---	--

3.9.5. Número variable de parámetros

Desde JDK 1.5 es posible definir un método que no tenga un número fijo de parámetros. Es decir, unas veces recibe 2 argumentos, otras veces 3, etc. Hasta JDK 1.5 para simular esto lo que se hacía era pasar un *array* (fíjate que en los parámetros no se indica el tamaño). Sin

embargo, esto obliga a hacer ciertas tareas adicionales y el código no expresa lo que realmente se quiere indicar: la variabilidad de argumentos.

Así pues, JDK 1.5 introdujo la posibilidad de añadir argumentos variables (o también conocido como *varargs*) gracias a la sintaxis: Type ...

```
public void draw(String format, int... args){ }
public void draw(String format, Object... args){ }
```

Para utilizar parámetros variables existe una restricción: sólo el último parámetro puede ser definido como variable, es decir, con los tres puntos (...).

Los tres puntos indican que los argumentos correspondientes al último parámetro pueden ser pasados como un *array* o una secuencia de argumentos separados por comas. Sea cual sea el formato, el compilador se encargará de empaquetar todos los *varargs* en un *array*. Así pues, el último parámetro lo tendremos que tratar como un *array* para obtener los diferentes argumentos.

```
public void print(String ... texts){ //Versión varargs
    for(String text : texts){
        System.out.println(text);
    }
}
public void print(String text1, String text2){ //Sobrecarga con 2 textos
    System.out.println(text1 + " " +text2);
}
public static void main(String ... args){ // equivalente a String[] args
    print("Hola", "UOC", "¿qué tal?"); //Llama a la versión vargs
    print("Hola", "UOC"); //Llama a la sobrecarga con 2 textos
    print({"Hola", "UOC"}); //Llama a la versión varargs
}
```

Con el ejemplo anterior vemos que:

- De las sobrecargas de un método, la versión *varargs* será la última en ser llamada. Siempre se llama primero a la versión más específica/concreta que a la más genérica (i.e. *varargs*).
- Desde JDK 1.5 el parámetro del método *main* se puede escribir *String[] args* o *String ... args*.
- En el ejemplo anterior no podríamos definir una sobrecarga del método *print* con la siguiente firma porque se confundiría con la versión *varargs*:

```
public void print(String[] texts){ //Coincide con la versión varargs}
```

3.9.6. Cast implícito en los parámetros

Un método puede recibir como argumento un tipo primitivo numérico que sea de rango inferior al declarado para el correspondiente parámetro. Es decir, si el parámetro es declarado como float, entonces al hacer la llamada se le puede pasar un argumento de tipo short, int o float.

<pre>int number = 8; int sum = add(number, 3*2);</pre>	<pre>public int add(float number1, int number2){ return number1 + number2; }</pre>
--	--

Sin embargo el caso contrario no es aceptado por el compilador porque hay una pérdida de precisión. Si aun así queremos pasar un tipo numérico superior como argumento a un parámetro de tipo numérico inferior, deberemos hacer un *cast* explícito para que el compilador sepa que somos consciente de la pérdida de precisión que se va a producir.

<pre>float number = 8.0; int sum = add((int)number, 3*2);</pre>	<pre>public int add(int number1, int number2){ return number1 + number2; }</pre>
---	--

3.10. Ámbito de las variables

Cuando declaramos las variables, éstas existen o no –es decir, pueden ser usadas o no– en un punto de un programa dependiendo de dónde dichas variables hayan sido declaradas. Esa parte, área o región del programa en que la variable puede ser utilizada recibe el nombre en inglés de **scope** (en español, ámbito, alcance o, incluso, visibilidad de una variable).

El ámbito de una variable viene determinado por los bloques de instrucciones. Recordemos que un bloque de instrucciones está definido por una llave inicial { y una final }. Así pues, una variable estará disponible/visible/accesible en aquel bloque en el que se haya declarado. Es decir, el *scope* de la variable coincidirá con el bloque en el que se haya declarado.

<pre>public int operate(int a, int b, char op){ //a, b y op son accessibles en //cualquier parte de la función/método if(op == '+'){ int result = a + b; //result sólo es accesible en el bloque if. }else if(op == '-'){ return a - b; } return result; //error: result sólo es accesible dentro del if. Ni //siquiera es accesible dentro del else-if. }</pre>
--

Un práctica muy común es crear en los bloques de iteración de tipo **for**, variables locales cuyo *scope* sea el cuerpo del propio bloque **for**. Veamos un ejemplo:

<pre>public int add(int[] numbers){ //numbers es accesible en cualquier parte de //la función/método int result = 0; //result es accesible en cualquier parte del método add.</pre>
--

```

        for(int i = 0; i<numbers.length; i++){ //i sólo existe dentro del for.
            result += numbers[i];
        }
        System.out.println(i); //error: i no es accesible fuera del bloque for.
        return result;
    }
}

```

Sin embargo, el siguiente código sí sería correcto:

```

public int add(int[] numbers){ //numbers es accesible en cualquier parte de
//la función/método

    int result = 0, i = 0; //result e i son accesibles en cualquier parte
//del método add.

    for(i = 0; i<numbers.length; i++){ //i sólo existe dentro del for.
        result += numbers[i];
    }
    System.out.println(i); //correcto: i sería igual a numbers.length
    return result;
}

```

Ten en cuenta que el *scope* de un bloque afecta al *scope* de los bloques que tenga anidados. Así pues, si declaramos una variable en un bloque que tiene anidado un segundo bloque, este segundo bloque no podrá declarar una variable llamada igual, puesto que las variables del primer bloque existen dentro del segundo bloque.

```

public void printNumbers(int[] numbers){

    int i = 0;

    for(int i = 0; i<numbers.length; i++){ //error: el bloque for está
//anidado dentro del bloque del método print y éste ya declaró una variable i

        System.out.println(numbers[i]);
    }
}

```

Intenta adivinar qué muestra por pantalla el siguiente código:

```

public void printNumbers(int[] numbers){

    for(int i = 0; i<10; i++){

        System.out.println(numbers[i]);
    }

    int i = 50;

    System.out.println(i);
}

```

El código anterior imprimirá los números del 0 al 9 debido al bucle y posteriormente imprimirá el número 50. ¿Por qué? Pues porque se declara una variable `i` dentro del `for` que sólo existe dentro del `for` y a posteriori, una vez se ha terminado el `for` y, por lo tanto, se ha destruido la variable `i` del `for`, se declara en el bloque superior (i.e. en el método/función) una variable `i` que a partir de ese momento empieza a existir (no antes).

BORRADOR © D.G.S. - uoc

4. Orientación a objetos en Java

! Antes de seguir leyendo es importante que leas y entiendas, más o menos, los conceptos explicados en los módulos de teoría, como mínimo, el titulado <>Abstracción y encapsulación>>. En esta guía no vamos a explicar qué es una clase, ni un método, ni un modificador de acceso, etc. Lo que vamos a hacer es ver cómo estos conceptos se ponen en práctica mediante el lenguaje de programación Java. Obviamente, aparecerán algunos conceptos teóricos que detallaremos para este lenguaje. El hecho de ver cómo los conceptos teóricos se ponen en práctica te ayudará a acabar de entender algunas dudas que puedas tener sobre ellos.

4.1. Definiendo una clase

En este apartado vamos a ver cuál es la sintaxis mínima para declarar una clase así como los tres elementos esenciales que definen toda clase: los miembros de la clase (i.e. atributos y métodos) y constructores.

4.1.1. Estructura mínima

Para definir una clase en Java debemos tener presente que su esqueleto mínimo es el siguiente:

```
modificadorDeAcceso class NombreClase{
    //TODO
}
```

En Java el modificador de acceso que puede tomar una clase principal en Java es `public` o `package-private`. Por el momento, haremos que todas las clases sean `public`.

Por su parte, Java sigue la siguiente convención para nombrar las clases:

- El nombre es un sustantivo o un sintagma nominal en singular.
- Nombre escrito en *camel case*, i.e. todas las palabras que componen el nombre con la primera inicial en mayúscula, p.ej. `Ball`, `MemberVip`, `Person`, `Player`, `BankAccount`, etc.

Así pues, lo mínimo para definir la clase `Person` en Java sería:

```
public class Person{
    //TODO
}
```

4.1.2. Miembros de la clase

Como ya sabes, las clases tienen miembros. Hay dos tipos de miembros de la clase: los atributos y los métodos. Así pues, hay que definirlos. Lo más habitual es definir primero los atributos y después los métodos (y en medio los constructores). Es decir:

```
public class Person{
    //Atributos primero
    //Constructores después
    //y Métodos al final
}
```

Atributos

Los atributos se definen igual que las variables (incluso respetan las convenciones usadas en las variables para los nombres), pero indicando además su modificador de acceso. De hecho, puedes pensar en ellos como variables globales dentro la clase. Es decir, todos los atributos de una clase, independientemente del modificador de acceso que se les asigne, son visibles en cualquier parte de la clase en la que están definidos. La única diferencia con las variables es, como hemos dicho, que a los atributos hay que asignarles un modificador de acceso y a las variables no. Veamos algunos ejemplos:

<<comentario al margen>>

Nomenclatura para los atributos

Además del término *atributo* (*attribute*), también se emplea el término *campo* (*field*).

```
public class Person{

    private String name, surname;

    private int age = 0; //podría haber sido byte para ocupar menos espacio

    private int height; //entendemos que se indica en centímetros

    private float weight;

}
```

Si te fijas, antes de la declaración del atributo le indicamos el modificador de acceso que queremos que se aplique (observa las palabras en rojo). Lo más habitual es declarar los atributos como `private`. Si sabemos que la clase va a participar en una relación de herencia, entonces también puede ser una buena opción `protected` (dependerá de nuestro diseño).

¿Qué ocurre si no se indica un modificador de acceso? Entonces se entiende que el atributo tiene un modificador de acceso `default` o también llamado como `package-private` (un modificador de acceso especial de Java).

<<comentario al margen>>

Modificadores de acceso

Más adelante prestaremos especial atención a los diferentes modificadores de acceso que proporciona Java. Por ahora, sólo considera `public` y `private`.

Observa la declaración del atributo `age` de la clase `Person` en el ejemplo anterior. ¿Ves alguna diferencia con el resto de atributos? Efectivamente, se le ha asignado un valor, concretamente, cero. Esta inicialización se podría haber hecho en uno de los constructores de `Person` (lo veremos más adelante), pero al ser tan simple, lo hemos hecho en la propia declaración. Si la inicialización del atributo `age` hubiera necesitado alguna lógica (p.ej. usar un bucle), entonces no podríamos haberlo hecho en la declaración. La inicialización en la declaración sólo es posible si es simple, es decir, ocupa una línea.

Igualmente, **si un atributo no es inicializado en ninguna parte de la clase** (i.e. declaración, constructor, bloque de inicialización, etc.), **entonces el compilador le asignará el valor por defecto del tipo que se le ha asignado**, p.ej. 0 para atributos de tipo int, false para boolean y null para tipos referencia.

Finalmente indicar que los atributos de una clase, a diferencia de las variables, se crean dentro del objeto. Por consiguiente, a nivel de ubicación en memoria, los atributos están en el *heap* (como los objetos), no en el *stack*.

Métodos

Todo lo que comentamos en el apartado 2.9 sirve ahora que estamos usando Java orientado a objetos. No obstante, vamos a explicar algunas cuestiones adicionales que aparecen al usar clases.

La primera de ellas es que el modificador static ya no lo vamos a usar si no es estrictamente necesario. Hasta ahora usábamos este modificador porque empleábamos Java en un paradigma de programación estructurada y, sin static, no podíamos llamar a los métodos desde el main.

En segundo lugar, vamos a ver una situación que sólo puede darse cuando usamos atributos dentro de los métodos. Mira el siguiente método *setter*:

```
public class Person{

    private String name, surname;
    private int age;
    private int height;
    private float weight;

    public void setAge(int ageNew){

        if(ageNew<0){

            System.out.print("Error: new age must be a positive number");
            age = 0;
        }else{
            age = ageNew;
        }
    }
}
```

El método setAge anterior lo primero que hace es comprobar si el parámetro ageNew es negativo. Si lo es, entonces escribe un mensaje de error por pantalla y asigna el valor 0 al atributo age. En caso contrario, asigna al atributo age el valor del parámetro ageNew. Sin

embargo, lo ideal hubiera sido llamar al parámetro igual que al atributo, para saber que están relacionados. Es decir, el código "ideal" hubiera sido:

```
public void setAge(int age){
    if(age<0){
        System.out.print("Error: new age must be a positive number");
        age = 0;
    }else{
        age = age;
    }
}
```

Entonces, ¿cómo sabe el compilador (y nosotros) en cada sentencia escrita en rojo si age es el atributo o el parámetro? El caso más evidente de confusión es la asignación del else: age = age. ¿El age del lado izquierdo de la asignación es el atributo o el parámetro? ¿Y el age de la derecha? Así pues, para evitar problemas, muchos programadores le ponen al parámetro del método el mismo nombre del atributo de la clase más una coletilla. De ahí que la declaración inicial del método setAge fuera:

```
public void setAge(int ageNew){
    //Código
}
```

Es obvio que con esta solución de añadir una coletilla (en el ejemplo "New") no hay ninguna duda para el compilador ni para nosotros de qué es qué. A pesar de que podemos usar una solución como la anterior, lo más habitual y mejor (como hemos comentado) es utilizar como nombres de los parámetros de los métodos los mismos nombres que los de los atributos de la clase, en este caso, age. Para resolver el conflicto de nombres, Java nos proporciona una palabra reservada llamada this. Así pues, dada una versión simplificada del método setAge como la siguiente:

```
public void setAge(int ageNew){
    age = ageNew;
}
```

Su equivalente con this sería:

```
public void setAge(int age){
    this.age = age;
}
```

El `this` hace de coletilla e indica que aquello que lleva `this` pertenece al objeto/clase y no es ni un parámetro ni una variable declarados dentro del método. Concretamente `this` hace referencia al objeto (o instancia) que es del tipo de la clase. Por lo tanto, gracias a `this`, estamos diciendo que al atributo `age` del objeto (o si te es más fácil de entender, de la clase) hay que asignarle el valor del parámetro `age` (llamado `ageNew` en la versión anterior del código).

4.1.3. Constructor

En Java podemos definir tantos constructores como deseemos. Incluso podemos no declarar ningún constructor. En este último caso, el compilador de Java lo detectará y creará uno por defecto de manera transparente a nosotros que no hará nada, sólo permitir que se puedan instanciar objetos de esa clase (¡casi nada!).

Así pues con el código anterior de la clase `Person` ya tenemos un constructor por defecto. No estará creado por nosotros, sino que lo hará el compilador. Es decir, el código anterior es equivalente a haber escrito lo siguiente:

```
public class Person{

    private String name, surname;

    private int age = 0; /

    private int height;

    private float weight;

    //Constructor por defecto

    public Person(){
    }

}
```

Como te habrás dado cuenta, el constructor, a diferencia de los métodos, no define un tipo de retorno, ni siquiera `void`. Gracias a esta diferencia, el compilador puede saber que se trata de un constructor y no de un método. Además, obligatoriamente, el nombre del constructor debe coincidir con el nombre de la clase.

Constructor por defecto

Vamos a ver cómo podríamos llenar un constructor por defecto.

```
public class Person{

    private String name, surname;

    private int age;

    private int height;

    private float weight;
```

```
//Constructor por defecto

public Person(){

    name = "David";

    surname = "García";

    age = 36;

    height = 172;

    weight = 66;

}

}
```

El constructor por defecto anterior es correcto. Incluso, no haría falta por qué inicializar todos los atributos (fíjate que hemos eliminado la inicialización en la declaración de age... no tiene sentido duplicar). Ahora bien, si te acuerdas de lo que comentamos en el apartado 2.6 del módulo <>Abstracción y encapsulación>>, es interesante y una buena práctica utilizar los métodos *setter* y *getter* dentro de la propia clase con tal de conseguir la máxima consistencia, estabilidad y robustez de los atributos. Así pues, una mejor codificación del constructor anterior sería:

```
public Person(){

    setName("David");

    setSurname("García");

    setAge(36);

    setHeight(172);

    setWeight(66);

}
```

El hecho de usar los *setters*, nos ayuda a crear un punto común de comprobaciones y asignaciones. Si te acuerdas, dentro del método *setAge* se comprueba que la edad pasada como argumento no sea inferior a 0. Si lo es, imprimimos un mensaje de error y asignamos el valor 0 al atributo *age*. En caso contrario, asignamos el valor que nos facilitan.

Constructor con argumentos

Como hemos dicho, nuestras clases pueden tener tantos constructores como necesitemos. Para ello lo que tendremos que hacer es sobrecargar el constructor. Veamos dos ejemplos de constructor con argumentos.

```
public class Person{  
  
    private String name, surname;  
  
    private int age;  
  
    private int height;  
  
    private float weight;  
  
  
    //Constructor por defecto  
  
    public Person(){  
  
        setName("David");  
  
        setSurname("García");  
  
        setAge(36);  
  
        setHeight(172);  
  
        setWeight(66);  
  
    }  
  
  
    //Constructor con argumentos 1  
  
    public Person(String name, String surname, int age){  
  
        setName(name);  
  
        setSurname(surname);  
  
        setAge(age);  
  
        setHeight(172);  
  
        setWeight(66);  
  
    }  
  
  
    //Constructor con argumentos 2  
  
    public Person(String name, String surname, int age, int height, float weight){  
  
        setName(name);  
  
        setSurname(surname);  
  
        setAge(age);  
  
        setHeight(height);  
  
        setWeight(weight);  
  
    }  
}
```

Si te fijas, el primer constructor con argumentos sólo tiene 3 parámetros, mientras que el segundo tiene 5 (tantos como atributos tiene la clase). ¿Te imaginas que hubiera pasado si no llamáramos a los métodos *setters* en los tres constructores? Pues que en los tres repetiríamos el mismo código de comprobación que ahora tenemos centralizado en el método `setAge`. Una vez más, se ve la importancia de llamar a los *getters* y *setters* también desde dentro de la propia clase. Aun así, puedes ver que los tres constructores tienen un código común, i.e. llamar a los *setters* de los atributos. ¿Por qué repetir líneas de código idénticas? Java también se dio cuenta de que esto era muy común, por lo que creó un par de maneras de reducir la repetición de código en los constructores.

Método especial `this`

Java pone a disposición de los programadores un método especial llamado `this`. Este método tiene la misma firma que cualquiera de los constructores que hemos declarado. Éste se utiliza en los constructores para llamar a otros constructores. Tiene la restricción que, si se quiere utilizar, la llamada al método `this` debe ser la primera instrucción que escribamos en el constructor donde lo utilizaremos. Veamos el código anterior simplificado gracias al uso del método especial `this`.

```
public class Person{

    private String name, surname;
    private int age;
    private int height;
    private float weight;

    //Constructor por defecto
    public Person(){

        this("David","García",36,172,66);
        //Aquí podríamos añadir las instrucciones que quisiéramos
    }

    //Constructor con argumentos 1
    public Person(String name, String surname, int age){

        this(name,surname,age,172,66);
        //Aquí podríamos añadir las instrucciones que quisiéramos
    }

    //Constructor con argumentos 2
    public Person(String name, String surname, int age, int height, float weight){

        setName(name);
        setSurname(surname);
        setAge(age);
        setHeight(height);
        setWeight(weight);
    }
}
```

Si te fijas, el cambio se ha producido en los dos primeros constructores (observa las dos instrucciones en rojo). Lo que hemos hecho es llamar al último constructor (el más genérico) desde los otros dos constructores. Observa que la firma del método `this` coincide con la del último constructor con argumentos.

Bloque de inicialización de instancia

Para los casos en los que la inicialización de los atributos para los objetos (instancias) creados en una clase tienen valores conocidos a priori, entonces Java proporciona lo que se conoce como bloque de inicialización (*initializer block*).

Imagina por ejemplo que la edad siempre sea 36 para todos los objetos creados para la clase Person. Entonces podríamos hacer:

```
public class Person{

    private String name, surname;
    private int age;
    private int height;
    private float weight;

    //Esto es un bloque de inicialización. Tan simple como abrir y cerrar llaves {}
    {
        setAge(36); //Podríamos haber hecho age = 36; (mejor usar el setter)
    }

    //Constructor por defecto
    public Person(){

        this("David","García",172,66);
        //Aquí podríamos añadir las instrucciones que quisiéramos
    }

    //Constructor con argumentos 1
    public Person(String name, String surname){

        this(name,surname,172,66);
        //Aquí podríamos añadir las instrucciones que quisiéramos
    }

    //Constructor con argumentos 2
    public Person(String name, String surname, int height, float weight){

        setName(name);

        setSurname(surname);

        setHeight(height);

        setWeight(weight);
    }
}
```

Cuando se llame a cualquiera de los tres constructores anteriores, lo primero que ejecutará será el bloque de inicialización. Es decir, primero se llamará al método `setAge` con el argumento 36 y a continuación el resto de código del constructor llamado. Puedes observar que el código ha cambiado un poco, las instrucciones donde se han introducido cambios están resaltadas en rojo. Así pues, debe quedar claro que **el bloque de inicialización de instancia de una clase se ejecuta antes que los constructores de dicha clase**. Una clase puede tener tantos bloques de inicialización de instancia como se requieran. En este caso, el orden de ejecución será *top-down*, es decir, aquél que aparezca primero en el código será el primero en ejecutarse y así sucesivamente.

En resumen:

- Cada bloque de inicialización de instancia se ejecuta cada vez que se instancia un objeto nuevo y se ejecuta para ese objeto en creación.
- Si hay más de un bloque de inicialización de instancia, entonces el orden de ejecución empieza por el que primero aparece en el código de arriba abajo.
- Si la clase en la que hay un bloque de inicialización hereda de otra clase, entonces primero se llama a los bloques de inicialización de instancia de la clase padre, luego al constructor de la clase padre, después los bloques de inicialización de la clase para la que estamos creando el objeto y finalmente el código del constructor que hemos usado para instanciar el objeto.
- Si un atributo se ha inicializado en la declaración del propio atributo, entonces esta inicialización se hace antes de la llamada al bloque de inicialización de instancia.

<<comentario al margen>>

Herencia y clase padre

El concepto de *herencia* y *clase padre* lo veremos más adelante tanto en los apuntes teóricos como en esta guía.

4.2. Objetos

4.2.1. Instanciar (creando objetos)

Como ya sabes, un objeto es un ejemplar de una clase. Así pues, a la hora de crear un objeto, debemos seguir los siguientes pasos:

1) Declarar una variable o atributo de tipo de la clase (en términos generales es un tipo referencia):

```
Person david;
```

En este momento la variable/atributo `david` no apunta a ningún objeto, su referencia es `null`.

2) Instanciar el objeto. Es decir, crear un objeto a partir de una clase.

```
david = new Person();
```

Como has visto en el apartado 3.1.4 y en el código justo anterior, para instanciar/crear un objeto nuevo debemos escribir la palabra reservada `new` seguida de una llamada a uno de los constructores definidos en la clase. En este caso hemos utilizado el constructor por defecto, pero también podríamos haber usado cualquiera de los dos constructores parametrizados. Por ejemplo:

```
david = new Person("David", "García");
```

En este momento ya tenemos un objeto de tipo `Person` ubicado en memoria (*heap*) y con una variable/atributo de tipo referencia en apuntando a este objeto.

Los pasos 1 y 2 se pueden agrupar en un único paso:

```
Person david = new Person();
```

Como el segundo paso se le llama <<instanciar>>, ahora entenderás por qué a los objetos también se les llama instancias.

4.2.2. Usando un objeto (mensajes)

Una vez creado un objeto y referenciada por una variable/atributo, lo más lógico es hacer algo con ella, normalmente acceder a un atributo o a un método. Para ello usaremos mensajes. La forma de un mensaje es:

```
referencia.miembro
```

donde `referencia` puede ser una variable o un atributo y `miembro` puede ser tanto un atributo como un método de la clase a la que pertenece el objeto.

```
Person elena = new Person("Elena", "Lázaro");
elena.setAge(41); //llamamos al método setAge del objeto al que apunta elena.
Person david = new Person();
david.setAge(elena.getAge()); //llamamos al método setAge del objeto al que
//apunta david y le pasamos como argumento el valor devuelto por el método
//setAge del objeto al que apunta elena.
System.out.println(david.setAge()); //Imprime 41
```

4.3. Modificadores de acceso

Te habrás dado cuenta de que en los ejemplos anteriores los atributos, métodos y constructores tenían la palabra `public` o `private` al inicio de su declaración. Como bien sabes, tanto `public` como `private` son dos modificadores de acceso, cuya funcionalidad es determinar la accesibilidad/visibilidad de un elemento. Cuando hablamos del uso de modificadores de acceso, nos estamos refiriendo a que si tenemos un objeto de una clase A dentro de otra clase B, según cómo estén declarados los miembros de la clase A, éstos podrán ser accedidos directamente mediante un mensaje o no desde B usando el objeto de la clase A.

```
public class B{
    A objectA;
    public B(){
        objectA.member; //esto será correcto o no dependiendo del
        //modificador de acceso con el que sea declarado "member" en la clase A.
    }
}
```

Todos los miembros de una clase, independientemente de su modificador de acceso, son accesibles directamente desde dentro de la propia clase.

En Java existen 4 modificadores de acceso: public, private, protected y package-private. Cada uno de estos cuatro modificadores permite que un elemento sea más o menos accesible desde fuera de la propia clase, siendo public y private los extremos.

Dependiendo del elemento, a éste se le podrá asignar unos modificadores de acceso u otros. Principalmente:

Tipo de nivel	Tipos de elemento	Modificador de acceso permitido
Alto nivel (o elemento contenedor)	Clases, interfaces y enumeraciones (enum)	public package-private
Miembros	Atributos, métodos, constructores, clases anidadas, enumeraciones (enum)	public protected private package-private

Modificador package-private

Para asignar este modificador, no hay que escribir ningún modificador al declarar el elemento. Es decir, cuando no se escribe ni public, ni protected ni private, Java entiende que el modificador que se le quiere asignar al elemento es package-private. Así pues, nunca hay que escribir la palabra compuesta package-private. Por este motivo, a este modificador también se le conoce como *no modifier* o *default*.

Como podemos ver, al declarar, por ejemplo, una clase, sólo podemos asignarle, o bien el modificador public o el modificador package-private. Sin embargo, si declaramos una clase dentro de otra clase, es decir, declaramos una clase anidada (*nested class*), entonces dicha clase se comporta como un miembro de la clase contenedora y, entonces, podemos asignarle cualquiera de los 4 modificadores de acceso que proporciona Java.

Llegados a este punto, cabe saber las diferencias entre los cuatro modificadores. Es decir, qué implicaciones tienen cada uno de ellos. Vamos a comentar los 4 modificadores teniendo en cuenta que dentro de una clase B tenemos un objeto de la clase A con un miembro al que queremos acceder:

- public: indica que el elemento puede ser accedido desde cualquier parte del programa. Así pues, si dentro de la clase B queremos acceder a un miembro público de un objeto de la clase A, dicho miembro podrá ser accedido desde B de la siguiente manera:

```
objectA.publicMember; //OK
```

- private: indica que el miembro sólo puede ser accedido desde dentro de la propia clase que lo declara. Así pues, si intentamos acceder a un miembro privado de la clase A usando un objeto de la clase A perteneciente a la clase B, entonces obtendremos un error de compilación:

```
objectA.privateMember; //KO
```

Acabamos de ver los dos modificadores extremos. Es decir, `public` es el modificador más permisivo y `private` el más restrictivo. Nos faltan dos modificadores de acceso que son muy similares entre sí y que sólo difieren en el caso de que exista una relación de herencia entre las clases implicadas (i.e. superclase-subclase).

- `package-private`: indica que el miembro declarado como `package-private` (o `default`) sólo puede ser accedido desde elementos (p.ej. una clase) que estén dentro del mismo *package* que el elemento en la que está el miembro `package-private`. Dicho de otro modo, lo que nos está diciendo el modificador `package-private` es que los elementos declarados como tales (i.e. sin modificador) son accesibles/visibles directamente para todos los elementos declarados dentro del mismo *package*. Así pues, sólo si la clase B y la clase A pertenecen al mismo *package*, entonces la siguiente sentencia en la clase B será correcta:

```
objectA.defaultMember;
```

- `protected`: además de lo que permite el modificador `package-private`, también permite que los miembros declarados como `protected` sean visibles/accesibles desde cualquier subclase de la clase en la que se ha declarado el miembro `protected`. Eso sí, desde una subclase (independientemente de si están en el mismo *package* que la superclase) sólo es accesible a través de una referencia/objeto que sea como mínimo del mismo tipo que la subclase. Así pues, `protected` es un poco menos restrictivo que `package-private`.

En resumen:

Modificador	Desde el propio contenedor (i.e. clase, interfaz o enum)	Desde el mismo package	Subclase en otro package	El resto de elementos
<code>Public</code>	✓	✓	✓	✓
<code>Protected</code>	✓	✓	✓ (sólo usando un objeto de la propia subclase o subclase de ésta)	✗
<code>package-private</code> (sin modificador)	✓	✓	✗	✗
<code>Private</code>	✓	✗	✗	✗

Como ya hemos comentado y como se puede observar en la columna “Desde el propio contenedor” de la tabla anterior, **cualquier elemento es accesible desde dentro del elemento que lo declara**. Por ejemplo, un atributo declarado en la clase A, independientemente del modificador de acceso asignado, siempre es accesible desde cualquier punto de la clase A (p.ej. cualquier método declarado en la clase A). Así pues, **cualquier elemento contenedor siempre tiene acceso a todos sus miembros**. Esto es algo que debe quedar muy claro.

<<comentario al margen>>

Package y subclase

Por ahora no te preocupes por los conceptos *package* y *subclase* (así como *superclase* y *herencia*). El primero lo veremos en esta guía y el segundo lo explicaremos tanto en los apuntes teóricos como en esta guía.

En cambio, tal y como indica la tabla anterior, una clase A creada dentro del *package* P1 no podrá acceder a un elemento *private* declarado en una clase B del mismo *package* P1. Sin embargo, si el mismo elemento hubiera sido declarado como *public*, *protected* o *package-private*, entonces la clase A sí que podría haber accedido a él y utilizarlo.

Si seguimos analizando la tabla anterior, vemos que aquellos elementos declarados *public* o *protected* pueden ser accedidos desde cualquiera de sus subclases, estén en el mismo *package* o no. Aquí hay que hacer una apreciación importante para el caso *protected* (de ahí el asterisco en la tabla anterior). Cuando la superclase (p.ej. A) y la subclase (p.ej. B) pertenecen a distintos *packages*, entonces no podemos acceder a un miembro *protected* de la superclase dentro de la subclase con un objeto de la superclase: *objectA.protectedMember*; En cambio, sí podremos acceder usando un objeto de la propia subclase B o de otras subclases de B. De ahí que hayamos dicho que un miembro *protected* es accesible desde una subclase de la clase que lo declara a través de una referencia que sea, al menos, del mismo tipo que la subclase. Esto es así para asegurar que los miembros de instancia *protected* son sólo accedidos por parte de la jerarquía de clases al que pertenecen (i.e. subclase y sus subclases), excluyendo clases hermanas (i.e. otras clases que heredan de la clase que heredan de la clase que declara el miembro *protected*).

Finalmente, un elemento declarado como *public* es visible para cualquier otro elemento del programa. Es decir, si tenemos un método declarado como *public* en una clase A, éste podrá ser accedido desde un objeto A declarado en otra clase B, independientemente de si A y B están en el mismo *package* o no, y de si entre A y B existe una relación de herencia (i.e. superclase-subclase).

Veamos algunos ejemplos para afianzar estos conceptos. Usaremos la siguiente clase A.

```
package mypackage;
public class A{
    public int age;
    private String name;
    protected double weight;
    double height;
}
```

Imaginemos que tenemos la siguiente clase B con un objeto de la clase A anterior:

```
package mypackage;
public class B{
    A objectA;
    public B(){
        objectA = new A();
        objectA.age = 5; //correcto porque "age" es public
        objectA.name = "David"; //incorrecto porque "name" es private
        objectA.weight = 63.2; //correcto porque "weight" es protected y
//las clases A y B están en el mismo package
        objectA.height = 1.73; //correcto porque "height" es package-
//private y las clases A y B están en el mismo package
    }
}
```

Imaginemos que ahora tenemos la siguiente clase C con un objeto de la clase A anterior:

```
package anotherpackage;

import mypackage.A; //por ahora no hagas caso a esta sentencia

public class C{
    A objectA;

    public C(){
        objectA = new A();
        objectA.age = 5; //correcto porque "age" es public
        objectA.name = "David"; //incorrecto porque "name" es private
        objectA.weight = 63.2; //incorrecto porque "weight" es protected
//y las clases A y C están no en el mismo package
        objectA.height = 1.73; //incorrecto porque "height" es package-
//private y las clases A y C no están en el mismo package
    }
}
```

Ahora imaginemos que la clase D es una subclase (concepto que veremos más adelante) de la clase A y tiene un objeto de la clase A. Ambas clases pertenecen al mismo package.

```
package mypackage;

public class D extends A{ //por ahora no hagas caso a "extends A"
    A objectA;

    public D(){
        super(); //por ahora no hagas caso a este método especial.
        objectA = new A();
        objectA.age = 5; //correcto porque "age" es public
        objectA.name = "David"; //incorrecto porque "name" es private
        objectA.weight = 63.2; //correcto porque "weight" es protected y
//las clases A y D no están en el mismo package
        objectA.height = 1.73; //correcto porque "height" es package-
//private y las clases A y D están en el mismo package
    }
}
```

Finalmente veamos una clase E que es sublcase de A, pero ambas clases pertenecen a packages diferentes.

```
package anotherpackage;

import mypackage.A; //por ahora no hagas caso a esta sentencia

public class E extends A{ //por ahora no hagas caso a "extends A"
    A objectA;

    public E(){
        super(); //por ahora no hagas caso a este método especial.
        objectA = new A();
        objectA.age = 5; //correcto porque "age" es public
        objectA.name = "David"; //incorrecto porque "name" es private
        objectA.weight = 63.2; //incorrecto porque "weight" es protected
// y A y E no están en el mismo package; y usamos un objeto de la superclase A
        objectA.height = 1.73; //incorrecto porque "height" es package-
//private y las clases A y E no están en el mismo package
    }
}
```

Veamos una variante de la clase E anterior:

```

package anotherpackage;

import mypackage.A; //por ahora no hagas caso a esta sentencia

public class E extends A{ //por ahora no hagas caso a "extends A"
    A objectA;
    E objectE;

    public E(){
        super(); //por ahora no hagas caso a este método especial.

        objectA = new A();
        objectA.age = 5; //correcto porque "age" es public
        objectA.name = "David"; //incorrecto porque "name" es private
        objectA.weight = 63.2; //incorrecto porque "weight" es protected
// y A y E no están en el mismo package; y usamos un objeto de la superclase A
        objectA.height = 1.73; //incorrecto porque "height" es package-
//private y las clases A y E no están en el mismo package

        objectE = new E();
        objectE.age = 5; //correcto porque "age" es public
        objectE.name = "David"; //incorrecto porque "name" es private
        objectE.weight = 63.2; //correcto porque "weight" es protected y
//accedemos con un objeto E donde E es subclase de la clase A. Podríamos haber
//acedido con un objeto cuyo tipo fuera una subclase de E.

        objectE.height = 1.73; //incorrecto porque "height" es package-
//private y las clases A y E no están en el mismo package
    }
}

```

4.4. Destructor de una clase

En Java no existe un destructor como tal. Sin embargo, proporciona un método llamado `finalize` que hace funciones típicas de destructor. Este método `finalize` lo tienen todas las clases, puesto que, como comentaremos más adelante, todas las clases en Java heredan implícitamente de la clase raíz `Object`. Así pues, `finalize` es un método de la clase `Object` que cualquier clase hereda, quiera o no.

El método `finalize` que proporciona la clase `Object` no hace nada por lo que puede ser sobrescrito (veremos qué significa esto). Básicamente quédate con la idea de que si quieres que el método `finalize` haga algo, debes escribirlo explícitamente y añadir un código en su cuerpo. Si no tienes que hacer nada especial, entonces no es obligatorio sobrescribirlo (i.e. añadir código). Lo más habitual es no escribir el método `finalize`.

La manera de explicitar (sobrescribir) el método `finalize` sería añadir el siguiente código en tu clase, como si fuera un método más (normalmente se pone el último para encontrarlo fácilmente):

```

public void finalize(){
    //Tu código aquí
}

```

Algunos motivos por los que puedes utilizar el método `finalize` son: liberar recursos compartidos, cerrar conexiones (p.ej. bases de datos, sockets, etc.), etc. Sin embargo ten en cuenta que este método es llamado automáticamente dentro del proceso conocido como *garbage collection* cuando la JVM lo estima oportuno. Es decir, cuándo el método `finalize` es realmente llamado es algo incierto. Así pues, no debes depender de este método para hacer algunas gestiones críticas. Lo que sí debes tener presente que el método `finalize` no se llamará hasta que un objeto no tenga ninguna variable/atributo de tipo referencia apuntándole. **Si un objeto no es apuntado por ninguna referencia, entonces se llamará a su método `finalize`**, ¿cuándo? Es una incógnita. La JVM tiene un *garbage collector* que periódicamente librea memoria usada por objetos que nunca más están referenciados. El *garbage collector* hace este trabajo de manera automática cuando cree que es el momento adecuado.

```
Person p1 = new Person(); //Creamos un objeto Person y lo asignamos a la
//variable p1

Person p2 = p1; //p2 apunta al mismo objeto que p1

p1 = null; //p1 ya no apunta al objeto Person creado en la primera
//instrucción. Sin embargo, no se llamará al método finalize del objeto,
//puesto que está siendo apuntado por la variable p2.

p2 = null;
```

Después de la última instrucción del código anterior (`p2 = null;`), tanto `p1` como `p2` no apuntan a ningún objeto. El objeto `Person` creado en la primera instrucción no es referenciada/apuntada por nadie, así que se llamará automáticamente (i.e. como programadores no tenemos que hacer nada) a su método `finalize` y se liberará el espacio que ocupa en memoria (*heap*). ¿Cuándo se hará esta llamada? Cuando el *garbage collector* lo crea oportuno.

Como puedes ver, la gestión de la memoria en Java es mucho más sencilla que en otros lenguajes como puede ser C++. Obviamente esto también implica menos libertad a la hora de “jugar” con la memoria del ordenador y poder optimizar procesos.

4.5. Packages

4.5.1. ¿Qué es un package y cuál es su utilidad?

En Java existe un tipo de contenedor abstracto llamado *package* que permite organizar elementos de alto nivel (i.e. clases, interfaces y enumeraciones) que están relacionados entre sí. La manera más sencilla de entender qué es un *package* sería haciendo una analogía con un sistema de directorios. Por ejemplo, imagina que tenemos un proyecto con la siguiente estructura de directorios:

```
_project
|__database
|__images
|   |__hd
|   |   |__user.png
|   |__sd
|   |   |__user.png
|__views
```

Si nos fijamos, tanto en las subcarpetas hd como sd tenemos dos imágenes llamadas exactamente igual: user.png. ¿Cómo podemos distinguirlas únicamente? A través de su ruta (*path*): project/images/hd/user.png es diferente a project/images/sd/user.png.

Pues bien, los *packages* en Java hacen exactamente esto: organizar nuestro código para que sea más fácil encontrar lo que queremos de manera única y, además, añadir información lógica a nuestro *software*.

Así pues, lo primero que debemos saber es que en un programa podemos tener muchas clases escritas y algunas de ellas pueden incluso llamarse igual. Por ejemplo, la API (*Application Programming Interface*) de Java, la cual no deja de ser una librería con centenares de clases ya creadas que nos proporciona el propio lenguaje para programar nuestro propio software de manera más sencilla, está organizada en *packages*: <https://docs.oracle.com/en/java/javase/13/docs/api/allpackages-index.html>.

En la API de Java hay elementos llamados exactamente igual pero que pertenecen a *packages* distintos. Por ejemplo, la List existe tanto en java.util como en java.awt. En este caso concreto, además, se da la circunstancia que List es una interfaz en java.util, mientras que en java.awt es una clase, ¡son dos cosas diferentes que se llaman igual!! En este caso, el *package* java.awt incluye un conjunto de clases para crear interfaces de usuario, mientras que java.util contiene un conjunto de elementos relacionados con colecciones (i.e. estructuras de datos): listas, colas, pilas, tablas de hash, etc.

Para decirle al compilador de Java qué elemento List usar en nuestro programa, debemos indicarlo haciendo un import. Por ejemplo:

```
import java.awt.List; //Indicamos que queremos la clase List de java.awt
import java.awt.Frame; //Indicamos que queremos la clase Frame de java.awt
//(existe otro package para interfaces gráficas llamado swing que tiene Frame
public class MyAwtList{
    public static void main(String [] args){
        Frame window = new Frame();
        List family = new List(4);
        family.setBounds(100,100,75,50);
        family.add("David");
        family.add("Elena");
        family.add("Marina");
        family.add("Pau");
        window.add(family);
        window.setSize(300,300);
        window.setLayout(null);
        window.setVisible(true);
    }
}
```

Así pues un *package* sirve para:

- Empaquetar/agrupar elementos (clases, interfaces y enumeraciones) que están relacionados.
- Crear *namespaces* y evitar conflictos cuando dos o más elementos se llaman exactamente igual. Al estar los elementos homónimos en *packages* diferentes, éstos son identificados únicamente. Por lo tanto, el nombre completo del elemento está compuesto por el nombre package + el nombre del elemento en sí. Esto es lo que se conoce como *fully qualified name*, el cual permite desambiguar el elemento al que nos referimos. Así pues, los *fully qualified names* de los List anteriores son: `java.awt.List` y `java.util.List`.
- Permitir acceso especial entre sí a los elementos que pertenecen a un mismo *package*, gracias al modificador `package-private`.

4.5.2. Creando un *package*

Crear un *package* es muy sencillo. Sólo debemos añadir la siguiente sentencia como **primera línea de código de cada uno de los elementos** que pertenecen a un *package*.

```
package packageName;
```

donde `packageName` es el nombre del paquete al que pertenece el elemento en donde estamos escribiendo dicha sentencia.

Así pues, imaginemos que tenemos dos clases que queremos que pertenezcan al *package* `mypackage`. El código sería:

<pre>package mypackage; public class A{ //Code }</pre>	<pre>package mypackage; public class B{ //Code }</pre>
---	---

Con el código anterior estamos diciendo que las clases A y B pertenecen al *package* `mypackage`. En un IDE como Eclipse si añadimos la sentencia `package` como primera línea del código de un elemento, el IDE nos dará un error si el elemento no está dentro del package que acabamos de indicar. El IDE nos sugerirá como solución crear el *package* e incluir dentro el elemento. También podemos crear de manualmente un *package* y añadir (p.ej. arrastrando) los elementos que pertenecen a dicho *package*.

Todos aquellos elementos a los que no escribimos la sentencia `package`, es decir, no indicamos explícitamente a qué *package* pertenecen, son tratados por Java como elementos pertenecientes a un mismo package llamado `default` o `unnamed`. El uso de un `default package` es muy común cuando se empieza a programar en Java o se hacen aplicaciones temporales o muy pequeñas. Sin embargo, no se recomienda no crear packages para organizar

un programa. De hecho, IDEs como Eclipse nos crean un *package default* cuando creamos un proyecto Java nuevo con la finalidad de tener organizado nuestro código desde el primer momento.

Si miramos la estructura de directorios de nuestro programa, por ejemplo, en el workspace de nuestro IDE, veremos que dentro de la carpeta src se ha creado una estructura de subcarpetas basada en los *packages* declarados. Así pues, los packages a nivel de organización no dejan de ser directorios con los que organizar nuestro código.

Llegados a este punto, seguramente te estarás preguntando: ¿Existe alguna convención para nombrar *packages*? La respuesta es <>sí>>. Hay que respetar lo máximo posible la convención que explicaremos a continuación para evitar crear dos *packages* llamados exactamente igual, porque de lo contrario, no podremos identificar únicamente dos elementos con el mismo nombre pertenecientes a *packages* distintos.

A la hora de nombrar un *package* debemos tener siempre presente que:

- El nombre un package se escribe, todo él, en minúsculas.
- Si el programa se está desarrollando dentro de o para una compañía, entonces usamos el nombre invertido del dominio de Internet de esa compañía para crear un prefijo (i.e. una raíz común) a partir del cual colgarán el resto de . Por ejemplo, si estamos desarrollando un programa en/para la UOC, cuyo dominio en Internet es uoc.edu, entonces todos los *packages* que creemos irán dentro de edu.uoc: **edu.uoc.mypackage**, **edu.uoc.graphics**, **edu.uoc.assessment**, etc.
- Si dentro de una compañía (mismo programa) existe un conflicto entre dos o más packages, entonces se debe solucionar utilizando convenios internos de la compañía. Por ejemplo, imaginemos que los programadores de las sedes de Barcelona y Madrid de la UOC han creado el *package assessment*. Esto significaría que ambos equipos tendrían edu.uoc.assessment pero con contenido (i.e. clases, interfaces, enumeraciones, etc.) totalmente distintos. Esta situación generaría conflictos si ambos packages se quieren usar/integrar en un mismo programa. Por lo tanto, hay que buscar alguna manera de desambiguar. Cómo se desambigua lo decide en estos casos la propia compañía. Una posibilidad, en este caso, sería añadir la sede: edu.uoc.barcelona.assessment y edu.uoc.madrid.assessment. Con este pequeño cambio ahora sí que no hay ambigüedad posible.
- Existen algunos nombres de dominio de Internet que no son adecuados para crear *packages* o, cuando menos, crean nombres conflictivos que hay que solventar. Por ejemplo:

Dominio de Internet	Nombre de prefijo de <i>package</i> incorrecto	Nombre de prefijo de <i>package</i> correcto
eimt.int	int.eimt	int_.eimt
uoc.8edu	8edu.uoc	eight edu.uoc
uoc-eimt.edu	edu.uoc-eimt	edu.uoc_eimt

- Se recomienda no usar ni las palabras `java` ni `javax` como primera palabra del nombre de un *package*, ya que la API de Java utiliza ambos nombres en la declaración de sus *packages*.

4.5.3. Usando los elementos incluidos dentro de un *package*

Vamos a ver cómo usar un elemento (clase, interfaz o enumeración) incluido dentro de un *package*. A los elementos incluidos en un *package* se les llama miembros del paquete (en inglés, *package members*). Hay tres formas de indicar que queremos usar un elemento de un paquete.

Importar el elemento a usar

Esta manera ya la hemos visto en el apartado 3.5.1. Recordemos que consistía en hacer un import del elemento dentro del código del elemento donde lo queríamos usar.

```
package edu.uoc.example; //si indicamos el package, debe ser la primera línea

import java.awt.List; //Indicamos que queremos la clase List de java.awt

public class GuiFamily{

    private List members;

    public MyAwtList(){
        members = new List(4);
    }
}
```

Importar el package entero

Una manera similar a la anterior es importar todos los elementos incluidos dentro del *package* (i.e. los *package members*), no sólo el elemento que queremos.

```
package edu.uoc.example; //si indicamos el package, debe ser la primera línea

import java.awt.*; //Indicamos que queremos todo lo que esté dentro de
//java.awt. Esto incluye List y también Frame, etc.

public class GuiFamily{

    private List members;

    public MyAwtList(){
        Frame window = new Frame();
        members = new List(4);
    }
}
```

El código anterior hubiera sido “equivalente” a este otro:

```
package edu.uoc.example; //si indicamos el package, debe ser la primera línea

import java.awt.Frame;

import java.awt.List;

public class GuiFamily{
    private List members;

    public MyAwtList(){
        Frame window = new Frame();
        members = new List(4);
    }
}
```

Hemos dicho que el código anterior es “equivalente” porque lógicamente no es exactamente igual a poner simplemente `import java.awt.*`.

Cuando ponemos `import java.awt.Frame` e `import java.List` sólo estamos importando los elementos `Frame` y `List` del *package* `java.awt`, mientras que con `java.awt.*` estamos importando todos los elementos (clases, interfaces y enumeraciones) del *package* `java.awt`. Así pues, con `java.awt.*` también estaríamos importando, por ejemplo, la clase `Label` –que sirve para crear etiquetas (trozos de texto) en una interfaz gráfica–, así como otras clases e interfaces (p.ej. `ItemSelectable`, `Shape`, etc.) del *package* `java.awt`.

¿Cuál de las dos formas es mejor? Sin duda, la primera, i.e. `import java.awt.Frame` e `import java.List`. Es decir, siempre es mejor importar aquello que realmente usamos. El primer motivo es que de esta manera queda claro, para nosotros y otros programadores, qué utilizamos y qué no de un *package*. Además, importando todo puede darse alguna situación no deseada de manera accidental, como por ejemplo, cuando dos *packages* tienen elementos que se llaman igual. Imaginemos que tenemos dos *packages* –`package1` y `package2`–, donde en el primero (`package1`) hay dos clases `A` y `B`, mientras que en el `package2` tenemos `B`, `C` y `D`. Del `package1` usamos la clase `A`, mientras que del `package2` usamos la clase `B`. Si hacemos:

```
import package1.*;
import package2.*;

public class Test{
    private A fieldA;
    private B fieldB;
}
```

¿A qué clase `B` estamos haciendo referencia, a la del `package1` o a la del `package2`? El compilador nos avisará con un mensaje. Por el contrario, si hacemos:

```
import package1.A;
import package2.B;

public class Test{
    private A fieldA;
    private B fieldB;
}
```

Usando el *fully qualified name* del elemento

Otra manera de indicar qué elemento queremos utilizar, es usando el nombre completo del elemento. Es decir, incluyendo el nombre del paquete más el del elemento sin necesidad de importar nada.

```
public class Test{
    private package1.A fieldA;
    private package2.B fieldB;
}
```

Aunque el código anterior es correcto, no suele ser habitual ni práctico escribir todo el tiempo el *fully qualified name* de los elementos. Es mucho más cómodo importar el elemento, como en los apartados anteriores. Sin embargo, esta forma de identificar únicamente un elemento puede ser útil en casos como el siguiente:

```
import package1.B; //podríamos poner import package1.*;
import package2.B; //podríamos poner import package2.*;

public class Test{
    B fieldB1;
    B fieldB2;
}
```

¿De qué clase B es el fieldB1? ¿Del package1 o del package2? ¿Y el fieldB2? La manera de solucionar esta ambigüedad es usando el *fully qualified name*.

```
import package1.B; //podríamos poner import package1.*;
import package2.B; //podríamos poner import package2.*;

public class Test{
    package1.B fieldB1;
    package2.B fieldB2;
}
```

Usando el fully qualified name como en el código anterior, la ambigüedad desaparece para el compilador y no nos dirá nada.

4.5.4. Packages importados automáticamente por Java

El compilador de Java importa automáticamente y de manera implícita dentro de cualquier elemento el propio *package* al que pertenece y el *package* java.lang. Es decir, no es necesario hacer un import ni del propio *package* ni de java.lang.

Importación implícita del propio package

Cuando usamos una clase A del *package* mypackage dentro de una clase B del mismo *package*, no es necesario importar el *package*, puesto que ambas clases están en el mismo *package* y “se ven” la una a la otra.

<pre>package mypackage; public class A{ //Code }</pre>	<pre>package mypackage; public class B{ public B(){ //Podemos usar A porque A y B //pertencen a mypackage A objectA = new A(); } }</pre>
--	--

Importación implícita del package `java.lang`

El compilador de Java incluye automáticamente este *package*. De hecho, ya hemos visto algunas clases que están dentro de este package y hasta ahora no habíamos hecho ningún import. Por ejemplo, la clase `String` es una de las clases que están incluidas de `java.lang`. De igual manera, la clase `System` con la que imprimimos texto por la consola de comandos también está dentro de este *package*.

4.6. Modificador `static`

Como ya has leído en los apuntes teóricos, existe la keyword `static` que puede ser aplicado a un atributo, método o clase anidada.

4.6.1. Atributo estático

Si aplicamos el modificador `static` a un atributo, éste es compartido por todos los objetos pertenecientes a una misma clase. Así pues, deja de ser un atributo de la instancia para ser un atributo de la clase. Es por ello que los atributos con el modificador `static` son llamados o atributos estáticos (*static fields*) o atributos de la clase. Los atributos estáticos ocupan un solo espacio de memoria para todos los objetos, es decir, no hay se pide memoria para ese atributo por cada objeto que se instancia.

Cualquier objeto puede acceder y modificar el valor del atributo estático, pero este atributo también puede ser accedido sin necesidad de crear un objeto de la clase. Así pues, si tenemos:

```
public class A{
    public static int num = 0;
    //TODO
}
```

Desde otra clase podemos hacer:

```
public class B{
    public B(){
        A.num = 8; //los atributos estáticos se pueden acceder a ellos
        //referenciando la clase a la que pertenecen → forma recomendada
        A objectA = new A();
        objectA.num = 9; //los atributos estáticos también pueden ser
        //accedidos mediante un objeto de la clase
        System.out.println(A.num); //9
    }
}
```

Se recomienda acceder a los atributos estáticos a través de la clase, no de un objeto para facilitar la legibilidad del código.

4.6.2. Bloque de inicialización estático

Para los casos en los que queremos inicializar atributos estáticos, éstos pueden ser inicializados en la declaración del propio atributo o en un bloque de inicialización estático (similar al bloque de inicialización que vimos en el apartado 4.1.3).

Imagina que tenemos un atributo `id` que sirve para asignar un identificador a cada objeto `Person` creado y que éste `id` debe ser único. Podríamos tener el siguiente código:

```

public class Person{

    private static int id; //Podríamos haber puesto id = 0 y ahorrarnos el bloque de
    //inicialización estático

    private String name, surname;

    private int age;

    private int height;

    private float weight;

    //Esto es un bloque de inicialización estático. Sólo hay que poner static{}

    static{

        id = 0; //También puede llamar a métodos static
    }

    { //Bloque de inicialización de instancia

        setAge(36);
    }

    //Constructor por defecto

    public Person(){

        this("David", "García", 172, 66);
    }

    //Constructor con argumentos 1

    public Person(String name, String surname){

        this(name, surname, 172, 66);
    }

    //Constructor con argumentos 2

    public Person(String name, String surname, int height, float weight){

        setName(name);

        setSurname(surname);

        setHeight(height);

        setWeight(weight);

        id++;
    }
}

```

Es importante entender que el bloque de inicialización estático, a diferencia del de instancia, sólo se ejecuta una vez por programa. ¿Cuándo se hace esa única llamada? Pues, o bien cuando se crea por primera vez un objeto de la clase, o bien cuando se accede por primera vez a un miembro (i.e. atributo o método) estático de la clase. En ambos casos, el bloque de inicialización estático se llama antes que el constructor o antes de acceder al miembro.

4.6.3. Método estático

Java permite usar la keyword `static` con métodos. Igual que los atributos, los métodos estáticos pertenecen a la clase no a la instancia. Así pues, éstos pueden ser llamados mediante un objeto o mediante el nombre de la clase (esta forma es la recomendada).

```
public class A{
    public static void doSomething(){
        //TODO
    }
}
```

Desde otra clase podemos hacer:

```
public class B{
    public B(){
        A.doSomething(); //los métodos estáticos se pueden acceder a
//ellos referenciando la clase a la que pertenecen → forma recomendada
        A objectA = new A();
        objectA.doSomething(); //los métodos estáticos también pueden ser
//accedidos mediante un objeto de la clase
    }
}
```

Si nos fijamos, el método especial `main` es estático, se llama haciendo `java NombreClase` sin instanciar ningún objeto de la clase.

Es importante tener presente que existe una limitación al usar este tipo de métodos. **Desde un método estático no podemos acceder a atributos ni métodos de instancia directamente** (necesitamos instanciar un objeto).

```
public class A{
    private int num = 0;
    public static void doSomething(){
        num = 8; //ERROR: No podemos acceder a un atributo de instancia
//desde un método de la clase.
        //TODO
    }
}
```

Asimismo, un método estático no puede usar la keyword `this` porque no hay un objeto al que referirse.

4.6.4. Clase estática

En Java no podemos declarar una clase de nivel superior con el modificador `static`. Sólo las clases anidadas puede ser `static`. Para saber más sobre las clases anidadas (y las `static`), te recomendamos leer el apartado *Clases anidadas* de esta guía.

4.7. Modificador abstract

En Java podemos usar la keyword `abstract` tanto para clases como para métodos, pero no para atributos.

4.7.1. Clase abstracta

En Java, cuando una clase es declarada como abstracta significa que dicha clase no puede ser instanciada. Es decir, no podemos crear objetos de esa clase.

```
public abstract class Vehicle{  
    //TODO  
}
```

4.7.2. Método abstracto

Cuando un método es declarado abstracta, estamos diciendo que su codificación la delegamos en subclases que hereden la clase en la que se ubica el método abstracto. Asimismo, cuando declaramos un método abstracto, obligatoriamente su clase también debe ser declarada abstracta.

```
public abstract class Vehicle{  
    public abstract void run();  
    //TODO  
}
```

En el ejemplo anterior, el método `run` es abstracto. Por consiguiente no tiene código (ni siquiera hemos escrito las llaves que limitan el cuerpo del método). Será en una subclase (o subclase de la subclase) donde le proporcionaremos un código.

```
public class Car extends Vehicle{  
    public void run(){  
        System.out.println("I'm a car... brrr!!!");  
    }  
    //TODO  
}
```

4.8. Modificador final

En Java existe la keyword `final` que puede ser aplicado tanto a una clase, un método como a un atributo.

4.8.1. Clase final

Como ya sabes de los apuntes teóricos, una clase `final` es aquella de la que no se puede heredar.

```
public final class Vehicle{  
    //TODO  
}
```

La keyword `final` puede escribirse antes o después del modificador de acceso. El siguiente código es equivalente al anterior.

```
final public class Vehicle{
    //TODO
}
```

Así pues, hacer lo siguiente provocaría un error en tiempo de compilación.

```
public class Car extends Vehicle{ //Error de compilación, Vehicle es final
    //TODO
}
```

4.8.2. Método final

Como ya sabes de los apuntes teóricos, un método declarado como `final` implica que no puede ser sobrescrito en una subclase. Si lo intentamos, obtendremos un error de compilación.

```
public class Vehicle{
    public final void run(){
        //TODO
    }
}
```

Dado la declaración anterior, sería incorrecto hacer lo siguiente:

```
public class Car extends Vehicle{
    public void run(){
        //TODO: no podemos sobrescribir run porque Car es subclase de Vehicle
    }
}
```

4.8.3. Atributo final

Como ya sabes de los apuntes teóricos, un atributo declarado como `final` actúa como constante. Por lo tanto, sólo le podemos asignar un valor una vez en todo el programa. Así pues, una vez asignando un valor, ya no podemos modificarlo.

```
public class Vehicle{
    private final int MAX_SPEED = 250; //Usamos la convención de las constantes
}
```

Si intentamos hacer lo siguiente, obtendremos un error de compilación.

```
public class Vehicle{
    private final int MAX_SPEED = 250; //Usamos la convención de las
//constantes
    public Vehicle(){
        MAX_SPEED = 200; //Error de compilación
    }
}
```

Es muy habitual combinar los modificadores static y final para crear constantes:

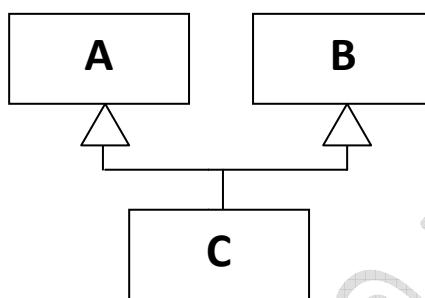
```
public static final double E = 2.718287828459;
```

4.9. Herencia

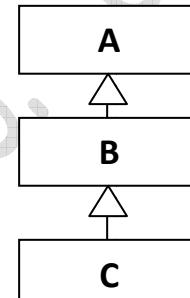
4.9.1. Concepto y sintaxis

La herencia es un mecanismo esencial en la programación orientada a objetos. Ésta consiste en permitir que una clase –llamada subclase– se defina a partir de la codificación de otra ya existente –llamada superclase. Parece algo simple, pero es muy potente. Por ello, es necesario dominarlo, ya que, entre cosas, facilita la reutilización de código.

En Java sólo existe la herencia simple, es decir, una clase sólo puede heredar directamente de otra clase. Es decir, la clase C no puede heredar directamente de A y B, a la vez. Sin embargo, existe la transitividad. Es decir, si una clase B hereda de una clase A, y una clase C hereda de B, entonces la clase C hereda de B y aquello que haya heredado B de A.



Herencia múltiple, no permitida



Transitividad en la herencia, permitida

La sintaxis para indicar que una clase hereda de otra es la siguiente:

```
public class B extends A{
    //TODO
}
```

Con el código anterior estaríamos diciendo que la clase B hereda de la clase A. O, dicho de otro modo, la clase B es una subclase de la clase A y, a su vez, la clase A es una superclase de la clase B.

4.9.2. La clase Object

En este punto cabe decir que en Java todas las clases así como los arrays heredan implícitamente de una superclase llamada Object definida por el propio lenguaje Java: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html>. Es decir, queramos o no, todos los objetos y arrays heredan de Object.

Esta clase define un conjunto de métodos, entre los que destaca: clone, equals y toString.

4.9.3. ¿Qué hereda una subclase?

Es clave saber qué hereda la clase B de la clase A. La respuesta es todo. ¿Todo? Sí, otra cuestión es si la clase B tiene acceso a todos los miembros heredados de A como si éstos se

hubieran declarado en la propia clase B. Sobre esto último, la respuesta es que una subclase sólo tiene acceso directo a los miembros declarados como `public` o `protected` en la superclase. En el caso de los miembros `package-private` sólo tiene acceso directo si la superclase y la subclase pertenecen al mismo *package*.

Hay mucha documentación (incluida la documentación oficial) que mezcla <>lo que hereda la subclase>> con <>lo que se puede acceder desde la subclase>> y, por ello, es muy habitual leer que en Java no se heredan los miembros privados. Por este motivo en esta guía pensamos que es más sencillo de entender cómo funciona la herencia en Java si decimos que se hereda todo de la superclase, pero que la subclase no puede acceder directamente a los miembros `private` de la clase A ni a los miembros `package-private` si superclase y subclase pertenecen a *packages* distintos.

```

public class Animal{
    private String name;
    public int age;
    protected int numberOfLegs;
    boolean vegetarian;

    public Animal(){
        this("Bobby",0,0, false);
    }

    public Animal(String name, int age){
        this(name, age, 0, false);
    }

    public Animal(String name, int age, int numberOfLegs, boolean vegetarian){
        setName(name);
        setAge(age);
        setNumberOfLegs(numberOfLegs);
        setVegetarian(vegetarian);
    }

    public void setName(String name){
        this.name = name;
    }

    public String getName (){
        return name;
    }

    public void setAge(int age){
        this.age = age;
    }

    public int getAge(){
        return age;
    }

    private void setNumberOfLegs(int numberOfLegs){
        this.numberOfLegs = numberOfLegs;
    }

    public int getNumberOfLegs(){
        return numberOfLegs;
    }

    public boolean isVegetarian(){
        return vegetarian;
    }

    public void setVegetarian(boolean vegetarian){
        this.vegetarian = vegetarian;
    }

    protected void speak(){
        System.out.println("My name is "+getName());
    }
}

```

```

public class Dog extends Animal{
    private String owner;
    private String breed;

    public Dog(String name, int age, String owner, String breed){
        super(name, age, 4, false); //el metodo super nos permite llamar
                                   //a un constructor de la superclase
        setOwner(owner);
        setBreed(breed);
    }

    public void setOwner(String owner){
        this.owner = owner;
    }

    public String getOwner(){
        return owner;
    }

    public void setBreed(String breed){
        this.breed = breed;
    }

    public String getBreed(){
        return breed;
    }

    @Override //Sobrescribimos el método speak de Animal
    public void speak(){
        super.speak(); //con super llamamos al método speak de Animal
        System.out.println("Woof!!!");
    }

    @Override
    public String toString(){
        return "I am the dog "+getName()+" ("+getBreed()+"). I am
               "+getAge()+" and I have "+numberOfLegs+" legs.";
    }
}

```

El código anterior requiere un análisis en profundidad:

- La clase Animal es la que menos explicaciones requiere. Hemos asignado un modificador diferente a los atributos para exemplificar el comportamiento de los mismos durante la herencia. Cabe destacar el uso del método especial `this` en el primer y segundo constructor para llamar al tercer constructor y, de esta manera, minimizar el código a escribir.
- La clase Dog hereda de la clase Animal. Es decir, Dog es una subclase de Animal y, por consiguiente, Animal es una superclase de la clase Dog.
- Las subclases no heredan los constructores definidos en su superclase, puesto, como ya sabemos, los constructores (y los destructores) no son considerados miembros de una clase.
- En el constructor de la clase Dog cabe destacar el uso del método especial `super`, el cual se utiliza en la subclase para llamar a uno de los constructores de la superclase. La firma del método `super` debe coincidir con alguna de las definidas en la superclase. Si no se explicita una llamada al método `super()`, entonces el compilador llama automáticamente a `super()`, es decir, llama al constructor por defecto de la superclase. La llamada al constructor de la superclase debe ser la primera instrucción del constructor de una subclase.

- Desde dentro de la subclase Dog además de tener acceso directo a los métodos `setOwner`, `getOwner`, `setBreed`, `getBreed`, `speak` y `toString`, también tiene acceso directo a los métodos de la superclase Animal declarados como `private` o `protected`. Es decir: `getName`, `setName`, `setAge`, `getAge`, `getNumberOfLegs` y `speak`. Fíjate que no tiene acceso directo al método `setNumberOfLegs`, ya que ha sido declarado `private` en la clase Animal. Así pues, en el código de la clase Dog no podemos llamar directamente al método `setNumberOfLegs`. También podremos acceder directamente al atributo `vegetarian` si Animal y Dog están en el mismo *package*.
- En las subclases podemos cambiar el comportamiento de los métodos definidos en la superclase. A este proceso de redifinición se le llama sobrescritura (*overriding*). Esto es justo lo que hace la clase Dog con el método `speak`. Si nos fijamos, hemos vuelto a codificar el método `speak` en la clase Dog con exactamente la misma firma (si la cambiamos entonces estamos sobreponiendo). De esta forma, le asignamos un comportamiento distinto al de la superclase Animal. Si lo hubiéramos recodificado (i.e. sobrescrito), entonces estaría presente en la clase Dog pero con el comportamiento definido en la superclase Animal.
- Para avisar al compilador (y a nosotros mismos) que hemos sobreescrito un método de la superclase es una buena práctica usar la anotación `@Override` justo antes de la firma del método. No es obligatorio su uso, pero sí muy recomendable, ya que el compilador hará comprobaciones extras para avisarnos si hemos cometido algún error a la hora de hacer la sobreescritura.
- En la versión sobreescrita del método `speak`, es decir, el método `speak` de la clase Dog, vemos que llamamos al método `speak` de la superclase. Para hacer esto, que no es obligatorio hacer, debemos usar el objeto `super`. De esta forma, el compilador sabe que estamos llamado al método `speak` de la superclase y no el de la propia clase. Así pues, el método `speak` de un objeto de tipo Dog además de escribir "My name is " también escribirá "Woof ! !".
- Démonos cuenta de que el método `speak` en la subclase Dog tiene un modificador de acceso menos restrictivo que en la superclase Animal. Ha pasado de `protected` a `public`. Esto es correcto, ya que en la subclase podemos asignarle a un método un modificador menos restrictivo que el que tiene en la superclase. Al revés no es correcto. Es decir, a un método sobreescrito no le podemos asignar un modificador de acceso más restrictivo en la subclase que en la superclase. Así pues, si en Dog hubiéramos declarado `speak` como `private`, el compilador nos lanzaría un error.
- El último método que encontramos es `toString`. Como vemos este método está sobreescrito, ¿pero de qué superclase si en Animal no está? Pues bien, como hemos dicho previamente, todas las clases en Java heredan implícitamente de la superclase

Object definida por el propio lenguaje Java. Esta clase Object tiene definidos, entre otros métodos, el método `toString` (que ya habíamos visto anteriormente en esta guía). El método `toString` se utiliza para dar información sobre el objeto. Su código por defecto, es decir, lo que codifica la clase Object es devolver un `String` con una representación interna del objeto e información de la referencia del objeto, es decir, la posición de memoria que ocupa en el *heap*.

- Si nos fijamos en el nuevo código que le hemos asignado al método `toString` en la clase Dog vemos que no llamamos al método `getNumberOfLegs`, ya que éste es `private` en la superclase Animal y, por lo tanto, desde la subclase Dog no tenemos acceso a él. Sin embargo, como el atributo `numberOfLegs` está definido como `protected` en la superclase Animal, sí que podemos acceder directamente a él dentro de la subclase Dog.
- Un código alternativo para el método `toString` con el mínimo número de llamadas a métodos sería:

```
return "I am the dog "+getName()+" ("+breed+"). I am "+age+" and I have
"+numberOfLegs+" legs.;"
```

Fíjate que no podemos acceder al atributo `name` directamente desde Dog porque este atributo fue declarado `private` en la superclase. Así pues, la única manera de acceder a él es indirectamente usando un método declarado en la superclase como `public` o `protected` (o `package-private` si Animal y Dog están en el mismo *package*) que nos dé acceso a dicho atributo.

- Llegados a este punto debemos preguntarnos qué ocurre en la subclase Dog con el atributo `vegetarian` de su superclase Animal. Éste ha sido declarado sin modificador de acceso, es decir, como `package-private`. Pues bien, si la clase Animal y su subclase Dog pertenecen al mismo *package*, entonces desde dentro de la clase Dog se puede acceder al atributo `vegetarian` como si hubiera sido declarado dentro de Dog. En caso contrario, se comporta como si fuera `private`, es decir, no se podría acceder a `vegetarian` directamente dentro del código de Dog.

Ahora veamos otro código Java que utiliza las clases Animal y Dog:

```
public class Example{

    public static void main(String[] args){
        Animal animal1 = new Animal();
        Dog dog1 = new Dog("Lassie",5,"David","Bulldog");

        animal1.speak(); //Si Example pertenece al mismo package que Animal,
//entonces esta llamada es correcta. Si están en packages diferentes, entonces no
//puede ser llamado desde una clase que no sea ni la propia clase ni una subclase

        dog1.speak(); //Imprime "My Name is Lassie" y en la siguiente línea
                    //imprime "Woof!!".

        dog1.age = 10; //es correcto, porque se hereda de Animal como public
        animal1.age = 5; //es correcto, porque es public
```

```

        animal1.numberOfLegs = 5; //incorrecto si Example y Animal pertenecen
//a packages distintos. Llamada correcta si pertenecen al mismo package.

        animal1.setNumberOfLegs(5); //incorrecto, porque es private

        dog1.numberOfLegs = 5; //incorrecto si Example y Animal (no Dog)
pertenecen a packages distintos. Llamada correcta si pertenecen al mismo package.

        dog1.setNumberOfLegs(5); //incorrecto, porque Dog no tiene acceso a
//este método al haber sido declarado private en la superclase Animal

        dog1.setName("Pelusa"); //es correcto, porque se hereda de Animal
//como public

        dog1.setBreed("Chihuahua"); //es correcto, porque es public en Dog

        animal1.setBreed("Chihuahua"); //incorrecto, porque setBreed es de la
//subclase Dog, no pertenece a la superclase Animal. Animal no hereda nada de Dog.

        animal1.vegetarian = true; //el resultado dependerá: si la clase
//Example pertenece al mismo package que Animal, entonces esta sentencia sería
//correcta. En caso de pertenecer a packages diferentes, entonces sería incorrecta.
    }
}
    
```

4.9.4. El modificador `abstract` y la herencia

Como ya vimos en Java existe el modificador `abstract`, el cual puede ser aplicado tanto a clases como métodos. Lo único que afecta en el caso de la herencia es que en la subclase debemos codificar todos aquellos métodos declarados como `abstract` en la superclase. Si no se codifican, entonces éstos deben ser `abstract` en la subclase para que una subclase de la subclase los codifique. Además, la subclase también tendrá que ser una clase declarada como `abstract`.

Una duda que suele surgir con las clases abstractas es si tiene sentido declararles un constructor si una clase abstracta, por definición, no puede ser instanciada. Pues bien, una vez visto cómo se relaciona una subclase con su superclase y que en el constructor de la subclase se debe llamar a algún constructor de la superclase, entonces la respuesta está clara: sí, tiene sentido declarar un constructor en una clase abstracta si ésta será superclase de alguna clase. El constructor de la superclase `abstract` no tiene porqué ser `public`, pero puede ser `protected` para que así tengan acceso directo los constructores de la subclase mediante el método especial `super`.

4.9.5. El modificador `static` y la herencia

Con este modificador hay que diferenciar tres situaciones:

- **Método no estático en la superclase:** si en la subclase sobre escribimos el método de la superclase y lo definimos `static`, entonces el compilador nos dará un error.
- **Método estático en la superclase:** si en la subclase lo sobre escribimos pero sin hacerlo `static`, entonces el compilador nos dará un error.
- **Método estático en la superclase:** si en la subclase lo sobre escribimos manteniendo su calidad de `static`, entonces el método de la superclase queda oculto. De hecho, lo correcto es decir que un método estático en la superclase no puede ser sobreescrito en

una subclase, sino que puede ser ocultado. Así pues, en la subclase en verdad están los dos métodos `static` y se llamará uno u otro en función de si dicho método es invocado usando la superclase o la subclase.

Lo explicado hasta ahora para los métodos, también ocurre con los atributos estáticos. Es decir, en la subclase podemos declarar un atributo llamado exactamente igual que uno de la superclase. En tal caso, lo que hacemos es ocultar dicho atributo y podremos llamar a uno u a otro en función de si lo llamamos usando la superclase o la subclase.

4.9.6. El modificador `final` y la herencia

Como ya vimos este modificador tiene una afectación directa con la herencia. Recordemos que si una clase es `final` entonces no puede ser heredada, por lo tanto, no podremos definir subclases a partir de la clase `final`.

En cuanto a los métodos declarados como `final`, entonces éstos no podrán ser sobreescritos en las subclases. Es decir, el comportamiento definido en la superclase no podrá ser modificado en ninguna subclase.

4.9.7. Ocultación de atributos

En una subclase podemos declarar un atributo con el mismo nombre que un atributo declarado en la superclase, aunque sus tipos sean distintos. Si desde la subclase se puede acceder directamente al atributo de la superclase, al declarar un atributo en la subclase con el mismo nombre lo que estaríamos haciendo es ocultar el atributo heredado de la superclase. Así pues, si queremos acceder al atributo de la superclase deberemos usar el objeto `super`.

<pre>public class A{ protected int value; }</pre>	<pre>public class B extends A{ protected int value; public B(){ super(); value = 10; //clase B super.value = 15; //clase A } }</pre>
---	---

No se recomienda ocultar atributos de una superclase en sus subclases.

4.10. Interfaces

4.10.1. Concepto y sintaxis

Como ya sabemos, las interfaces son contratos que indican qué métodos deben ser codificados en las clases que las implementan. Las interfaces no proporcionan el código de los métodos, sólo sus firmas.

Para definir una interfaz en Java debemos seguir la siguiente sintaxis:

<pre>public interface Movable{ void goAhead(); }</pre>
--

```

    void reverse();

    void left();

    void right();

    double getSpeed();

    void setSpeed(double speed);

}

```

Como podemos ver, hemos asignado el modificador de acceso `public` a la interfaz `Movable`. Esto quiere decir que cualquier clase o interfaz de cualquier package de nuestro programa, puede usar la interfaz `Movable`. En caso de no haber puesto el modificador `public`, entonces sólo las clases e interfaces que pertenecen al mismo package que la interfaz `Movable`, pueden usarla. Otros modificadores no son posibles a alto nivel. Sin embargo, si la interfaz está anidada dentro de otra clase, entonces sí puede ser, por ejemplo, `private`.

En una interfaz no es necesario indicar el modificador de acceso de los métodos. Si no se escribe explícitamente, se entiende que es `public`. De hecho, los métodos en una interfaz Java, por defecto, son `public` y `abstract`. Pero no es necesario escribir ninguno de los dos modificadores porque se sobrentiende que lo son.

4.10.2. Uso de una interfaz

Cualquier clase puede implementar una interfaz mediante la palabra reservada (*keyword*) `implements`. Una clase puede implementar múltiples interfaces a la vez. Para cada una de llas, tendrá que proporcionar el código de los métodos definidos en cada una de las interfaces que implementa. En caso contrario, el compilador dará un error.

```

public class Animal implements Movable, Comparable{
    //TODO: codificar los métodos definidos en las interfaces Movable y Comparable
}

```

Un interfaz no puede implementar otra interfaz, puesto que no podría proporcionarle su código, pero sí puede heredar una o más interfaces.

```

public interface Movable extends Comparable{
    //TODO
}

```

4.10.3. ¿Qué elementos puede contener una interfaz?

Ésta es una pregunta muy habitual. También cabe decir que la respuesta depende de la versión de Java de la que estemos hablando, puesto que el concepto de interfaz en Java ha ido evolucionando con el paso de los años. Eso sí, lo que se introdujo en una versión, se ha mantenido en las siguientes.

Desde Java SE 7 las interfaces sólo permitían constantes (atributos que implícitamente son `public static final`) y métodos abstractos.

```
public interface Movable{

    String ERROR_MSG = "Speed cannot be negative!!";

    void setX(double x);

    void setY(double y);

    double getX();

    double getY();

    void up();

    void down();

    void left();

    void right();

    double getStep();

    void setStep(double step);

}
```

A partir de Java 8, las interfaces fueron capaces de incluir también métodos estáticos y por defecto (en inglés, *default methods*). La inclusión de estos dos tipos de métodos nos permite escribir código dentro de un método.

```
public interface Movable{

    String ERROR_MSG = "Speed cannot be negative!!";

    void setX(double x);

    void setY(double y);

    double getX();

    double getY();

    default void up(){
        setY(getY()+getStep());
    }

    void down();

    void left();

    void right();

    double getStep();

    void setStep(double step);

    static boolean isFast(double speed){

        return speed>120;
    }
}
```

En el código anterior hemos añadido un método `static` llamado `isFast` y hemos convertido el método `up` en un método `default` al que le hemos proporcionado una codificación (el hecho de convertirlo en `default` nos obliga a codificarlo). Así pues, cualquier clase que implemente la interfaz `Movable` ya tendrá una codificación por defecto (de base) para el método `up` y no será necesario, si no quiere, que sobrescriba dicho método.

En Java 9 se introdujo la posibilidad de definir métodos privados y métodos `static` privados. Con ellos se facilita la reutilización de código dentro de las interfaces y su encapsulación. Así pues, los métodos `private` son útiles si, por ejemplo, dos métodos `default` comparten un trozo de código. Gracias a los métodos privados podríamos compartir código entre métodos sin exponerlo fuera de la interfaz. Los métodos privados deben cumplir las siguientes reglas:

- Los métodos privados de una interfaz no pueden ser abstractos. O, dicho de otro modo, los métodos privados de una interfaz deben tener código en su cuerpo.
- Los métodos privados de una interfaz sólo pueden usarse dentro de la interfaz en la que han sido definidos.
- Los métodos privados `static` de una interfaz pueden usarse tanto en métodos estáticos como no estáticos de la interfaz.
- Los métodos privados que no sean `static` no pueden usarse dentro de métodos privados `static`.

```
public interface Movable{

    String ERROR_MSG = "Speed cannot be negative!!";

    void setX(double x);

    void setY(double y);

    double getX();

    double getY();

    default void up(){
        log("Calling up");
        setY(getY()+getStep());
    }

    void down();

    void left();

    void right();

    double getStep();

    void setStep(double step);

    static boolean isFast(double speed){

        log("Calling isFast");

        return speed>120;
    }
}
```

```

private void log(String msg){

    System.out.println(msg);

}
}

```

En el código anterior usamos un método privado llamado `log` para hacer un seguimiento de las llamadas que se hacen a los métodos de la interfaz.

Resumiendo:

Elemento	Desde cuándo se puede usar
Atributos public static final	Java 7
Métodos public abstract	Java 7
Métodos public default	Java 8
Métodos public static	Java 8
Métodos private	Java 9
Métodos private static	Java 9

4.11. Polimorfismo

Como comentamos en el módulo <<Herencia (relación entre clases)>>, el polimorfismo es uno de los pilares del paradigma de la programación orientada a objetos. Es por eso muy importante dominarlo.

Ya sabemos que polimorfismo es la capacidad de que un objeto declarado como tipo superclase pueda comportarse como un objeto de cualquiera de sus subclases. Veamos esto con código:

<pre> public abstract class Animal{ private String name; public Animal(String name){ this.name = name; } public void greeting(){ System.out.print("Ah! "); } public String getName(){ return name; } } </pre>	<pre> public class Dog extends Animal{ public Dog(String name){ super(name); } @Override public void greeting(){ System.out.print("Woof!! "); } } public class Cat extends Animal{ public Cat(){ super("Pelusa"); } @Override public void greeting(){ System.out.print("Meow!! "); } } </pre>
---	--

Si ahora tuviéramos el siguiente código en otra clase, ¿cómo se comportaría el programa?

```
public class Check{
```

```

public static void main(String[] args){

    Animal animalDog = new Dog("Eddie"); //linea 1

    animalDog.greeting(); //linea 2

    Dog dog1 = (Dog) animalDog; //linea 3

    dog1.greeting(); //linea 4

    Dog dog2 = (Dog) animalDog; //linea 5

    Cat cat1 = (Cat) animalDog; //linea 6

    Cat cat2 = animalDog; //linea 7

    animalDog = new Cat(); //linea 8

    animalDog.greeting(); //linea 9

}
}

```

La línea 1 sería correcta. El tipo estático de la variable `animalDog` es `Animal` que es superclase del tipo dinámico que se le asigna: objeto de la subclase `Dog`.

La línea 2 imprimirá por pantalla el texto propio de la clase `Dog`, porque `animalDog` se comporta como un `Dog`, al ser su tipo dinámico `Dog`. Así pues, imprimirá "Woof!!".

En la línea 3 se hace un *casting* de `animalDog` a `Dog`. Ahora la variable `dog1` tiene el objeto `Dog` que hay en `animalDog`. Tanto `animalDog` como `dog1` apuntan a la misma referencia, al mismo objeto.

La línea 4 imprime "Woof!!", puesto que tanto el tipo estático como dinámico de la variable `dog1` es `Dog`.

Tras ejecutar la línea 5, tanto `dog1`, como `dog2` y `animalDog` apuntan al mismo objeto, son la misma referencia.

La línea 6 es errónea. Sin embargo, el error aparece en tiempo de ejecución, no de compilación. El error se debe a que no podemos hacer un *casting* de un objeto `Dog` a un objeto `Cat`.

La línea 7 es errónea, pero a diferencia de la línea anterior el error se produce en tiempo de compilación. Así pues, debemos eliminarla para poder ejecutar el programa. ¿Por qué es errónea la línea 7? Porque el tipo dinámico de `animalDog` es `Animal` y no puede ser asignado/convertido a una variable `Cat`.

La línea 8 es el caso contrario a la línea anterior. Por lo tanto es correcta, puesto que `Animal` es superclase de `Cat`. Justo esto es el ejemplo más sencillo de polimorfismo.

La última línea, la 9, imprimiría por pantalla "Meow!!", ya que el tipo dinámico de la variable `animalDog` es `Cat`.

Una de las utilidades del polimorfismo es simplificar el código. Esto se ve claro si tenemos el siguiente código:

```
Animal [] animals = new Animal[2];
animals[0] = new Dog("Bobby");
animals[1] = new Cat();
for(var animal : animals){
    animal.greeting();
}
```

El código anterior ejecutaría en la primera iteración del enhanced for el método greeting de Dog, puesto que la primera casilla del array animals es un objeto Dog. Es decir, la casilla 0 de animals tiene como tipo estático Animal, pero su tipo dinámico es Dog. En la segunda iteración, por la misma razón, se ejecutaría el método greeting de Cat. Así pues, en vez de tener dos arrays (o dos variables), una de tipo Dog y otro Cat, con una única variable (animals) podemos realizar todas las operaciones según sea su tipo dinámico en cada caso. Es decir, animals adopta muchas formas (poli-morfismo) dentro del mismo programa.

El polimorfismo también es interesante cuando usamos interfaces, esto se ve muy claramente en Java cuando usamos, por ejemplo, interfaces que modelan colecciones. Éste es el caso de la interfaz List.

```
List<Person> lista = new ArrayList<Person>(); //new LinkedList();
public List<Person> getPeople(){
    return lista;
}
```

Fíjate en el getter getPeople. Independientemente de cómo inicialicemos el atributo lista, como objeto de la clase ArrayList o de la clase LinkedList, el método getPeople será el mismo en ambos casos, ya que devolvemos List<Person>, generalizando/asbtrayendo el método. Evidentemente los siguientes dos códigos serían correctos:

```
ArrayList<Person> lista = new ArrayList<Person>();
public ArrayList<Person> getPeople(){
    return lista;
}
```

```
LinkedList<Person> lista = new LinkedList<Person>();
public LinkedList<Person> getPeople(){
    return lista;
}
```

Sin embargo, los dos códigos anteriores tendrían un peor mantenimiento a la larga, puesto que al cambiar la inicialización/instanciación, deberíamos cambiar la firma del método

getPeople para adaptarlo a la nueva clase y, del mismo modo, todo el código donde no generelicemos con el uso de List<Person>.

Finalmente el polimorfismo también nos permite relacionar elementos que no están relacionados por una jerarquía de herencia, sino por una jerarquía de interfaces (es decir, de capacidades). Es decir, gracias al polimorfismo podremos relacionar elementos que a priori no están relacionados. Por ejemplo, si las clases Car y Fish implementan la interfaz Movable, entonces podríamos hacer:

```
Movable [] elements = new Movable[2];
elements[0] = new Car();
elements[1] = new Fish();
for(var element : elements){
    element.up(); //el método up está declarado en la interfaz Movable y, por
    //lo tanto, tanto la clase Car como la clase Fish le han tenido que dar un código.
}
```

4.12. Excepciones

4.12.1. Concepto

Para entender qué es una excepción, imaginemos que un programador quiere leer datos de un fichero de texto en su programa. Lo habitual en lenguajes que no incluyen el manejo de excepciones es dejar a juicio del programador el control de las excepciones (casos anómalos o indeseados). Veamos una posible solución adoptada por un programador (sin entrar en detalles de código, lo importante es la estructura):

```
if(fichero existe){
    //abrimos fichero
    if(fichero abierto correctamente){
        while(existen líneas de texto){
            //leemos una línea de texto
            if(no ha habido problema con la lectura){
                //Hacemos algo con la información leída
            }else{
                //ERROR: Mensaje de que ha habido un problema al leer
            }
        }
        //cerramos el fichero
        if(fichero mal cerrado){
            //ERROR: Mensaje de que el fichero no se ha cerrado correctamente.
        }
    }else{
        //ERROR: Mensaje de que el fichero no se ha abierto correctamente.
    }
}else{
    //ERROR: Mensaje de que el fichero no existe.
}
```

En el código anterior, si estuviera dentro de un método, en vez de escribir mensajes de error podríamos haber optado por devolver un código de error y desde donde se invocó escribir un mensaje u otro dependiendo de dicho código.

Como podemos ver en el código anterior, la lógica del programa en sí y la gestión de las excepciones está mezclada. Esto hace que el código sea más difícil de leer y entender. Es por

eso que Java incluye la gestión de excepciones quitando responsabilidad a los programadores. Veamos el código anterior en su versión "Java":

```

try{
    //abrimos fichero
    while(existen líneas de texto){
        //leemos una línea de texto
        //Hacemos algo con la información leída
    }
    //cerramos el fichero
}catch(excepción de que el fichero no existe){
    //Hacer algo, i.e. tratar la excepción.
}catch(excepción al abrir fichero){
    //Hacer algo, i.e. tratar la excepción.
}catch(excepción al leer una línea del fichero){
    //Hacer algo, i.e. tratar la excepción.
}catch(excepción al cerrar el fichero){
    //Hacer algo, i.e. tratar la excepción.
}

```

Como puedes ver, gracias al bloque try-catch, Java nos permite separar la lógica del programa de la gestión de las excepciones. El flujo normal del programa está dentro del bloque try, mientras que cada excepción es tratada en el bloque catch correspondiente. Esta separación realmente facilita mucho la lectura y entendimiento del código.

4.12.2. Declarando y lanzando una excepción

Continuemos viendo otro código:

<pre> package edu.uoc.exceptions; public class Person{ private int age = 30; public boolean setAge(int age){ if(age<0){ return false; }else{ this.age = age; return true; } } } </pre>	<pre> package edu.uoc.exceptions; public class Main{ public Main(){ Person david = new Person(); if(!david.setAge(-1)){ System.out.print("Incorrect age"); } } } </pre>
---	---

En el código anterior si pasamos como argumento al método setAge de la clase Person un valor negativo, entonces este método no asigna el valor y devuelve false para avisar de que ha habido un error (i.e. el intento de asignar una edad negativa), no pudiéndose realizar la acción esperada correctamente. En la clase Main se comprueba el valor devuelto por setAge del objeto david para saber si dicho método se ha ejecutado correctamente o no. Si el método setAge devuelve el valor false, entonces quiere decir que dicho método no se ha ejecutado bien. Así pues, desde donde se ha hecho la llamada a setAge gestionamos este caso no deseado advirtiendo al usuario final con un mensaje por pantalla.

El uso de un boolean (o de devolver un valor numérico conocido que haga de código de errores, como se hace en el protocolo HTTP con los códigos 404, 500, etc.) es una manera muy habitual de controlar y gestionar los errores o situaciones anómalas en lenguajes basados en el paradigma de la programación estructurada. Sin embargo, no es la mejor manera para tratar los errores, anomalías o casos excepcionales. Es por ello que lenguajes como Java incluyeron el concepto de excepción o evento excepcional.

Definición de excepción

Una *excepción* es un evento indeseado que ocurre durante la ejecución de un programa y que interrumpe, de alguna manera, el flujo normal (o deseado/esperado) de ejecución del programa.

Cuando ocurre una excepción, el programa crea un objeto llamado *exception object* que incluye información valiosa sobre el error como, por ejemplo, el tipo de excepción, dónde ha sucedido dicho error, etc. El *exception object* es lanzado al JVM quien lo gestiona. Este proceso de lanzar *exception objects* es lo que se conoce como lanzar una excepción (en inglés, *throwing an exception*). Veamos cómo sería el código anterior usando excepciones.

```
package edu.uoc.exceptions;

public class Person{
    private int age = 30;

    public void setAge(int age) throws Exception{
        if(age<0){
            throw new Exception("Incorrect age");
        }
        this.age = age;
    }
}
```

Vemos que el método `setAge` de la clase `Person` ha experimentado algunos cambios. Para empezar ya no devuelve nada (es `void`). En segundo lugar, hemos añadido `throws Exception` al de su firma. Con esto estamos diciendo que este método lanza una excepción. ¿Cuándo un método lanza una excepción? Pues puede haber dos momentos:

- Cuando dentro del método se llama a otro método que lanza una excepción (es decir, cuando la firma de este método que se llama tiene `throws Exception`)
- Cuando el programador lo indica explícitamente con la instrucción `throw new Exception("Mensaje de error a mostrar")`. En el caso de `setAge` sucede el segundo caso. Como puedes ver, en el `throw` se crea un objeto de la clase `Exception` (pertenece al package `Exception`).

En el ejemplo anterior, si el programa ejecuta la línea de código `throw new Exception("...")`, entonces el método se parará y saldrá. Es decir, no ejecutaría las líneas restantes del método. En el caso de `setAge`, si éste lanzara la excepción, no se ejecutaría nunca la línea de código `this.age = age`. Lo mismo ocurriría si donde ahora tenemos `throw`

`new Exception(" . . .")` tuviéramos en su lugar una llamada a otro método que lanza una excepción y se lanzara dicha excepción.

4.12.3. Flujo que sigue una excepción

Cuando se sale de un método por culpa de una excepción, el programa no se detiene de golpe, sino que lanza una excepción hacia arriba esperando que en algún momento se capture esta excepción y se trate. Cuando decimos que la excepción se lanza hacia arriba, queremos decir que ésta va hacia atrás en el orden de ejecución del programa. Este orden de ejecución está formado por una pila con los métodos que han sido llamados –llamada *call stack*– hasta que ha ocurrido la excepción.

Así pues, imaginemos que tenemos la clase Person definida anteriormente cuyo método `setAge` lanza una excepción cuando la edad pasada como parámetros es negativa. Y que además tenemos el siguiente código:

```
package edu.uoc.exceptions;

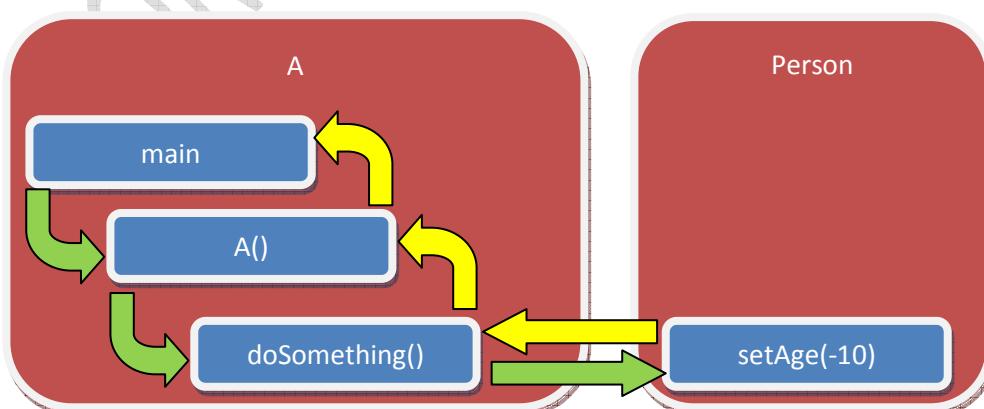
public class A{

    public A() throws Exception{
        doSomething();
    }

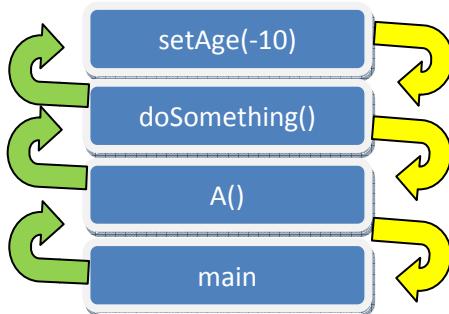
    public void doSomething throws Exception{
        Person p = new Person();
        p.setAge(-10);
    }

    public static void main(String[] args){
        A objectA = new A();
    }
}
```

Ahora ejecutamos en la consola de comandos la instrucción `java A`. Entonces la ejecución del programa en un formato visual sería:



Fíjate que no está la llamada al constructor de Person porque se ejecuta sin problemas y digamos que desaparece, quizás viéndolo en formato *call stack* te es más fácil de entender:



En ambos gráficas las flechas verdes indican llamada a un método y las flechas amarillas son lanzamientos de la excepción (i.e. el camino que sigue una excepción hasta que es tratada). Sabiendo esto, vamos a analizar el programa. En primer lugar, al hacer `java A` se ejecuta el `main` de dicha clase y en ella se llama al constructor de la clase `A` cuando instanciamos un objeto para la referencia `objectA`. En el constructor de la clase `A` se llama al método `doSomething` de la clase `A` que tiene dos instrucciones. Primero crea un objeto de la clase `Person` (llamada que al terminar correctamente desaparece de la *call stack*) y a continuación llama al método `setAge` con un argumento negativo (`-10`). Al ejecutar el código del método `setAge` se lanza una excepción porque el valor pasado como argumento no es correcto y, por consiguiente, el flujo de ejecución entra en el `if` que crea (`new`) un objeto `Exception` y lo lanza (`throw`). Este *exception object* es enviado hacia arriba (flecha amarilla), pues así lo indica la coletilla `throws Exception` de la firma del método `setAge`. Entonces, el objeto `Exception` creado en `setAge` es recibido por el método `doSomething` que decide si trata la excepción o no. En este caso hemos decidido que no la trate (o como se dice informalmente, que no la *capture*). Puedes ver que hemos decidido que el método `doSomething` no la capture porque la firma del método `doSomething` tiene añadido `throws Exception`. Por lo tanto, el método `doSomething` está delegando/pasando el tratamiento de la excepción del método `setAge` hacia arriba (hacia quien le ha llamado), en este caso, hacia el constructor de la clase `A`. Como podemos ver, el constructor de la clase `A` también delega hacia arriba el tratamiento del error. Cabe decir que, en cualquier programa en Java, el método especial `main` es el último lugar donde podemos capturar y tratar las excepciones lanzadas. Si en el `main` no capturamos y tratamos el error, entonces el programa termina repentinamente.

4.12.4. Tratando/capturando una excepción

Vamos a ahondar en el bloque `try-catch` que ya hemos visto. Como hemos explicado, cuando se sale de un método por culpa de una excepción, el programa no se detiene de golpe, sino que va retrocediendo en el *call stack* para ver si alguien trata/captura la excepción. ¿Cómo indicar en un punto del código que queremos capturar la excepción? Con un bloque llamado `try-catch` que tiene la siguiente sintaxis:

```

try{
    //Código que puede generar una excepción
} catch(ExceptionType variable){
    //Código a ejecutar en caso de lanzarse una excepción desde el código del try
}
  
```

Vamos a seguir con el ejemplo del apartado anterior, pero ligeramente modificado para introducir un try-catch:

```

package edu.uoc.exceptions;

public class A{

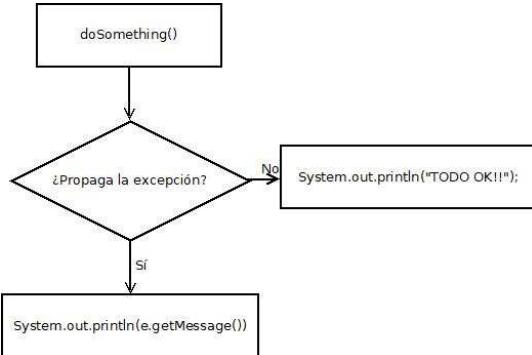
    public A(){ //Hemos quitado "throws Exception" y ya no propaga la excepción
        try{
            doSomething();
            System.out.println("TODO OK!!"); //se ejecuta si doSomething no lanza
//una excepción
            }catch(Exception e){
                System.out.println(e.getMessage());//Imprimirá el texto "Incorrect
//age!" que indica el método setAge de la clase Person si doSomething lanza una
//excepción
            }
            //Si aquí hubiera código siempre se ejecutaría...
        }

        public void doSomething throws Exception{
            Person p = new Person();
            p.setAge(-10);
        }

        public static void main(String[] args){
            A objectA = new A();
        }
}

```

Si nos fijamos, ahora el constructor de la clase ya no propaga/delega la excepción del método doSomething hacia quien le ha llamado (en este caso el método main), ya que no está la coletilla throws Exception en la firma del constructor de la clase A. Ahora en la clase A capturamos o tratamos la excepción generada por doSomething (si se genera). Si el método doSomething no propaga ninguna excepción, es decir, al llamar a setAge no se lanza una excepción, entonces se ejecutaría el println con el mensaje "TODO OK!!" y el constructor de la clase A terminaría. En caso contrario, como sucede en el ejemplo, como p.setAge(-10) hace que setAge lance una excepción, doSomething propagará esta excepción y será en el constructor de la clase A donde se tratará. ¿Dónde se tratará? En el trozo de código del bloque catch. Es decir, al producirse una excepción, todas las sentencias que haya después del método que ha provocado la excepción no se ejecutarán. En el ejemplo no se ejecutará la sentencia println con el texto "TODO OK!!" y el flujo del programa ser irá de doSomething al código que hay en el bloque catch. Es decir, después de doSomething se ejecutará el println que imprime la información que hay en el objeto e, en este caso, el mensaje que tiene el objeto e de tipo Exception asignado en el método setAge: "Incorrect age!". Toda esta explicación gráficamente sería:



Cuando el diagrama anterior dice <<¿Propaga la excepción?>> quiere decir si el método doSomething recibe una excepción lanzada por setAge y esta se ha propagado hasta el constructor de la clase A. Si hay una excepción se para el flujo de ejecución del bloque try y el programa sigue en el bloque catch, por lo que ejecuta el código que hay en dicho bloque, en el caso del ejemplo: System.out.println(e.getMessage()); Si no hay excepción, la ejecución continúa de manera normal con el resto del código ubicado dentro del try.

Cuando usamos el bloque try-catch estamos indicando que en ese punto queremos capturar una excepción que se produzca dentro del try y tratarla dentro del catch.

4.12.5. Checked y unchecked exceptions

En Java hay dos categorías de excepciones: las *checked* y las *unchecked*. Así pues, vamos a definir ambos tipos:

Definición de checked exception

Las excepciones *checked* (o comprobadas o verificadas) son aquellas para las que el compilador comprueba que se capturen o se propaguen. En caso de que una excepción sea checked y ni se capture ni se propague, entonces obtendremos un error en tiempo de compilación.

Por su parte:

Definición de unchecked exception

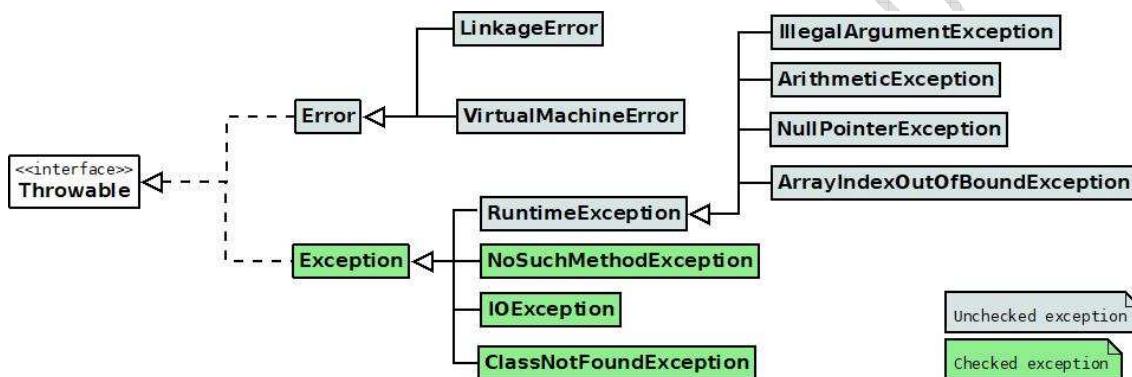
Son aquellas excepciones para las cuales el compilador no comprueba que sean capturadas o propagadas. Por consiguiente, como programadores no tenemos obligación de capturarlas ni propagarlas.

¿Por qué unas excepciones son *checked* y otras *unchecked*? Java considera que las *unchecked* son todas aquellas excepciones en las que el programador poco puede hacer para lograr que el flujo de ejecución del programa no se vea alterado. En otras palabras, Java considera que de las excepciones *unchecked* es difícil recuperarse, es decir, encontrar una solución que permita continuar la ejecución del programa como si no se hubiera dado dicha excepción.

Por ejemplo, cuando dividimos un número entre cero se produce una excepción llamada *ArithmetricException* que es de tipo *unchecked*. Poco podemos hacer, a priori, si hay una división entre cero, seguramente se debe a una mala lógica del programa ya que antes de

dividir dos números deberíamos comprobar que el denominador sea distinto a cero. Lo mismo ocurre cuando intentamos acceder a una casilla que no existe de un array. Por ejemplo, si queremos acceder a la casilla/posición 24 de un *array* de 5 casillas, la JVM durante la ejecución del programa (no en tiempo de compilación) lanzará una excepción de tipo `ArrayIndexOutOfBoundsException`. En este caso, poco podemos hacer. Nuevamente se debe a un error en nuestro código: o bien hemos usamos un índice cuyo valor está mal asignado (un error clásico cuando se empieza a aprender a usar los bloques de iteración), o bien no hemos comprobado que el índice que se usa para acceder a un elemento del *array* esté dentro del rango correcto del *array*.

Las excepciones en Java están organizadas en torno a una jerarquía de clases y, por lo tanto, basada totalmente en el mecanismo de herencia (concepto que veremos tanto en los materiales teóricos como en esta guía). Muy resumidamente y sin entrar en demasiados detalles veamos una pequeña muestra de estas clases relacionadas con las excepciones:



Todas las clases en verde son *checked*, lo que quiere decir que se deben capturar o propagar obligatoriamente en nuestro código. ¿Por qué? Para este tipo de errores podemos gestionar qué hacer en el programa para solucionar la aparición de dicha excepción, p.ej. realizar una acción alternativa, hacer un *rollback* (i.e. deshacer operaciones que se han hecho), salir del programa avisando amablemente al usuario final (no repentinamente) y cerrando ficheros, sockets, conexiones a bases de datos para dejarlos en un estado consistente, etc.

En cambio, todas las clases en color grisáceo son *unchecked* y no es necesario capturarlas ni propagarlas. Como puedes ver, son errores (de hecho heredan/cuelgan de la clase `Error`, excepto `RuntimeException` y sus subclases) que ocurren con poca frecuencia y que poco se puede hacer como programador para solucionar el problema.

4.12.6. Declarando, lanzando y capturando más de una excepción

Llegados a este punto cabe destacar que, como hemos visto anteriormente, podemos tener tantos `catch` como excepciones queramos tratar de manera diferente y es que dentro de un bloque `try` se pueden dar diferentes tipos de excepciones. Es más, un único método puede lanzar más de una excepción. Por ejemplo:

```

public void read(String fileName) throws FileNotFoundException, IOException{
    File file = new File(fileName);
    FileReader fr = new FileReader(file); //Lanza FileNotFoundException
    int data = fr.read(); //Lanza IOException
  
```

```

        while(data!=-1){
            data = fr.read(); //Lanza IOException
        }
        fr.close(); //Lanza IOException
    }
}

```

Entonces donde se llama al método `read` debemos hacer:

```

try{
    read("c:/file.txt");
}catch(FileNotFoundException e){ //Si read lanza FileNotFoundException, este catch
//lo capturará
    System.out.println("File not found");
}catch(IOException e){ //Si read lanza IOException, este catch lo capturará
    System.out.println("Error reading file");
}

```

Imaginenmos que el código anterior lo cambiáramos para hacer esto:

```

try{
    read("c:/file.txt");
}catch(FileNotFoundException e){
    e.printStackTrace();
}catch(IOException e){
    e.printStackTrace();
}

```

Fíjate que tanto si se captura una excepción de tipo `FileNotFoundException` como `IOException`, lo que hacemos es lo mismo: llamar al método `printStackTrace` del *exception object*. Para evitar repetir código, desde JDK 7 es posible hacer lo siguiente:

```

try{
    read("c:/file.txt");
}catch(FileNotFoundException | IOException e){
    e.printStackTrace();
}

```

Como podemos ver, el código se ha reducido facilitando la legibilidad del código.

Ahora observa el siguiente código el cual parece similar al que llama `printStackTrace` por separado, pero su comportamiento no es exactamente idéntico:

```

try{
    read("c:/file.txt");
}catch(IOException e){
    e.printStackTrace();
}catch(FileNotFoundException e){
    e.printStackTrace();
}

```

Si te fijas, lo que hemos cambiado es el orden de los `catch`, anteponiendo el `catch` de `IOException` al de `FileNotFoundException`. Este sutil cambio puede parecer menor y que, por lo tanto, no tenga consecuencias en nuestro código, pero las tiene. ¿Por qué? Porque nunca se ejecutará el `catch` de `FileNotFoundException`. ¿Por qué? Porque, para empezar, los `catch` se ejecutan en orden de aparición. Además

`FileNotFoundException` es una subclase de la clase `IOException` y, por consiguiente, `IOException` es capaz de capturar tanto las excepciones `IOException` como las de todas sus subclases. Dadas estas dos condiciones, es lógico que nunca se ejecute el `catch` de `FileNotFoundException`. Para que lo veas más claro:

```
try{
    read("c:/file.txt");
}catch(Exception e){
    e.printStackTrace();
}catch(IOException e){ //Nunca se ejecutará
    e.printStackTrace();
}catch(FileNotFoundException e){ //Nunca se ejecutará
    e.printStackTrace();
}
```

El código anterior utiliza en primera instancia un `catch` de `Exception`. Como ya hemos visto `IOException` es una sublcase de `Exception` y, como `FileNotFoundException` es subclase de `IOException`, también lo es de `Exception`. Así pues, el `catch` de `Exception` puede capturar cualquier *exception object* que sea de una subclase de `Exception`. Sabiendo esto, es importante que en caso de necesitar un caso default de captura de excepciones, es decir, un `catch` de `Exception`, éste sea el último que escribas en tu código.

Llegados a este punto quizás te preguntes y, ¿por qué no hacemos un `catch` de `Exception` y así simplificamos el código al máximo? Pues porque nos puede interesar hacer acciones diferentes en función del tipo de excepción.

4.12.7. Bloque finally

El bloque try-catch puede completarse, opcionalmente, con el bloque `finally`. Es decir:

```
try{
    //método que lanza una excepción
    doSomething();
}catch(Exception1 e){
    //Hacer algo, i.e. tratar la excepción de tipo Exception1.
}catch(Exception2 e){
    //Hacer algo, i.e. tratar la excepción de tipo Exception2.
}finally{
    //Este bloque de código siempre se ejecuta, tanto si hay try como si hay catch
}
```

El bloque `finally` tiene la particularidad de que siempre se ejecuta independientemente de qué ocurra. Es decir, se ejecute correctamente el `try` o se lance una excepción que capture un `catch`, el código ubicado dentro de `finally` se ejecutará a continuación del código `try` o `catch`. También se ejecutará si lanzamos una excepción y la propagamos. Antes de hacer la propagación de la excepción (y salir del método), el bloque `finally` se ejecutará. El único caso en que no se ejecuta el bloque `finally` es si el código que hay dentro del `catch` que ha capturado la excepción finaliza de manera inesperada.

4.12.8. Métodos del exception object

Cualquier objeto que sea de una clase que hereda de `Throwable`, es decir, todas las que usamos para propagar excepciones, tiene métodos que nos proporcionan información adicional sobre la excepción. Veamos los tres más interesantes:

`getMessage`

Devuelve un `String` con el texto que se haya indicado en el constructor del *exception object*, p.ej. si se ha usado `Exception(String msg)` o `Throwable(String msg)`, etc.

`toString()`

Devuelve un `String` con una breve descripción del objeto. Básicamente el nombre de la clase de tipo `Throwable` seguido del símbolo : y el texto de `getMessage()`.

`printStackTrace`

Imprime a través de `System.err` el contenido del método `toString` seguido por la traza del *call stack*.

4.12.9. Sentencia try-with-resources

En JDK 7 se añadió esta sentencia para cerrar recursos de forma automática y, de este modo, simplificar el código. Aquellas referencias cuyas clases implementan la interfaz `AutoCloseable` pueden declararse en el bloque `try-with-resources` y, así, sus métodos `close()` serán llamados automáticamente después del bloque `finally`.

Veamos un ejemplo de código antes de JDK 7:

```
public int read(String fileName) throws IOException{
    File file = new File(fileName);
    FileReader fr = new FileReader(file); //Lanza FileNotFoundException

    try{
        return fr.read(); //Lanza IOException
    }finally{
        if(fr != null) fr.close(); //Lanza IOException
    }
}
```

El código anterior utilizando la nueva funcionalidad `try-with-resources` de JDK 7:

```
public int read(String fileName) throws IOException{
    File file = new File(fileName);
    try(FileReader fr = new FileReader(file)){ //Lanza FileNotFoundException
        return fr.read(); //Lanza IOException
    }
}
```

El código anterior implícitamente realiza el mismo código que hacíamos en el bloque `finally`, pero nuestro código se ha simplificado.

La mayoría de clases e interfaces relacionadas con entrada y salida implementan la interfaz `AutoCloseable`: manipulación de ficheros (p.ej. `FileReader`), leer y escribir datos (`InputStream`), gestión de flujos de red (`DatagramSocket`), conexión a bases de datos (`Connection`)

4.13. Enumeraciones

4.13.1. Concepto y sintaxis

Si has codificado en C/C++ u otros lenguajes, habrás usado enumeraciones (en inglés, enumeration) o simplemente enum. Como ya sabrás, los enum son un tipo definido por el propio programador. Normalmente se utilizan cuando queremos restringir los posibles valores que puede tomar una variable/atributo o estos valores conforman un dominio concreto y conocido, p.ej. días de la semana, meses del año, planetas del Sistema Solar, etc.

En Java, no obstante, el tipo enum (introducido en JDK 1.5) es más potente que en otros lenguajes como por ejemplo C. Por defecto, Java define cada enum como una clase y, de hecho, en su declaración se pueden incluir atributos y métodos y, a la vez, esta “clase” incluye por defecto algunos métodos, como por ejemplo `values()`, que devuelve todos los valores del enum. De hecho, la manera ideal de declarar un enum es poniendo su código dentro de un fichero .java que tenga el mismo nombre que el enum (igual que una clase). No obstante, un enum se puede declarar dentro de una clase ya existente.

La sintaxis básica de un enum en Java es:

```
modificadorAcceso enum{
    ITEM1, ITEM2, ...
}
```

Por ejemplo:

```
public enum HandSign{
    ROCK, PAPER, SCISSORS
}

private enum Day{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Como ya hemos dicho reiteradamente, Java trata los *enumeration* como clases. Eso sí, son clases especiales que proporcionan una implementación *type-safe* (seguridad de tipos) de datos constantes. Es decir, podemos declarar una variable del tipo del *enumeration* a la cual sólo se le puede asignar los valores del *enumeration*, ninguno más.

```
Day d; //Declaramos una variable del tipo enum Day
d = Day.MONDAY; //Asigna un valor correcto del tipo Day (enum)
d = 0; //Error en tiempo de compilación
```

4.13.2. Diferencia entre usar constantes y un *enumeration*

Vamos a comentar la diferencia que hay entre hacer el enum de HandSign o utilizar constantes enteras que simulan un *enumeration* de la siguiente manera:

```
public class Game{
    public static final int STONE = 0; //Constante
    public static final int PAPER = 1;
```

```

public static final int SCISSORS = 2;
...
int playerHand = SCISSORS;
...
}

```

Usar constantes tiene los siguientes problemas:

- Ya que la variable no es de tipo enum, entonces podemos asignarle, por error, cualquier valor. En el ejemplo, a `playerHand` se le podría asignar el valor 33 y no daría error en tiempo de compilación.
- No crea un namespace. Mientras que los enum siempre crean el namespace con el nombre del *enumeration*. Así para usar el valor SCISSORS debemos usar HandSign.SCISSORS. ¿Qué sucedería si hubiera otra variable llamada SCISSORS en otra parte del código? Pues que tendríamos una colisión y deberíamos renombrar una de las dos variables, p.ej. HAND_SCISSORS en vez de simplemente SCISSORS.
- Fragilidad (en inglés, *brittleness*). Si quieras añadir un elemento nuevo dentro del enum, éste lo tienes que añadir después, porque si lo haces en medio, debes modificar todo el código. Imagina el caso de los días. Has creado una constante llamada SUNDAY con el valor 0, MONDAY con valor 1, etc., pero te das cuenta pasado un tiempo que te has olvidado el viernes y que el sábado es SATURDAY = 5. ¿Qué haces si quieres seguir la lógica de la semana? Pues si quieres que FRIDAY sea el 5 y SATURDAY el 6, lo puedes hacer, pero en el programa tendrás que modificar todo lo que haga referencia al "5", porque ahora es el viernes, no el sábado y puede que el programa tenga que hacer cosas diferentes si el valor que se le pasa es un viernes o un sábado.
- Print's sin información valiosa. Porque las constantes sólo son enteros, al hacer un `println` éste imprime los valores enteros: 0, 1 y 2. Esto no nos da información sobre lo que representan. En cambio, el enum imprime STONE, PAPER, SCISSORS.

Como ves, los *enumeration* introducen muchas ventajas respecto a simularlos con enteros u otro tipo de datos primitivos (p.ej. un `char`). Pues bien, si los *enumeration* ya conllevan ventajas de por sí, en Java aún más, porque como ya hemos comentado, Java trata los *enumeration* como una “clase” y les añade características extra respecto a los *enumeration* de otros lenguajes de programación.

4.13.3. Compatibilidad con switch

Los enum pueden usarse con `switch` como si de un entero u otro tipo básico se tratara.

```

enum Operation{
    PLUS, MINUS, TIMES, DIVIDE;
}

public class MainApp{

```

```

        double operate(double x, double y, Operation op){
            switch(op){
                case PLUS: return x + y;
                case MINUS: return x - y;
                case TIMES: return x * y;
                case DIVIDE: return x/y;
            }
        ...
    }
}

```

4.13.4. Implementación interna de las constantes (de los valores)

Además el compilador crea una instancia de la clase para cada constante definida dentro del enum. Estas instancias son `public static final`. Es decir, no se pueden modificar (`final`) y se acceden a ellas anteponiendo el nombre del enum. Por ejemplo: `Day.MONDAY`. Asimismo, no se puede instanciar ningún objeto de tipo del enum.

4.13.5. Herencia de la clase Enum

Cuando un enum es definido, Java crea una clase que hereda de la clase `Enum`. Por este motivo, ningún enum puede heredar de otra clase o enum. La clase `Enum` aporta los siguientes métodos:

```

public final String name() //Devuelve el nombre de la constante tal como se
//declara. No se puede sobreescribir.

public String toString() //Devuelve el nombre de la constante, tal como está
//en la declaración. Se puede sobreescribir. Es más utilizado y recomendable
//que name().

public final int ordinal() //Devuelve el número ordinal de la constante.
//Empieza por 0 (cero).

public static T valueOf(String name) //Devuelve la constante del enum a partir
//de un String de la constante.

```

Veamos su uso:

```

Day day = Day.valueOf("MONDAY"); //day apunta a Day.MONDAY
System.out.println(day.name()); //Imprimirá MONDAY
System.out.println(day); //Imprimirá "MONDAY" (llama a toString());
System.out.println(day.ordinal()); //Imprimirá 1

```

Además los enum tienen un método muy interesante llamado `values()` que devuelve un array con los valores del enum.

```

Day[] week = Day.values(); //devuelve un array en el que cada casilla es un
//valor del enum Day.

```

4.13.6. Constructor y miembros

Toda variable de tipo enum es una referencia a una zona de memoria, igual que una clase, una interfaz o un array. Esta variable es implícitamente final, ya que sus valores no pueden

modificarse. Sin embargo, el enum, al ser como “una clase especial”, puede definir un constructor y miembros del enum (i.e. atributos y métodos). Esto es precisamente lo que hace al enum de Java más poderoso en comparación con el enum de otros lenguajes de programación. Veamos un ejemplo:

```
public enum AlertLevel{
    HIGH (3), // llama al constructor con valor 3
    MEDIUM (2), // llama al constructor con valor 2
    LOW (1) //llama al constructor con valor 1
    ; //cuando se añaden atributos o métodos, se //necesita poner ;

    private final int levelIntensity;

    //Constructor
    private Level(int levelIntensity){
        this.levelIntensity = levelIntensity;
    }

    public int getLevelIntensity(){
        return levelIntensity;
    }

    @Override
    public String toString(){
        return getLevelIntensity()+" | "+this.name();
    }
}
```

Es importante resaltar que el constructor de un enum sólo puede ser `private` o `package-private` (por defecto).

4.13.7. Métodos abstractos

En un enum podemos declarar un método abstracto y hacer que cada valor, que como hemos dicho es una instancia, codifique su implementación/comportamiento.

```
public enum AlertLevel{
    HIGH (3){
        public AlertLevel next(){
            return LOW;
        }
    },
    MEDIUM (2){
        public AlertLevel next(){
            return HIGH;
        }
    },
    LOW (1){
        public AlertLevel next(){
            return MEDIUM;
        }
    };
}
```

```

private final int levelIntensity;

private Level(int levelIntensity){
    this.levelIntensity = levelIntensity;
}

public abstract AlertLevel next(); //Método abstracto
}

```

Su uso podría ser el siguiente:

```

for(AlertLevel level : AlertLevel.values()){
    System.out.println(level+", next is "+level.next());
}

```

4.14. Generics

4.14.1. Concepto

Los *generics* fueron introducidos en JDK 5. Estos permiten la definición y uso de tipos (i.e. clases e interfaces) y métodos genéricos. Los tipos y métodos genéricos difieren de los tipos y métodos “normales” ya que los primeros tienen parámetros de tipo.

Así pues, por ejemplo la clase `ArrayList<E>` es un tipo genérico (más comúnmente llamado **tipo parametrizado**) que tiene un parámetro de tipo llamado `E`. Cuando usamos `ArrayList<Person>` o `ArrayList<Animal>`, a éstos se les llama tipos parametrizados, donde `Person` y `Animal` son los argumentos de tipo, respectivamente. Veremos esto con más calma a continuación.

4.14.2. Clase genérica

Con Java podemos crear nuestras propias clases genéricas. Estas clases tienen uno o más parámetros de tipo, que son tipos genéricos. Su sintaxis más simple es el siguiente:

```

public class GenericClass <T>{
    private T element;
    //TODO
}

```

Como podemos ver en el código anterior, al definir la clase `GenericClass` hemos indicado que ésta recibe un **parámetro de tipo** llamado `T`. Esto lo hemos indicado usando el operador diamante (*diamond operator*), `<>`. De esta forma podemos reutilizar la misma clase para diferentes clases de entrada. Así pues, gracias a las clases e interfaces genéricas podemos abstraernos de los tipos.

A la hora de instanciar la clase `GenericClass` podremos hacer, por ejemplo:

```

GenericClass<Animal> gc1 = new GenericClass<Animal>();
GenericClass<Person> gc2 = new GenericClass<Person>();

```

En el primer caso, el atributo privado `element` de la clase `GenericClass` será del tipo `Animal` (es decir, `T` es `Animal` en la primera instancia) y en el segundo caso será del tipo `Person`.

Desde la versión 7 de Java, no es necesario indicar la clase al llamar al constructor. Es decir, el código anterior se puede reescribir así:

```
GenericClass<Animal> gc1 = new GenericClass<>();
GenericClass<Person> gc2 = new GenericClass<>();
```

Gracias a JDK 7 se reduce la verbosidad ya que `<>` (llamado *diamond operator*) infiere el parámetro de tipo a partir del especificado como tipo para el atributo o variable que se está instanciando.

4.14.3. Ventajas de *generics*

Llegados a este punto, aunque sólo hemos hablado de clases genéricas, vamos ver qué ventajas tiene el uso de *generics*:

- **Comprueba los tipos en tiempo de compilación.** El compilador comprueba que el tipo con el que se instancia la clase genérica no viola la seguridad de tipos. Si ocurre, el compilador lanzará un error en tiempo de compilación, lo cual es mucho mejor que en tiempo de ejecución.

```
GenericClass<Integer> gc1 = new GenericClass<Double>(36.5); //error de compilación
```

- **Elimina la necesidad de hacer *casting*.** Debido a que durante la instanciación indicamos el tipo de la clase que hace de parámetro en la clase genérica, no es necesario hacer *casting* cuando recuperamos el objeto. Esto, además, mejora la legibilidad de nuestro código.

```
ArrayList<String> lista = new ArrayList<String>(); //ArrayList es genérica
lista.add("Hola!");
String s = lista.get(0); //no es necesario hacer cast: (String) lista.get(0);
```

- **Facilita a los programadores crear clases (y sobre todo, algoritmos) que sean genéricas.** Esto ayuda a reutilizar una gran cantidad de código en diferentes programas. Esta ventaja queda de manifiesto claramente en el uso de clases proporcionadas por Java que conforman el conjunto de colecciones (ver apartado 4.15).

Veamos ahora la ventaja de usar una clase genérica en vez de hacer uso de la clase `Object` para la misma finalidad.

<pre>public class A{ private Object obj; public void set(Object obj){ this.obj = obj; } public Object get(){ return obj; } }</pre>	<pre>public class A <T>{ private T obj; public void set(T obj){ this.obj = obj; } public T get(){ return obj; } }</pre>
--	---

Los dos códigos anteriores son dos maneras de codificar la clase `A`. En el de la izquierda usamos la clase raíz `Object` de la que heredan todas las clases. En cambio, en el código de la derecha hemos convertido la clase `A` en una clase genérica.

A priori puede parecer que son el “mismo” código, pero hay diferencias significativas en el uso. En el código anterior, de hecho, no hay muchas ventajas de usar uno u otro código. La mayor ventaja es que la clase de la izquierda puede recibir cualquier clase y devolver cualquier clase porque trabajamos siempre con Object, en cambio, el código de la derecha permite definir con qué clase trabaja la clase genérica y, entonces, el compilador puede detectar errores y que la clase A de la derecha no será usada con clases que no hemos definido al instanciarla.

<pre>A objA = new A(); objA.set("hola!"); Integer a = objA.get(); //error en ejecución objA.set(50); //OK</pre>	<pre>A objA = new A<String>(); objA.set("hola!"); Integer a = objA.get(); //error //compilación objA.set(3); //error compilación</pre>
---	--

Por otro lado, ¿qué pasaría si tuvierámos un método en la clase A que hiciera uso del atributo obj para llamar a uno de sus métodos?

<pre>public class A{ private Object obj; public void set(Object obj){ this.obj = obj; } public Object get(){ return obj; } public void greeting(){ (Dog) obj.greeting(); } }</pre>	<pre>public class A <T>{ private T obj; public void set(T obj){ this.obj = obj; } public T get(){ return obj; } public void greeting(){ obj.greeting(); } }</pre>
---	--

Como podemos ver, en el código de la izquierda aplicando polimorfismo y un *downcasting*, el atributo obj puede ser cualquier clase. Sin embargo, no podemos garantizar la seguridad de tipo porque necesitamos convertir explícitamente de Object a otra clase y, ya sabemos, que al hacer un *downcasting* es frecuente crear accidentalmente errores en tiempo de ejecución. Asimismo, ¿qué ocurre si la clase A también debe servir para la clase Person que también tiene el método greeting pero no está relacionado con la clase Dog? Pues que el hecho de tener que hacer un casting explícito, no nos permite generalizar el uso de la clase A. Si la clase A la usáramos exclusivamente para Dog y Cat, entonces con hacer un *downcasting* a Animal, sería suficiente y el código de la izquierda funcionaría cuando obj “fuera” Dog o Cat.

4.14.4. Interfaz genérica

De manera similar a una clase, también podemos definir nuestras propias interfaces genéricas.

<pre>public interface GenericInterface <T>{ void doSomething(T element); }</pre>

Una interfaz de Java que es genérica es Comparable.

4.14.5. Método genérico

Los métodos genéricos son similares conceptualmente a la clase genérica, pero el parámetro de tipo sólo afecta al método, no a toda la clase. Un método genérico puede estar tanto en una clase o interfaz genérica como normal. El primer caso lo hemos visto en el subapartado anterior, cuando hemos definido el método doSomething en la interfaz GenericInterface. Ahora veamos la definición de un método genérico en una clase normal.

```
public class A{

    public <T> boolean findItem(T[] list, T item){

        for(T element : list){

            if(element==item) return true;

        }

        return false;
    }
}
```

Como podemos ver, antes de la declaración del tipo de retorno debemos poner los parámetros de tipo dentro de <>.

Para llamar a este método que encuentra un elemento dado en un array dado, podemos hacer:

```
A objA = new A();

String names = {"David", "Elena", "Marina", "Pau"};

System.out.println(a.<String>findItem(names, "Elena")); //true

System.out.println(a.findItem(names, "Elena")); //true
```

Al método generic se le puede llamar de dos formas diferentes: la primera es indicando el argumento de tipo y otra es sin indicarlo (dejando que el compilador lo infiera).

Veamos dos ejemplos más:

```
public class B{

    public static <T> int countNumItems(T[] list){

        return list.length;
    }

    public <T> T getLastItem(T[] list){

        return list[list.length-1];
    }
}
```

}

4.14.6. Parámetros de tipo

En primer lugar, es importante resaltar que podemos tener uno o más parámetros de tipo. Por ejemplo:

```
public class GenericClass <T,S>{

    private T field1;

    private S field2;

    //TODO

}
```

Asimismo cabe indicar que, por convenio, el nombre del parámetro de tipo suele ser un único carácter escrito en mayúsculas. Los nombres más habituales son:

- E: usado para elementos de una colección (ver apartado 4.15)
- T: tipo (i.e. una clase, una interfaz o un tipo parametrizado).
- S, U, V, etc., para segundo, tercer, cuarto tipo, etc. En una clase genérica podemos tener más de un parámetro de tipo.
- K: clave (para clases que modelan un clave-valor)
- V: valor.

Por otro lado, los tipos que pueden utilizarse como parámetros de tipo sólo pueden ser clases o interfaces. Así pues, si queremos usar tipos primitivos, no podemos. La solución será usar las correspondientes clases *wrapper* (envoltorio). Éstas son:

- Para el tipo primitivo byte, su clase *wrapper* es Byte.
- Para el tipo primitivo short, su clase *wrapper* es Short.
- Para el tipo primitivo int, su clase *wrapper* es Integer.
- Para el tipo primitivo long, su clase *wrapper* es Long.
- Para el tipo primitivo float, su clase *wrapper* es Float.
- Para el tipo primitivo double, su clase *wrapper* es Double.
- Para el tipo primitivo char, su clase *wrapper* es Character.
- Para el tipo primitivo boolean, su clase *wrapper* es Boolean.

4.14.7. Subtipos

Podemos tener un subtipo de una clase o interfaz genérica heredando de ella o implementándola, respectivamente:

```
public interface MyList<E,T> extends List<E>{

    //TODO

}
```

Con el código anterior estamos creando una interfaz llamada `MyList` que hereda de la interfaz de Java llamada `List` (hablaremos de ella en el siguiente apartado). Si `List` fuera `List<Integer>`, entonces nuestra interfaz `MyList` podría ser

MyList<Integer, Object>,
 MyList<Integer, Double>, MyList<Integer, String>, etc.

4.14.8. Usando tipos parametrizados como parámetro de tipo

Un parámetro de tipo puede ser una tipo parametrizado. Por ejemplo:

```
public class A<T,V>{
    public void doSomething(T obj1, V obj2){
        //TODO
    }
}

A<String, List<String>> lista = new A<>(); //List es una interfaz genérica
```

4.14.9. Uso del wildcard ?

Con los *generics* podemos usar el comodín (en inglés, *wildcard*) ?. Este comodín representa un tipo desconocido o, dicho de otro modo, <<cualquier tipo>>. Este comodín puede ser usado como parámetro de tipo, tipo de un atributo o incluso como tipo del retorno de un método (aunque esto último no es aconsejable). Sin embargo, ? no puede ser usado para llamar a métodos genéricos ni para instanciar objetos de clases genéricas.

```
public class A{
    public void areTheSame(? obj1, ? obj2){
        return obj1.equals(obj2);
    }
}
```

Otro ejemplo podría ser:

```
public class A{
    public void getSize(?[] list){
        return list.length;
    }
}
```

Así pues, el uso del comodín ? sería:

- Cuando podríamos usar en su lugar Object.
- Cuando los métodos en los que lo utilizamos no dependen del parámetro de tipo, p.ej. list.length, etc.

4.14.10. Limitación de tipos

Hasta ahora hemos visto que los parámetros de tipo podían ser sustituidos por cualquier clase o interfaz. Sin embargo, esto a veces no es lo deseado y queremos limitar los tipos que se pueden pasar a un parámetro de tipo.

Un ejemplo claro es si queremos que los tipos que se reciban sean numéricos. Es decir, Integer, Float o Double, pero no String ni otro tipo. Si tuviéramos el siguiente código, éste no funcionaría:

```
public class Maths <T>{
    public T add(T t1, T t2){
        return t1 + t2;
    }
}
```

```
}
```

Con el código anterior el compilador nos daría error porque no sabe que nuestra intención es usar la clase Maths sólo con tipos numéricos y que, por consiguiente, sumar dos elementos es correcto. Así pues, debemos indicarle que sólo usaremos clases que sean numéricas. Es decir, limitaremos los tipos que puede aceptar el parámetro de tipo T. Esto lo haremos con los *bounded types* (i.e. tipos limitados). El código anterior usando *bounded types* sería:

```
public class Maths <T extends Number>{ // podría haberse sustituido T por ?
    public T add(T t1, T t2){
        return t1 + t2;
    }
    //TODO
}
```

Como podemos apreciar, hemos usado la palabra reservada `extends` para decir que todos los tipos de que reciba `T` deben heredar de la clase `Number`. En este caso, tanto `Integer` como `Float` y `Double` heredan de esta clase. Ahora podremos hacer:

```
Maths m1 = new Maths<Integer>();
System.out.println(m1.add(5,6)); //11

Maths m2 = new Maths<Double>();
System.out.println(m2.add(5.5,6.3)); //11.8

Maths m3 = new Maths<String>(); //error de compilación
Maths m4 = new Maths<Person>(); //error de compilación
```

Otro ejemplo donde indicamos que los tipos pasados al parámetro de tipo cumplen una condición es el siguiente:

```
public class C{  
  
    public static <T extends Comparable> T findSmallestItem(T[] list, T item){  
  
        for(item == null || list.length == 0){  
  
            return null;  
  
        }  
  
        T smallest = list[0];  
  
        for(int i = 0; i<list.length; i++){  
  
            if(smallest.compareTo(list[i])>0){  
  
                smallest = list[i];  
  
            }  
        }  
    }  
}
```

}

Gracias al uso de `extends Comparable` le estamos diciendo que todos los tipos que recibe el parámetro de tipo T debe implementar Comparable.

```
String names = {"David", "Elena", "Marina", "Pau"};  
  
System.out.println(C.findSmallestItem(names)); //David; La clase String implementa Comparable  
  
Person[] people = {new Person("David"), new Person("Elena")};  
  
System.out.println(C.findSmallestItem(names)); //error; La clase Person no implementa Comparable, si la implementara entonces funcionaría correctamente.
```

Podemos indicar más de una restricción para un mismo parámetro de tipo usando &:

```
public class Maths <T extends Number & Comparable>{  
    //TODO  
}
```

El uso de `extends` no permite indicar una acotación (o límite) superior. De ahí que a la combinación de `extends` seguido de una clase se le llame *upper bound*. Si en vez de usar un nombre, p.ej. T, se usa el comodín ?, entonces al conjunto se le llama *upper bound wildcard*. En cualquier caso, el uso de `extends` lo que nos permite es decir que lo que está a la izquierda del `extends` debe ser un tipo que obligatoriamente tiene que ser un subtipo de la clase/interfaz que escribamos a la derecha o la misma clase/interfaz. Así pues:

```
public class Maths <T extends Number>{  
    //TODO  
}
```

El código anterior nos está diciendo que lo que pongamos en T debe ser o `Number` o una subclase de `Number` (i.e. una clase que herede directa o indirectamente de `Number`).

Veamos otro ejemplo usando `List`. Entre el siguiente código:

```
public void doSomething(List<? extends Number> elements){  
    //TODO  
}
```

Y éste:

```
public void doSomething(List<Number> elements){  
    //TODO  
}
```

¿Ves la diferencia? Con el primero `elements` podría ser cualquier `List`, es decir: `List<Integer>`, `List<Double>`, etc., mientras que con el segundo código sólo podríamos pasarle como argumentos un ser `List<Number>`. Aunque `Integer` es un subtipo de `Number`, `List<Integer>` y `List<Number>` no están relacionados, ya que

`List<Integer>` no es un subtipo de `List<Number>`. El el elemento común/padre de `List<Integer>` y `List<Number>` es `List<?>`.

Existe otro limitador llamado `super`. Con él estamos diciendo que el tipo que se utiliza (lado izquierdo) debe ser una superclase o superinterfaz del elemento indicado a la derecha o el propio elemento. Es decir:

```
<? super Animal>
```

Nos está diciendo que lo que usemos (?) debe ser o `Animal` o una superclase de `Animal`.

Llegados a este punto, nos enfrentamos a la gran pregunta: ¿cuándo usar `extends`, `super` o ninguno de ellos?

- Usaremos `extends` cuando sólo queramos consultar los valores (i.e. acceso de lectura).
- Usaremos `super` cuando sólo queramos escribir/añadir valores (i.e. acceso de escritura).
- No usaremos ni `extends` ni `super` si necesitamos tanto leer como escribir.

4.14.11. Restricciones en el uso de generics

Los `generics` tienen algunas restricciones:

- No se pueden instanciar tipos genéricos con tipos primitivos.

```
ArrayList<int> list = new ArrayList<int>(); //KO: ArrayList es una clase generic
ArrayList<Integer> list = new ArrayList<Integer>(); //OK
```

- No se pueden crear instancias de los parámetros de tipo.

```
public class A<T>{
    public A(){
        new T(); //KO
        T lista = new T[10]; //KO
    }
}
```

- No se pueden declarar atributos `static` cuyos tipos son parámetros de tipo.

```
public class A<T>{
    private static T field1; //KO
}
```

- No se pueden hacer *castings* con tipos parametrizados:

```
List<Integer> lista = new ArrayList<>();
List<Number> lista2 = (List<Number>) lista; //KO
```

- No se puede usar `instanceof` con tipos parametrizados.

```
public <E> void doSomething(List<E> list){
    if(list instanceof ArrayList<Integer>){ //KO
    }
}
```

- No se pueden crear *arrays* de tipos parametrizados.

```
ArrayList<Integer> [] elements = new ArrayList<Integer>[10]; //KO
```

- No se pueden crear, capturar o lanzar tipos parametrizados que extiendan de `Throwable`.

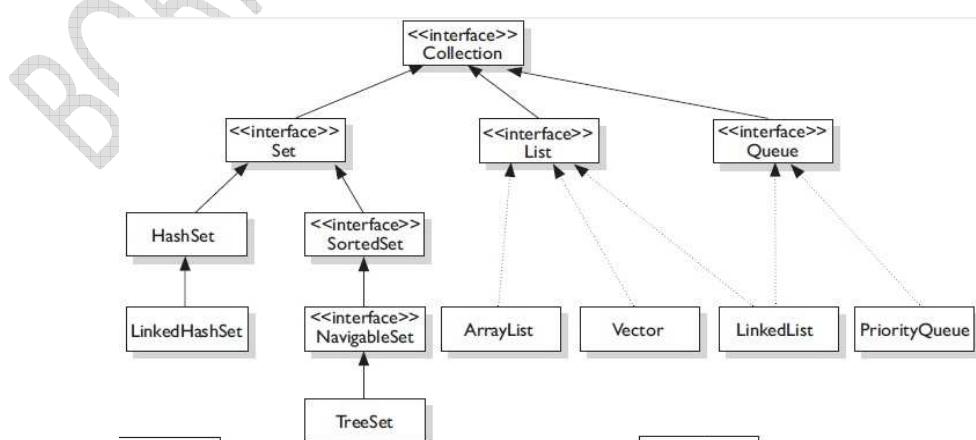
```
Public class MyOwnException<T> extends Exception{} //KO: Exception  
//hereda de Throwable
```

- No se puede sobrecargar un método que tengan la misma firma que otro después del *type erasure* (es el proceso que sigue Java a la hora de convertir el código a bytecode).

```
public class A{
    public void do(List<T> tList){}
    public void print(List<Integer> intList){}
    public void do(List<V> vList{}) //KO: los 3 métodos son el mismo
}
```

4.15. Colecciones

En Java existen un conjunto de **clases genéricas/parametrizadas** que se comportan como una estructura de datos que guardan objetos de cualquier clase. En este sentido, un subconjunto dentro del conjunto de clases que modelan una estructura de datos son las clases que implementan la interfaz `Collection`. Puedes ver una parte del diagrama de clases a continuación (siendo muy estrictos, decir que los triángulos tendrían que ser blancos).



Todas las clases representadas eliminan dos restricciones del *array*: (1) indicar obligatoriamente el número de ítems (i.e. casillas) que tendrá justo cuando se crea/inicializa, y

(2) no poder ampliar o reducir el número de casillas (i.e. elementos a guardar) una vez creado el *array*.

Como se puede ver en el diagrama, hay tres interfaces (*Set*, *List* y *Queue*) que heredan de la interfaz *Collection*. Las clases que implementan estas tres interfaces permiten añadir un número “infinito” de elementos (i.e. el asterisco de los diagramas de clases UML).

4.15.1. Interfaz Set

La primera interfaz que trataremos será *Set*. **Cada una de las clases que implementan *Set* se comporta como una estructura de datos que guarda un conjunto de elementos (en POO, objetos) únicos. Es decir las clases que implementan *Set* no permiten elementos duplicados.** No obstante, las diferentes clases que implementan *Set* añaden funcionalidades y comportamientos. Por ejemplo:

- **Ordenación:** *HashSet* guarda los elementos sin seguir un orden lógico, mientras que *LinkedHashSet*, guarda los elementos en el orden de inserción de los mismos y *TreeSet* los guarda siguiendo, o bien las indicaciones que codificamos en el método *compareTo()* (para usar este método, la clase del objeto que guardamos debe implementar la interfaz *Comparable* y sobrescribir dicho método), o bien mediante un objeto de una clase que implemente la interfaz *Comparator* y, en consecuencia, sobrescriba el método *compare* (dicho objeto se puede pasar como argumento en uno de los constructores de *TreeSet*).
- **Elemento null:** *TreeSet* no permite insertar el elemento *null*, mientras que *HashSet* y *LinkedHashSet* sí que permiten el elemento *null*, pero sólo una vez, puesto que la interfaz *Set* no permite elementos duplicados.

En líneas generales, de las tres clases se suele usar *HashSet*, puesto que es mejor en rendimiento, a no ser que realmente se necesite guardar elementos únicos (i.e. sin duplicados) en un orden específico. En este caso, se utiliza *LinkedHashSet* o *TreeSet*.

4.15.2. Interfaz List

Cómo puedes ver en el diagrama UML anterior, otra rama es la que empieza con la interfaz *List*. Como se puede apreciar, hay tres clases (en realidad hay más) que implementan *List*: *ArrayList*, *Vector* y *LinkedList*. Estas clases, las más usadas de *List*, tienen en común que se comportan como una lista (de aquí que implementen la interfaz *List*). Por lo tanto, **las tres clases permiten insertar objetos duplicados y, además, los objetos insertados están ordenados por índice**. Tanto *ArrayList* como *LinkedList* no son sincronizados. Esto quiere decir que no pueden ser usados directamente en programas con más de un hilo de ejecución (i.e. más de un *thread*, *multi-threading*). Para usarlos se tienen que sincronizar externamente. El concepto de *thread* lo estudiarás en la asignatura de Sistemas Operativos y también en Sistemas Distribuidos, así que por ahora no te preocupes. Por su parte, *Vector* es similar a *ArrayList*, excepto que es sincronizado. Así pues, se prefiere usar *ArrayList* cuando no se programa con múltiples hilos.

Tal y como están implementados `ArrayList` y `LinkedList`, los requisitos de memoria son menores para `ArrayList`. De hecho, `ArrayList` está basado en el concepto de *array dinámico (dynamic array)*, mientras que `LinkedList` en el concepto de *doubly linked list*.

Si se hacen muchas inserciones (i.e. `add`), especialmente en el medio de la lista, es preferible utilizar `LinkedList` que `ArrayList`, pues el coste es menor. En contra, si las inserciones se hacen al final, entonces `ArrayList` tiene un rendimiento un poco mejor, salvo que se pase de la capacidad inicial muy a menudo y se tenga que aumentar el *array* que lo implementa muchas veces.

Por otro lado, si se hacen más consultas de acceso aleatorio (i.e. `get`) que inserciones, es preferible usar `ArrayList`, ya que `LinkedList` recorre toda la lista hasta alcanzar la posición/índice indicado. El borrado, sobre todo si se hace al comienzo o en el medio de la lista, tiene un mejor coste en `LinkedList`, puesto que si se elimina un objeto del medio, no se deben mover todos los objetos posteriores como sucedería con la `ArrayList`.

Porque `LinkedList` también implementa la interfaz `Queue`, incluye métodos propios de las colas, como por ejemplo `peek` (obtener el primer elemento de la lista) o `poll` (obtener el primer elemento de la lista y borrarlo de la lista).

Finalmente llegamos a la rama de la interfaz `Queue` (en español, cola), cuyas clases siguen la filosofía FIFO (*First-In, First-Out*). De hecho, `LinkedList` implementa `Queue` además de la interfaz `List`, teniendo un comportamiento híbrido. La otra clase que implementa `Queue` es `PriorityQueue` que se trata de una cola con prioridad.

4.15.3. Interfaz Map

Lo que hemos visto hasta ahora son estructuras que, más o menos, te son conocidas, al menos, conceptualmente: lista, cola y conjunto (*set*). Nos ha faltado ver la pila, una estructura que también conoces y que sigue la filosofía LIFO (*Last-In, First-Out*) y que en Java está codificada con la clase `Stack` que hereda de la clase `Vector`.

Ahora queremos animarte a conocer unas nuevas estructuras de datos, muy frecuentes en el mundo de la programación, que se basan en el concepto de *array asociativo*. Un *array* (o matriz) *asociativo* (también conocido como *tabla hash* o *tabla de dispersión*) es una estructura de datos muy útil que consiste en un conjunto de pares (clave, valor). Muchos lenguajes de programación, como PHP y JavaScript, implementan la funcionalidad para trabajar con estas estructuras. Ponemos el ejemplo fácil de JavaScript:

```
var items = []; //declaramos un array, no entraremos en detalles
items["9781412902243"] = "Terracota Warriors";
items["9781412902244"] = "Venus de Milo";
items["9781412902245"] = "David by Donatello";
items["9781412902246"] = "The thinker by Rodin";
```

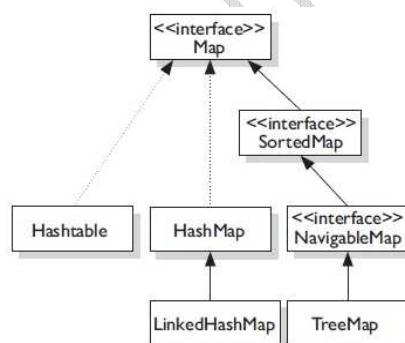
Si analizamos el código JavaScript anterior, lo que hacemos es crear un *array* (sin longitud, esto lo permite hacer JavaScript) y a cada casilla en vez de utilizar los índices 0, 1, 2 y 3, usamos una clave, en este caso de tipo textual (*String*). Ahora ya no usamos 0, 1, 2 y 3 para acceder a las casillas del *array*, sino que usamos claves. Así pues, en la casilla "9781412902243" (que no

quiere decir que sea ni la casilla 0 ni la casilla ubicada en la posición 9781412902243 del array) está el valor "Terracota Warriors". Por lo tanto, estamos usando una estructura basada en un *par clave-valor* (en inglés, *key-value pair*). Seguramente te estás preguntando, *¿para qué puede ser útil un array asociativo?* Pues, por ejemplo, imagina que la clave es el nombre corto utilizado para cada uno de los equipos (Team) que tenemos y dentro de la casilla correspondiente guardamos un objeto de la clase Team. Gracias al *array asociativo* seremos capaces de acceder de manera directa a todo el objeto Team sabiendo su nombre corto único. Si usamos otro tipo de estructura, p.ej. una lista, para poder encontrar el ítem con identificador "Barça", tendríamos que hacer una búsqueda por toda la estructura hasta encontrarlo. Eso sí, quizás te lo estás planteando, pero cada objeto guardado en el *array asociativo* tiene que tener una **clave única**, si no, sólo podremos guardar uno de los dos objetos como valor de aquella clave. Por ejemplo:

```
items[ "9781412902243" ]= "Terracota Warriors";
items[ "9781412902243" ]= "Venus de Milo";
```

En este caso el par clave-valor guardado será "9781412902243" – "Venus de Milo", es decir, en la casilla "9781412902243" no estará "Terracota Warriors".

En Java las clases asociativas se modelan, principalmente, con las clases que implementan la interfaz Map.



Como puedes ver en el diagrama, básicamente se usan cuatro clases: HashMap, Hashtable, LinkedHashMap y TreeMap, que tienen un comportamiento similar al que hemos comentado, con sus particularidades. En cualquier caso la clave tiene que ser un objeto (ya sea de una clase *wrapper* si queremos usar los tipos básicos, p.ej. Integer en vez de int, String, Float en vez de float, etc.; o una clase hecha por nosotros).

Ni HashMap ni Hashtable guardan los elementos ordenados, ni por clave ni por valor. Por el contrario, los elementos dentro de TreeMap están ordenados por clave usando el método compare de la interfaz Comparator y en LinkedHashMap se ordenan por orden de inserción.

Hashtable es como HashMap pero sincronizado. Esto quiere decir que puede usarse en programas con más de un hilo de ejecución. Obviamente esta característica le comporta un sobrecoste al programa.

Otra diferencia es que Hashtable no permite valores ni claves null, mientras que HashMap sí que permite valores null, y sólo una clave null. A la vez, LinkedHashMap sí

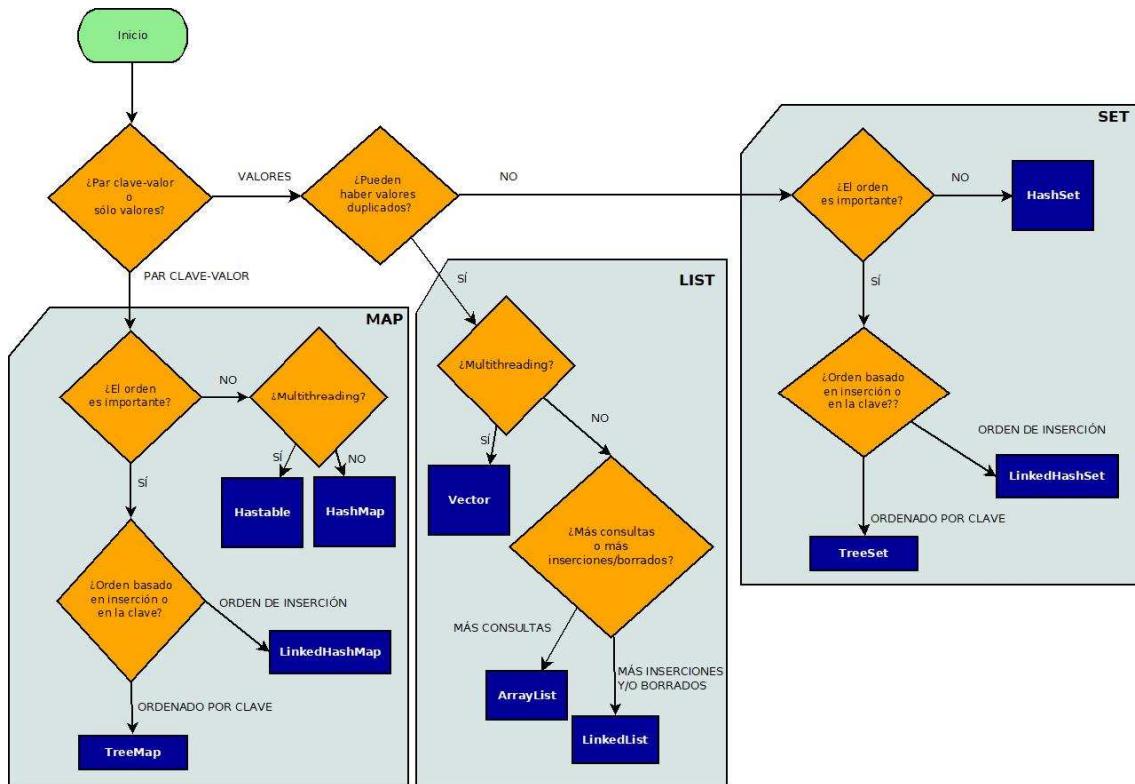
que permite tener duplicados, del mismo modo en que lo hace `HashMap`, pues hereda de esta clase. Por su parte, `TreeMap` sólo permite valores `null`, no claves.

4.15.4. Resumen

Con la explicación de las clases que implementan la interfaz `Map`, parece que todo son ventajas y lo mejor es usar una estructura de datos de tipo `Map`, en vez de tipos `List` o `Set`. Bien, pues evidentemente, cada clase tiene su uso. Por ejemplo:

- Las clases procedentes de `List`, como por ejemplo `ArrayList`, son útiles si necesitamos guardar el orden de los objetos que guardamos.
- Además, las clases de tipo `List` consumen menos memoria, pues sólo guardan el objeto que insertamos, mientras las que son de tipo `Map`, guardan dos objetos: la clave y el valor.
- Las clases de tipo `List` pueden guardar duplicados de valor, mientras que las de tipo `Map` sólo pueden duplicar valores, no claves. Las clases de tipo `Set` no permiten duplicados.
- Las clases de tipo `List` y `Map` permiten el valor `null`. Eso sí, las de tipo `Map` sólo permiten `null` una vez como clave.
- A diferencia de las clases de `List` y `Map`, las clases de `Set` no tienen un método para acceder directamente a un elemento guardado (conocido como *acceso aleatorio*), sino que se debe iterar sobre toda la colección hasta llegar al elemento deseado.

En la siguiente imagen te facilitamos un diagrama de decisión que te ayudará a la hora de escoger una estructura u otra. Es un diagrama genérico y, por ello, será el problema a tratar así como tu conocimiento de las estructuras lo que finalmente acaben determinando tu decisión.



5. Avanzado

5.1. La clase String

A diferencia de lenguajes de programación como C o C++ donde un *string* es un *array* de *char*, el *String* en Java, como ya sabes, una clase que hereda, como cualquier otra clase en Java, de la clase *Object*. Sin embargo, la clase *String* es especial respecto al resto de clases porque:

- Puedes asignar un texto con dobles comillas (*String literal*) a una variable de tipo *String* sin necesidad de llamar al constructor para crear una instancia de la clase *String*.

```
String text = "Hello!";
```

Así pues, un objeto *String* puede crearse como si fuera un tipo primitivo (p.ej. como hace un entero: `int a = 5;`) o como un objeto:

```
String s = new String("Hi!");
```

Cabe decir que lo más habitual es construirlo como si se tratara de un tipo primitivo, lo que se denomina *String literal* para diferenciarlo de la segunda forma llamada *String object*.

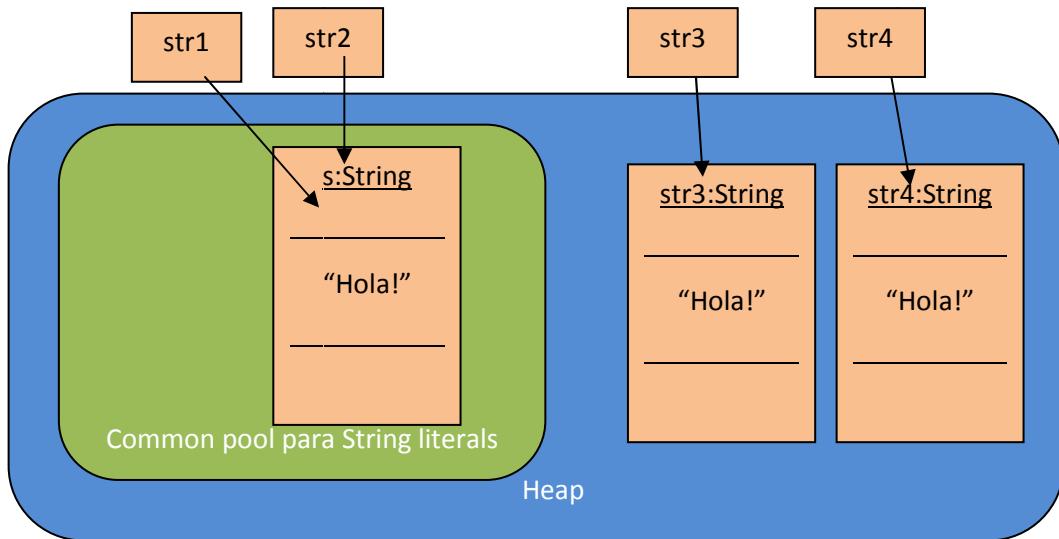
- El operador + está sobrecargado para poder concatenar dos *String*. Este operador no funciona con objetos de otras clases. Así pues:

```
String str1 = "Hola" + " David!";
```

- Los *String literals* son almacenados como objetos en un lugar común llamado *String common pool* dentro del *heap*. Esto facilita la compartición y eficiencia del almacenaje. Por su parte, cuando el *String* es creado con la llamada explícita al constructor, se crea un objeto (al que llamaremos *String object* para distinguirlo del *String literal*) que se almacena directamente en el *heap*. Mira este código:

```
String str1 = "Hola!";
String str2 = "Hola!";
String str3 = new String("Hola!");
String str4 = new String("Hola!");
```

El código anterior, gráficamente, sería:



Así pues, si dos *String literals* tienen el mismo contenido, entonces apuntan y comparten el mismo espacio de memoria en el *String common pool*. Por su parte, cada uno de los *String* creado como objeto (i.e. con `new`) se comporta como cualquier objeto de otra clase ocupando su propio espacio en el *heap*, aunque el contenido de dos *String objects* sea el mismo.

- Debido a que los *String literals* con idéntico contenido comparten la zona de memoria en el *String common pool*, **Java define String como inmutable**. Es decir, el contenido no puede ser modificado una vez ha sido creado. Por este motivo cuando llamamos a un método como `toLowerCase()`, este método no modifica el *String* que le pasamos, sino que crea uno nuevo con todas las letras en minúsculas. Esto se hace así para que *String* se comporte como inmutable, pero esto puede crearnos problemas:

```
String str1 = "Hola";
str1.concat(" David!");
System.out.println(str1); //Imprime "Hola";
```

En el código anterior tendríamos en el *String common pool* 2 *String*, uno con el texto "Hola" y otro con el texto "Hola David!". El problema es que mientras `str1` apunta a "Hola", nadie está apuntando a "Hola David!", lo cual haría que estuviéramos malgastando memoria. Es verdad que Java tiene un mecanismo especial llamado *garbage collector* que se encarga de ir borrando periódica y automáticamente aquellos objetos sin referencia. Sin embargo, que se hagan excesivas llamadas al *garbage collector* afectará gravemente al rendimiento de tu programa. Para hacerlo “correctamente” (y sólo tener una zona ocupada en el *String common pool* y no dos) tendríamos que hacer:

```
String str1 = "Hola";
str1 = str1.concat(" David!");
System.out.println(str1); //Imprime "Hola David!"
```

Sin embargo, el anterior código no es correcto, ya que si modificamos el contenido de un *String literal*, estaremos afectando a todas las referencias que apuntan al mismo lugar del *String common pool* (es decir, al resto de *String literals* que hasta ese momento compartían contenido). Esto puede llevar a situaciones impredecibles e indeseables. Así pues, el siguiente código sería inadecuado e impredecible:

```
String str = "Hola ";
for(int i = 0; i<100; i++) str += 1;
```

Si el contenido de un String tiene que ser modificado frecuentemente, entonces es mejor usar las clases *StringBuilder* o *StringBuffer*, que son mutables (el objeto se crea en el *heap* general, y cualquier cambio sobre el objeto modificará el objeto, no creará uno nuevo), aunque no son tan potentes en cuanto a métodos como *String*.

5.1.1. *StringBuilder* vs *StringBuffer*

La principal diferencia entre estas dos clases es que *StringBuilder* no es sincronizada, mientras que *StringBuffer* sí lo es. Esto significa que podemos usar *StringBuffer* en programas con más de un hilo de ejecución (en inglés, *thread*), es decir, cuando estamos haciendo programación paralela. En el caso de esta asignatura, como no se va a dar esta situación, usaremos *StringBuilder* porque además su eficiencia, al ser no sincronizado, es mejor que *StringBuffer*.

Como ambas clases son clases normales, entonces sólo se pueden crear a partir de un constructor, no podemos asignarles directamente un *String literal*. Tampoco se puede usar el operador +, sino los métodos *append()* o *insert()*.

5.1.2. Resumen

Si tienes que modificar frecuentemente un texto, entonces utiliza *StringBuilder* (en programas *single thread*) o *StringBuffer* (en programas *multi-thread*). Si no, utiliza *String* porque es más eficiente, ya que le das al compilador la posibilidad de optimizar el uso de memoria por parte del programa mediante el uso del *String common pool*.

5.2. Programación funcional (expresiones lambda y Stream API)

La versión 8 de Java trajo consigo la programación funcional. De esta manera, Java puso de manifiesto que a partir de entonces no hay que pensar en este lenguaje como exclusivamente orientado a objetos. La programación funcional está ganando popularidad año tras año. Incluso hay tendencias que muestran que empieza a ser más popular que el orientado a objetos.

5.2.1. Expresiones lambda

La versión 8 de Java trajo consigo las expresiones lambda. Básicamente, una expresión lambda es una función anónima. Esto quiere decir que no tiene un nombre ni requiere ser un método de una clase y, por lo tanto, no necesita un nombre para ser invocada. Su sintaxis es la siguiente:

(parámetros) -> {cuerpo-expresión-lambda}

Como ves, hay tres partes:

- Operador lambda que está formado por los símbolos matemáticos menos y mayor que, i.e. `->`.
- La parte izquierda hace referencia a los parámetros de la función. Si sólo hay un parámetro, entonces no es necesario poner los paréntesis. En el caso de que no haya parámetros o haya más de uno, entonces sí que son obligatorios. Además Java puede deducir los tipos de los parámetros que se les pasa y, por lo tanto, no es necesario indicarlos.
- La parte derecha hace referencia al cuerpo, es decir, al código en sí de la función. Si el cuerpo sólo tiene una línea, entonces no son necesarias ni las llaves ni la sentencia `return` en caso de que devuelva un valor.

Algunos ejemplos de expresión lambda son:

```
a -> a + 2;
```

```
() -> System.out.println("Hola");
```

```
(int width, int height) -> {width * height};
```

```
(String x) -> {
    x = x.concat("****");
    return x; }
```

```
(a) -> {
    System.out.println(a);
    return true;
}
```

Las expresiones lambda se basan en la programación funcional, que es un paradigma dentro de la programación declarativa basada en el uso de funciones matemáticas. Esto permite una programación más expresiva y elegante. La programación declarativa nos permite decirle a un programa qué es lo que queremos/necesitamos, pero sin decirle cómo lo tiene que hacer/llevar a cabo. Ésta es la principal diferencia con la programación imperativa (a la que estás acostumbrado/a), en la que indicas todos los pasos a seguir para lograr lo que quieras (tu objetivo). Por ejemplo, SQL (usado para manipular bases de datos) es un lenguaje declarativo:

```
SELECT * from users where surname= "Anderson";
```

La sentencia anterior (en SQL llamada *query*) pide a la base de datos que retorne todos los usuarios de la tabla `users` cuyo apellido sea Anderson. En este caso no sabes si la base de datos internamente usará un `for` o un `while` o un `if` o un `switch` o un `array` u otra estructura. Por así decirlo, tú pides lo que quieras, el programa se “busca la vida” para satisfacerte.

5.2.2. Interfaz funcional

Las expresiones lambda van íntimamente ligadas con el concepto de **interfaz funcional**, concepto también fue añadido en Java 8. Una interfaz funcional es igual que una interfaz Java normal pero añade 2 reglas:

- Tiene un sólo método abstracto.
- El resto de métodos deben ser métodos estáticos o default.

Aunque es opcional, en la declaración de la interfaz funcional se puede añadir la anotación `@FunctionalInterface` para hacer el código más fácil de entender. Por ejemplo:

```
@FunctionalInterface
public interface Operacion{
    /*método abstracto que suma 2 números y cuyo cuerpo se
     *implementará mediante una expresión lambda*/
    public void suma(int a, int b);
}
```

A continuación utilizamos la interfaz funcional que hemos declarado:

```
public class Test{
    public static void main(String[] args){
        //Escribimos el código del método abstracto de la interfaz con
        //una expresión lambda
        Operacion op = (a,b) → {System.out.println(a+b);};
        //Usamos el método con la implementación
        op.suma(10,5);
    }
}
```

Así pues, las interfaces funcionales se pueden usar para crear un tipo concreto para ser usado en expresiones lambda.

Llegados a este punto puede parecer que muchas veces tengamos que crear una interfaz funcional si queremos hacer algo complejo. Como hemos visto, las expresiones lambda pueden trabajar con interfaces funcionales para lograr la reutilización de su código a lo largo de nuestro programa, pero muchas veces usaremos una expresión lambda en un contexto concreto una sola vez. De hecho, Java tiene muchos métodos en sus diferentes clases que aceptan como parámetro una expresión lambda. Por ejemplo, ahora las clases que implementan la interfaz `Iterable` tienen el método `forEach`. Algunas de estas clases son `ArrayList`, `LinkedList`, `HashSet`, etc. Este método lo que hace es recorrer (o iterar) una estructura de datos y para cada elemento de la estructura ejecutar la operación (que será una expresión lambda) pasada por parámetros. Así pues:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(1);
lista.add(2);
lista.add(3);
```

```
lista.forEach(elemento -> System.out.println(elemento));
```

El código anterior imprime los números que hay en lista.

Finalmente decir que las expresiones lambda se están utilizando mayoritariamente con la nueva API Stream ([https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java.util.stream/package-summary.html](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html)) que introdujo JDK 8. De manera resumida, podemos decir que a través del API Stream podemos trabajar sobre colecciones como si estuviéramos realizando sentencias SQL (gracias a las expresiones lambda). Esto permite que trabajemos de una manera limpia y clara, evitando bucles y algoritmos que ralentizan los programas e incluso hacen que el código se torne inmanejable.

Existen 3 partes que componen un Stream que, de manera general, serían:

- Un Stream funciona a partir de **una lista o colección**, que también se la conoce como la fuente de donde obtienen información.
- **Operaciones intermedias** como por ejemplo el método filter, que permite hacer una selección a partir de un predicado.
- **Operaciones terminales**, como por ejemplo los métodos max, min, sum, forEach, findFirst, etc.

Así pues, una posible sentencia con Stream sería:

```
ArrayList<Empleado> empleados = new ArrayList<Empleado>();  
  
empleados.stream().filter(empleado -> empleado.getSalary()>=1000).sum();
```

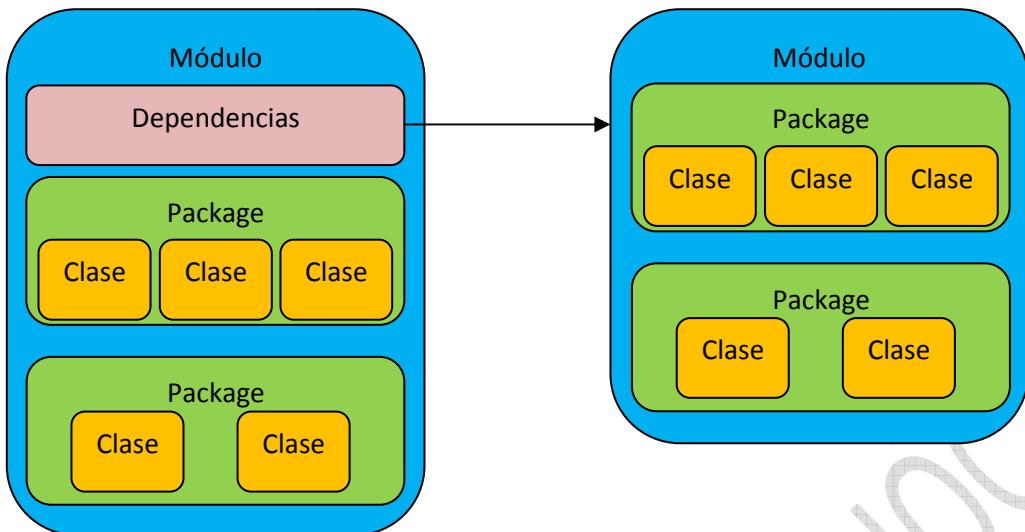
La expresión anterior filtra los empleados que cobran 1000 € o más y suma sus cantidades. Así sabemos cuál es el coste salarial de los empleados que son mileuristas.

5.3. Módulos

Suponemos que te habrás dado cuenta que en los proyectos Eclipse crea un fichero llamado module-info.java. El hecho es que JDK 9 introdujo el concepto de módulo para mejorar aún más la encapsulación. En Java, un módulo es:

Definición módulo

Un conjunto de clases que pueden contener uno o varios packages y que define las dependencias con el resto de módulos así como la visibilidad de las clases que contiene.



Para definir todo esto, los módulos usan **descriptores** que se escriben en un fichero llamado `module-info.java` que está ubicado en la raíz del código fuente del módulo. Fíjate que el nombre que tiene este fichero rompe la regla de nomenclatura de Java, la cual no permite poner un guión como nombre de fichero. De esta forma, las herramientas (`java`, `javac`, IDEs, etc.) no lo confunden con una clase Java normal.

Cada módulo definido en el fichero `module-info.java` tiene esta forma:

```
module nombre{}
```

Se recomienda que el nombre del módulo no coincida con el de una clase, interfaz o paquete a fin de evitar confusiones.

Dentro de las llaves podemos usar diferentes sentencias. Las dos más usadas son:

- `exports` indica que las clases públicas del paquete exportado son visibles para el resto del mundo.

```
module a{
    exports edu.uoc.logging;
}
```

Para este ejemplo sólo las clases públicas que están dentro del paquete `edu.uoc.logging` serán visibles fuera del módulo `a`. Así pues, las clases públicas de otro paquete del módulo `a`, por ejemplo, `edu.uoc.classroom`, no serán visibles fuera del módulo.

Por lo tanto vemos que, gracias a los módulos, aun siendo pública una clase en un paquete, ésta no lo será para los elementos de fuera del módulo si no se exporta explícitamente. Así pues, los módulos pueden ocultar visibilidad pública, lo cual mejora la encapsulación en Java.

- `requires` indica una dependencia a un módulo. Es decir, un módulo necesita o quiere usar otro módulo.

```
module b{
    requires a;
}
```

5.4. Clases anidadas

En primer lugar, vamos a definir qué es una clase anidada (en inglés, *nested class*).

Definición de clase anidada

Una clase anidada es aquella que está dentro de otra.

La posibilidad de crear clases anidadas se incorporó en Java 1.1. Aunque son un tema avanzado, son interesantes puesto que hacen referencia a aquellas clases que se utilizan en un punto concreto de nuestro programa. Además mejoran la encapsulación y la legibilidad del código.

Hay dos tipos de clases anidadas: (1) *static nested classes*, y las *inner classes*. Veamos cada una de ellas.

5.4.1. Static nested class

Veamos un ejemplo para entender el funcionamiento de este tipo de clase:

```
public class OuterClass{
    private static String name = "David";

    public static class StaticNestedClass{
        public void doSomething(){
            System.out.println("Hello! " + name); //Desde una clase
//anidada estática se pueden acceder a cualquier atributo estático de la clase
//externa.
        }
    }
}
```

La clase anidada estática tiene acceso a todos los miembros estáticos de la clase externa. Si desea acceder a miembros no estáticos, se deberá usar una referencia a un objeto de la clase externa. Por este motivo, las clases anidadas estáticas son poco utilizadas.

Desde la clase externa no se puede acceder a los miembros de la clase anidada estática. La clase anidada estática es un miembro de la clase externa y, por lo tanto, podemos asignarle el modificador de acceso que creamos oportuno.

Si quisieramos instanciar un objeto de la clase anidada estática, deberíamos hacer:

```
OuterClass.StaticNestedClass objeto = new OuterClass.StaticNestedClass();
```

5.4.2. Inner class

Una clase interna (*inner class*) es una clase anidada no estática. Su uso más habitual es proporcionar una clase que sólo se utiliza dentro de su clase externa. Se declara de la siguiente manera:

```
public class OuterClass{
    private static String name = "David";
```

```
public class InnerClass{  
}  
}
```

En este punto es importante destacar que una instancia de la clase interna sólo puede existir dentro de una instancia de la clase externa. Para crear una instancia de la clase interna, primero se debe crear el objeto de la clase externa:

```
OuterClass outerObject = new OuterClass();  
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Desde la clase interna se puede acceder a cualquier miembro de la clase externa.

```
public class OuterClass{  
    private static String name = "David";  
    private int age = 36;  
  
    public class InnerClass{  
        public void doSomething(){  
            System.out.println("Hello! "+name+". You are "+age);  
        }  
    }  
}
```

La clase interna es un miembro de la clase externa y, por lo tanto, podemos asignarle el modificador de acceso que creamos oportuno.

Dentro de la categoría *inner classes*, existen dos variantes: las clases internas locales y las clases internas anónimas. Dejamos en vuestras manos la profundización en estas dos variantes.

5.5.Anotaciones

Una anotación es un elemento que proporciona información extra (metadatos) sobre un elemento de nuestro código. Su presencia no tiene un efecto directo en la lógica de nuestro código.

Los usos más habituales de las anotaciones son:

- Proporcionar información al compilador: gracias al uso de una anotación, el compilador puede ayudarnos a detectar errores o a suprimir avisos (*warnings*) que ya conocemos.
- Algunas herramientas son capaces de procesar la información proporcionada por las anotaciones para generar código, ficheros, etc.
- Algunas anotaciones están disponibles en tiempo de ejecución.

El formato básico de una anotación es el símbolo @ seguido del nombre de la anotación, p.ej. @Override.

La anotación puede ser simplemente su nombre (sin elementos adicionales):

```
@Override

public void doSomething() {
    //TODO
}
```

o incluir elementos:

```
@SuppressWarnings{
    value = "unchecked"
}

public void doSomething() {
    //TODO
}
```

Podemos asignarle a un mismo elemento más de una anotación:

```
@SuppressWarnings{
    value = "unchecked"
}

@Override
public void doSomething() {
    //TODO
}
```

Java trae consigo un conjunto predefinido de anotaciones que están definidas en los *packages* `java.lang` y `java.lang.annotation`.

Anotación	Descripción
<code>@Deprecated</code>	Indica que el elemento (i.e. método, clase o atributo) marcado no debe ser usado nunca más. El compilador genera un aviso.
<code>@Override</code>	Avisa al compilador de que el método con dicha anotación es una sobreescritura de un método de una superclase. No es obligatorio usar esta anotación, pero sí nos ayuda a la hora de detectar errores, ya que el compilador verificará que se haga la sobreescritura correctamente.
<code>@SuppressWarnings</code>	Pide al compilador que no avise de según que <i>warnings</i> .
<code>@SafeVarargs</code>	Sólo se puede usar con métodos con <i>varargs</i> . Sirve para que no se realicen operaciones inseguras sobre el parámetro <i>varargs</i> .
<code>@FunctionalInterface</code>	Indica que la interfaz sea considerada funcional.

5.5.1. Creación de una anotación personalizada

Para crear nuestra propia anotación debemos crear un fichero .java con el nombre de la anotación y usar en él la siguiente sintaxis:

```
@interface nombreAnotacion{
    //Elementos
}
```

Un ejemplo:

```
@interface Article{
    String[] authors();
    String title();
    int currentVersion() default 1;
    String lastModified();
    String[] reviewers();
}
```

Como podemos ver:

- Usamos la anotación @interface para crear una anotación.
- Cada elemento es un método sin parámetros ni throws.
- Los tipos de retorno de los métodos son tipos primitivos, String, Class, enum, anotaciones y arrays de los tipos anteriores.
- Los métodosd pueden tener un valor por defecto. Lo indicamos con la palabra clave default.

En la creación de nuestra anotación podemos añadir algunas anotaciones que vienen en el package java.lang.annotation.

Anotación	Descripción
@Retention	Indica cuándo se puede acceder a la anotación.
@Documented	Indica que la información de la anotación debe ser añadida al generar documentación con javadoc.
@Target	Especifica el tipo de elemento al que se va a asociar la anotación.
@Inherited	Indica que la anotación será heredada automáticamente.
@Repeatable	Indica que la anotación puede utilizarse más de una vez para un mismo elemento.

Veamos un ejemplo:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface InfoMethod{
    String[] authors();
```

```

    int currentVersion() default 1;

    string lastModified();

    string[] reviewers();

}

```

La anotación @InfoMethod que acabamos de definir sólo se puede aplicar a métodos y se puede acceder a ella en tiempo de ejecución.

5.5.2. Uso de una anotación personalizada

Vamos a ver cómo usar la anotación @InfoMethod creada en el apartado anterior:

```

@InfoMethod(
    authors = {"David García", "Carlos Caballero"},
    lastModified = "03/03/2020",
    reviewers = {"Jordina", "Marta", "Sergio", "Romualdo", "Fran"}
)
public void doSomething(){
    //TODO
}

```

Debemos fijarnos que cuando usamos una anotación, los elementos de ésta van ubicados dentro de paréntesis.

5.6. Método requireNonNull

Con JDK 7 se añadió a la clase Object un método estático llamado requireNonNull que internamente realiza lo siguiente:

```

public static <T> T requireNonNull(T obj){
    if(obj == null)
        throw new NullPointerException();
    return obj;
}

```

Es decir, comprueba que el objeto pasado por parámetros no sea null. Si lo es, lanza una excepción de tipo NullPointerException. En caso de no ser null, devuelve el objeto en sí.

Existe una sobrecarga de este método que permite indicarle el mensaje de error a mostrar al lanzar la excepción de tipo NullPointerException.

Así pues el siguiente código:

```
public void doSomething(String name){  
  
    if(name == null)  
  
        throw new NullPointerException("Name cannot be null");  
  
    this.name = name;  
  
}
```

Puede ser simplificado usando requireNonNull:

```
public void doSomething(String name){  
  
    this.name = Object.requireNonNull("Name cannot be null");  
  
}
```

6. Extras

6.1. Javadoc

En los archivos Java se pueden poner comentarios en un formato especial que permite mediante el comando javadoc generar de forma automática documentación de los paquetes, clases, interfaces, atributos y métodos de nuestro proyecto. La sintaxis de Javadoc es:

```
/**
 * Texto de la descripción (se pueden añadir etiquetas HTML)
 *
 * A partir de aquí etiquetas Javadoc que comienzan por @
 *
 */
```

Los comentarios normales (i.e. `//` y `/* */`) Eclipse los colorea en **verde**. Por otro lado, los comentarios en formato Javadoc los colorea en **azul**. Así puedes distinguir visualmente entre comentarios normales y de Javadoc.

Para las clases se suele hacer una breve descripción de ésta y usar las etiquetas `@author` y `@version`. La primera indica el nombre del autor que ha realizado la clase (si hay más de uno, se pueden usar tantas etiquetas `@author` como autores). Por su parte, la segunda etiqueta informa de la versión de la clase (una clase puede verse modificada en el tiempo y, en un momento concreto, se puede considerar que es una nueva versión).

En el caso de los métodos (incluidos los constructores), la primera parte del comentario Javadoc es una descripción de lo que hace el método (de forma breve), y las siguientes líneas consisten en un conjunto de etiquetas Javadoc. Las etiquetas más usadas en los métodos son:

Etiqueta	Descripción
<code>@param</code>	Descripción de un parámetro. Cada parámetro tiene su etiqueta <code>@param</code> .
<code>@return</code>	Descripción de lo que devuelve el método, si el retorno no es <code>void</code> .
<code>@throws</code>	Descripción de la excepción que puede propagar. Habrá una etiqueta <code>throws</code> por cada tipo de excepción.
<code>@see</code>	Enlace a la documentación de otra clase, ya sea de Java, del mismo proyecto o de otro.
<code>@since</code>	Indica desde cuándo existe este método. Puede ser cualquier texto, pero normalmente es el número de versión de la clase.
<code>@deprecated</code>	Marca el método como obsoleto. Sólo se mantiene por compatibilidad y, por lo tanto, aún se puede usar. Es posible que en futuras versiones de la clase el método desaparezca definitivamente y no se pueda usar.

En el caso de los atributos de una clase, simplemente se pone su descripción con `/** */` justo antes de su declaración.

Una vez escritos los comentarios, existen varias maneras de crear la documentación. Vamos a ver dos de ellas:

6.1.1. Generar javadoc por línea de comandos

La otra forma de crear la documentación es por línea de comandos, es decir, sin usar ningún IDE como Eclipse. Para ello, tenemos que abrir una consola de comandos (cmd) y ubicarnos donde tengamos los ficheros .java. A partir de aquí podemos:

- a) Generar la documentación de una clase concreta (p.ej. Program.java):

```
>> javadoc Program.java
```

- b) O, lo más habitual, generar la documentación de todos los .java (i.e. clases) que haya en el directorio:

```
>> javadoc *.java
```

Es decir, de este modo generaremos la documentación de toda la aplicación (o programa).

En cualquiera de las dos versiones, podemos indicar que nos genere toda la documentación en un directorio específico (si no existe, lo crea) mediante -d nombreDirectorio:

```
>> javadoc *.java -d doc
```

Una vez ejecutado correctamente el comando javadoc, sólo tenemos que abrir el archivo index.html con un navegador web para ver la documentación.

Finalmente indicar que cuando se llama al comando javadoc por línea de comandos, sólo genera la documentación para los atributos y métodos que son públicos y protegidos. Si queremos que incluya también los privados, se lo debemos indicar así

```
>> javadoc *.java -d doc -private
```

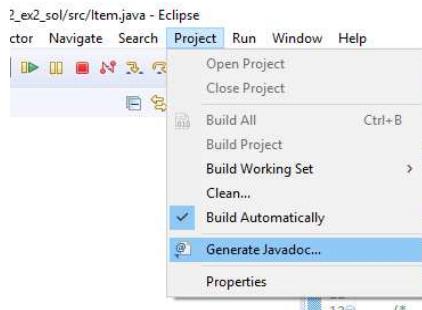
Dar información sobre los atributos y métodos privados es útil si estamos compartiendo información con alguien que debe programar aquella clase por dentro. En caso contrario, ¡¡no se deben dar detalles de los atributos y métodos privados!!! ¡¡¡Esta es la gracia de la encapsulación!!!

6.1.2. Generar javadoc con Eclipse

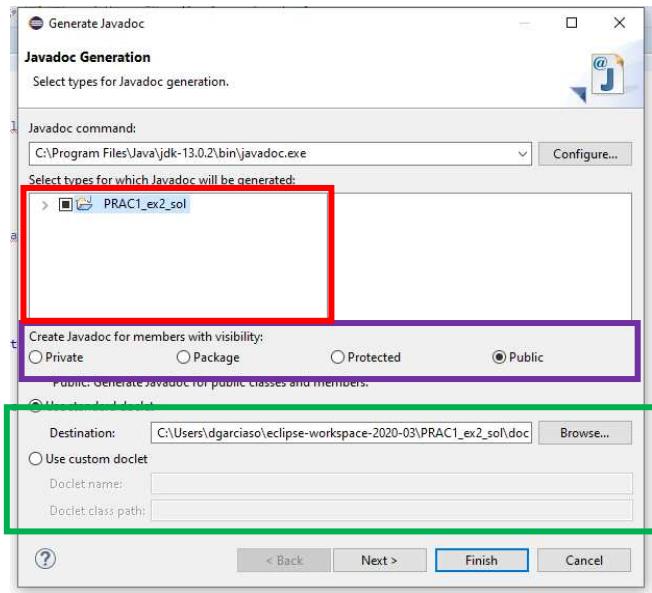
Para generar la documentación con javadoc en Eclipse debes seguir estos pasos:

Seleccionar el proyecto (si tenemos seleccionada una clase, sólo nos hará el Javadoc de esa clase).

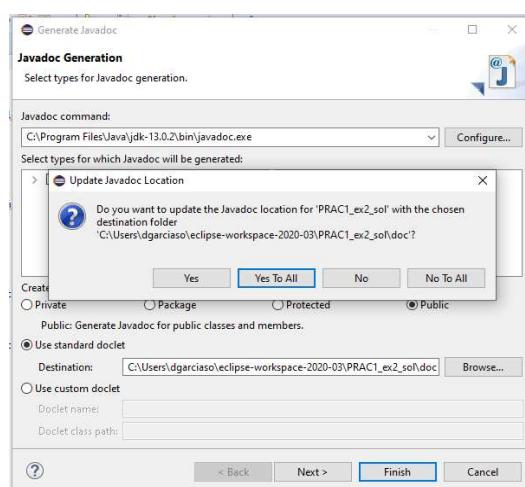
1. Ir al menú superior de Eclipse y escoger *Project → Generate Javadoc...*



2. Nos aparecerá una ventana como la que se muestra en la imagen siguiente.



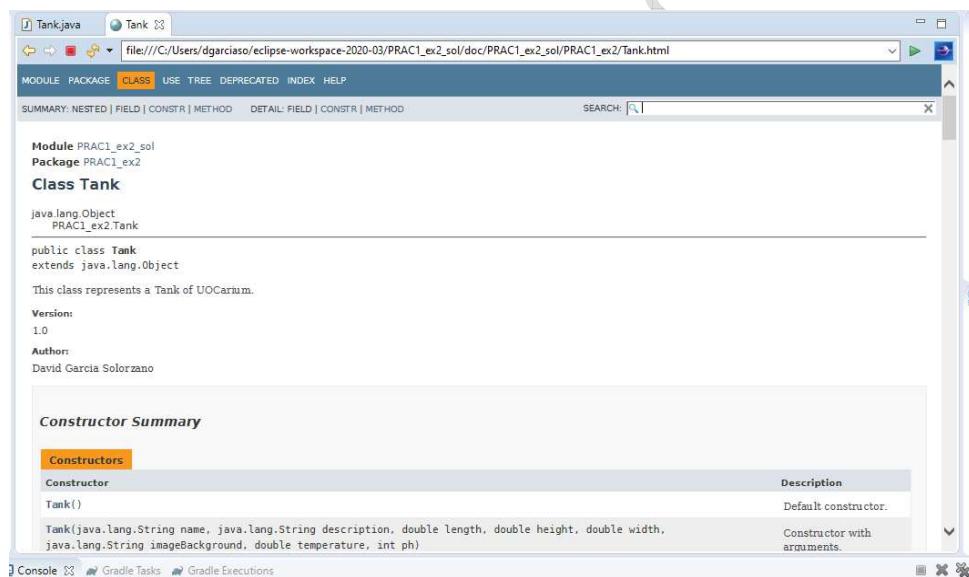
3. Si el campo *Javadoc command* está vacío, debemos indicar pulsando el botón *Configure...*, la ruta de nuestro javadoc.exe. Éste se encuentra en la carpeta de nuestro JDK, p.ej. C:\Program Files\Java\jdk-13.0.2\bin\javadoc.exe.
4. Indicamos para qué proyectos deseamos generar la documentación (ver recuadro rojo en la imagen anterior) y en qué carpeta queremos guardar la documentación generada (ver recuadro verde en la imagen anterior).
5. Eclipse permite indicar para qué atributos y métodos se genera la documentación (ver recuadro lila de la imagen anterior).
6. Una vez tenemos todo configurado, pulsamos el botón *Finish*. Obviamente podemos clicar en el botón *Next* para personalizar el comportamiento de javadoc.
7. Si dejamos la opción estándar, es posible que la primera vez nos aparezca la ventana de diálogo que se muestra en la siguiente imagen. Di *Yes To All*.



8. Si hacemos “F5” (o botón derecho “Refresh”) sobre el proyecto (no siempre hace falta), veremos que ha aparecido una nueva carpeta llamada doc. Esta carpeta contiene varios ficheros .html, que es el formato en que Javadoc genera la documentación.
9. Para ver correctamente la documentación generada, debemos ir a la carpeta doc en el propio Eclipse, hacer botón derecho sobre el fichero index.html y escoger la opción *Open With → Web browser*. En la web que aparecerá, escogemos la opción “Package” del menú superior y luego escogemos cualquier clase para ver sus detalles. También puedes abrir este fichero con el navegador web desde el workspace, es decir, sin usar Eclipse.

Como podemos imaginar, Eclipse hace por nosotros las llamadas al comando javadoc explicadas en el apartado 5.2.1 según sea la configuración que le hemos indicado.

A continuación podemos ver cómo se muestra la documentación generada con Javadoc en Eclipse para una clase llamada Tank.



6.2. JavaFX

6.2.1. Introducción

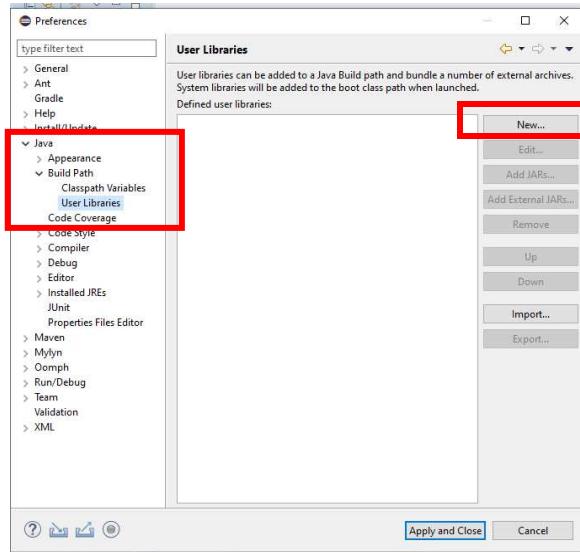
Las interfaces gráficas (en inglés, *Graphical User Interface*, GUI) son un elemento básico hoy en día. Java tiene tres bibliotecas para hacer interfaces gráficas: AWT, Swing y JavaFX. Resumiendo mucho, se podría decir que Swing es más avanzado que AWT y que JavaFX ha llegado para ser el sustituto de Swing. Hasta hace relativamente poco, Swing era la librería predominante, pero JavaFX está comenzando a ganarle terreno. No obstante, en un programa puedes combinar componentes (i.e. botón, campo de texto, desplegable, etc.) de ambas librerías, aunque no es muy recomendable. Oracle no ha dado por obsoleto a Swing, pero parece ser que no le va a incorporar mejoras en el futuro. Es por esto que en esta práctica nos centraremos en JavaFX. Asimismo, con JDK 11 Oracle ha decidido quitar del núcleo del JDK la librería JavaFX para que ésta sea independiente y tenga su propio ritmo de desarrollo, liberando así su evolución. Esto, evidentemente, conlleva algunos cambios, especialmente a la hora de configurar el entorno de trabajo (i.e. Eclipse).

Swing y JavaFX comparten muchas cosas, como es el uso de componentes (también conocidos como *widgets*), pero tienen cosas totalmente diferentes, como por ejemplo la forma de trabajar, además de sus capacidades.

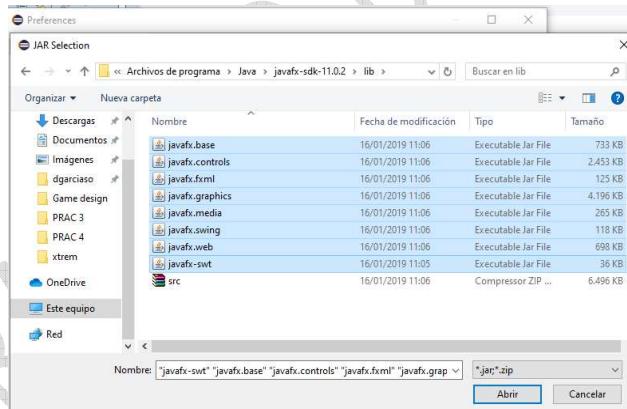
6.2.2. Configuración de JavaFX en Eclipse IDE

En este apartado vamos explicar, paso a paso, cómo configurar JavaFX en Eclipse.

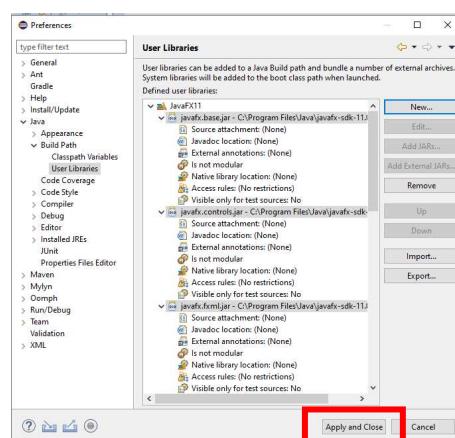
- Asegúrate de que tienes instalado JDK 11 o superior y además lo tienes establecido como compilador a usar en Eclipse (o en el proyecto donde quieras usar JavaFX).
- Como ahora JavaFX está fuera del núcleo de JDK, hay que ir a la siguiente página web:
<https://gluonhq.com/products/javafx/>
- En dicha página web instalaremos **JavaFX 11 (Long Term Support) o versión posterior**. Aquí presentamos los pasos para Windows. Así pues, descargamos la opción JavaFX Windows SDK.
- Descomprimimos el fichero donde creamos conveniente, p.ej. dentro de Archivos de programa/Java.
- Dentro de Eclipse ves a la opción Window del menú superior y escoge la opción Preferences.
- Se abrirá una ventana y escoges del menú de la izquierda la opción Java → Build Path → User Libraries. Haz click en el botón New...



- En la nueva pantalla nos pide un nombre para la librería. Escribimos: JavaFX11 y hacemos click en el botón OK.
- A continuación hacemos click en el botón Add External JARs... buscamos la ruta hasta el directorio de JavaFX11 que hemos descomprimido anteriormente. En ella hay tres subdirectorios. **Hay que entrar en lib y escoger todos los ficheros jar y NO seleccionar el fichero src.zip.** Una vez seleccionados clicamos en Abrir.



- El resultado de hacer los pasos anteriores debe ser el que se muestra en la siguiente figura. Si es así, haz click en el botón Apply and Close.



- Después haz botón derecho sobre el proyecto BalUOCesto y escoge la opción Properties. En la ventana que se abrirá escoge la opción Java Build Path del menú izquierdo. En la parte central escoge la pestaña Libraries y a continuación escoge Modulepath. Después haz click en Add Library... Luego elige User Libraries de la ventana flotante, clica Next, escoge JavaFX11 y finalmente Finish. Despues clica en Apply and Close.
- Para que Eclipse (en verdad JDK) detecte la librería JavaFX hay que ir importando los paquetes que usemos de JavaFX en el fichero module-info.java. Como en verdad este fichero no lo usamos, puesto que es un concepto muy avanzado (que se verá en la PAC 3), lo más sencillo, por ahora, es hacer lo siguiente: Ves al fichero module-info.java y comenta el código:

```

/*
module BalUOCesto{
}
*/

```

- En este punto los errores relacionados con el uso de clases de la librería JavaFX deberían desaparecer.
- No obstante, aún no puedes ejecutar correctamente la aplicación, puesto que hay que configurar la ejecución. Para ello ves a Run configurations → Arguments y escribe en VM arguments lo que se muestra a continuación.

```
--module-path "C:\Program Files\Java\javafx-sdk-11.0.2\lib" --add-modules javafx.controls,javafx.fxml
```

Lo resaltado en rojo lo debes cambiar por la ruta donde has descomprimido JavaFX.

- Ahora ves al menu superior Help → Install New Software... En el campo Work with de la ventana que te aparecerá escribe la dirección: <https://download.eclipse.org/efxclipse/updates-released/3.5.0/site/> y pulsa enter para que Eclipse busque paquetes en esta dirección. Una vez terminada la búsqueda, marca tanto e(fx)clipse - install como e(fx)clipse - single components.
- Haz click en el botón Next. Aparecerá otra ventana donde también debes hacer click en el botón Next.
- A continuación aparecerá una pantalla para que aceptes la licencia. Marca "*I accept the terms...*" y pulsa en Finish.
- Eclipse instalará los paquetes. Cuando termine es posible que te pida reiniciar el IDE. Hazlo.
- A partir de ahora podrás crear proyectos que usen JavaFX. Al hacer File → New → Other te aparecerá una carpeta llamada JavaFX con plantillas.

6.2.3. Modelo en JavaFX

Una de las ventajas de JavaFX es que usa el patrón MVC para crear los programas, lo cual, como hemos visto, es muy interesante. En esta práctica no trabajaremos la parte “modelo” al estilo JavaFX, es decir, para sacarle partido a todo el potencial de JavaFX. Esto te lo dejamos a ti, por si quieres indagar en JavaFX. Si estás interesado/a, busca los conceptos *properties* y *binding* de JavaFX. Nosotros nos limitaremos a usar el modelo de manera clásica/tradicional, es decir, con las clases que hemos codificado en el paquete `edu.uoc.baluocesto.model` siguiendo el paradigma POO.

6.2.4. Vistas en JavaFX (parte visual)

En cuanto a las vistas, decirte que éstas, o bien se crean programando en Java, o bien se crean mediante unos ficheros llamados FXML, que no dejan de ser unos ficheros que contienen un código XML ad hoc para JavaFX. La primera opción sólo debe usarse para crear componentes (o toda la vista si fuera necesario) en tiempo de ejecución (p.ej. mostrar un botón tras marcar un *checkbox*). La segunda forma, la más habitual, se asemeja a otros tipos de desarrollo, como es la creación de *apps* en Android (donde la parte visual de la vista es un fichero XML y la parte de interacción de la vista es una clase codificada en Java) o el *front-end* de una web (la parte visual hecha con HTML+CSS y la parte interactiva que se comunica con el controlador hecha en JavaScript).

Para que lo veas más claro, piensa que cada pantalla/vista de tu programa será un fichero FXML (parte visual) más un fichero Java (parte interactiva y de comunicación con el controlador). Gracias a esto se logra separar/desacoplar lo visual (p.ej. forma, estilo y ubicación de un botón) de la lógica (i.e. funcionamiento/comportamiento del programa, por ejemplo, al hacer click en un botón). Asimismo, como cada vista tiene un fichero Java para la parte interactiva, desde él podremos añadir, mediante código, componentes a la vista si fuera necesario, combinando así los dos modos de creación de componentes (i.e. vía código y vía FXML).

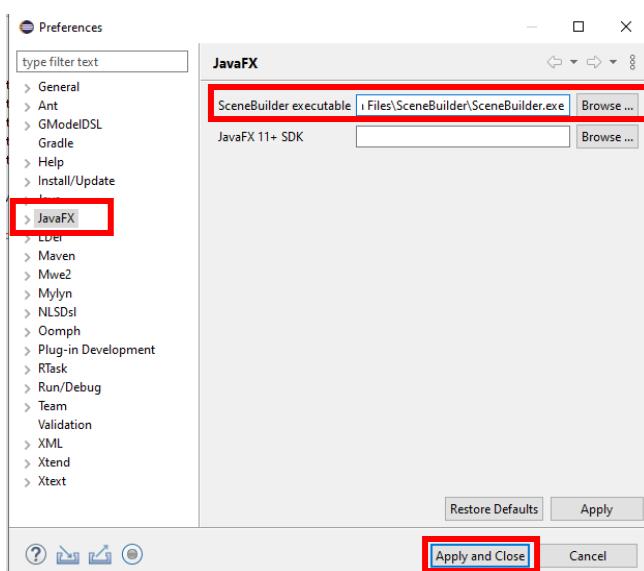
Seguro que estás pensando que debe de ser muy laborioso hacer una interfaz si la tienes que hacer escribiendo código FXML (que es un XML ad hoc). Tienes toda la razón, es muy arduo crear interfaces visuales mediante código FXML y además “a ciegas”. Por eso existen programas que ayudan a realizar los ficheros FXML. Uno de estos programas es [SceneBuilder](#). SceneBuilder te permite crear cada pantalla/vista de tu programa de manera visual, es decir, arrastrando y soltando componentes en la escena. Una vez tengas la pantalla/vista con la apariencia que deseas, sólo tienes que pedirle a SceneBuilder que te genere el fichero FXML que codifica la pantalla/vista que has creado. ¡Así de simple!

Para instalar Scenebuilder ves a la siguiente web: <https://gluonhq.com//products/scene-builder/#download> e instala SceneBuilder for Java 11 (sirve para JDK 11 y superiores). En Windows recomendamos usar al versión instalador, no el .jar.

Puedes trabajar con SceneBuilder en paralelo con Eclipse. Es decir, creas los ficheros FXML con SceneBuilder y luego los copias en tu proyecto de Eclipse. Pero también puedes hacer algo mejor, que SceneBuilder y Eclipse se entiendan. Es decir, que desde Eclipse crees un fichero FXML nuevo, lo abras y modifiques con SceneBuilder desde Eclipse, y los cambios se vean

reflejados automáticamente en tu proyecto. Para lograr que SceneBuilder y Eclipse trabajen colaborativamente, sigue los pasos que te detallamos a continuación:

- Dentro de Eclipse ves a la opción Window del menú superior y escoge la opción Preferences.
- En la ventana que te aparecerá, busca en el listado de la izquierda la opción JavaFX.
- Seleccionala. A la derecha te debería aparecer un único campo donde puedes indicar dónde está ubicado el ejecutable de SceneBuilder (ver la figura). Indícalo y haz click en el botón Apply and Close.

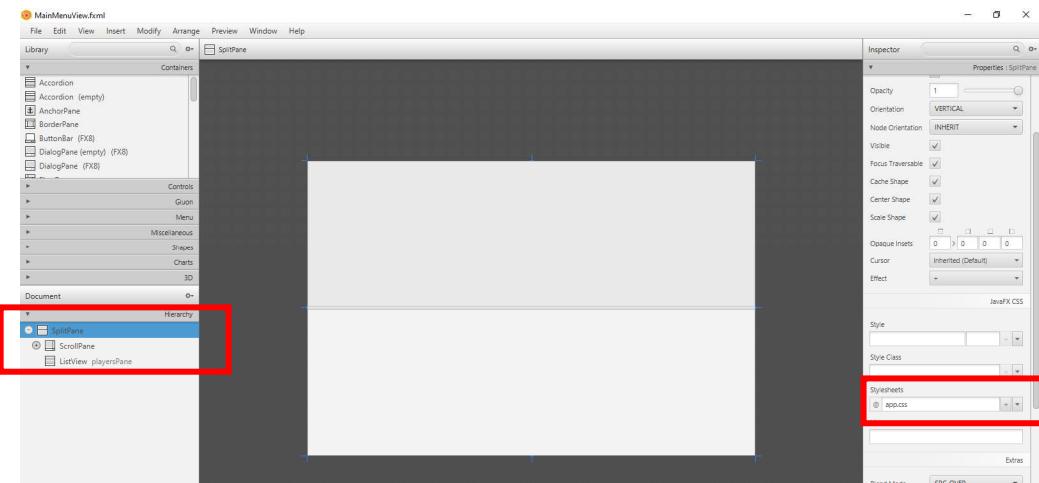


Gracias a esta configuración podrás hacer clic derecho sobre cualquier fichero FXML en Eclipse y escoger la opción Open with SceneBuilder para que se te abra dicho fichero en el software de edición SceneBuilder. Además, si guardas los cambios en SceneBuilder, éstos se verán reflejados en tu proyecto Eclipse. A veces hay que refrescar el proyecto (p.ej. haciendo F5 en Eclipse). Si en vez de abrir un fichero FXML con SceneBuilder, lo abres haciendo doble clic, entonces verás el código FXML en el editor de Eclipse.

Las vistas, es decir, cada FXML puede ser personalizado mediante un fichero CSS (o incluso varios FXML pueden compartir un mismo fichero CSS). Un fichero CSS (*Cascade Style Sheet*) es un documento en el que se especifican diferentes estilos, desde generales hasta concretos (p.ej. para un botón determinado). Estos ficheros son esenciales cuando se programan webs, ya que facilitan la escalabilidad, el mantenimiento y la gestión de posibles cambios. JavaFX usa CSS aunque utiliza sus propias propiedades, a pesar de que muchas son similares a las propiedades web, por lo que es sencillo usarlas si se sabe CSS web. Nosotros esto no lo trataremos aquí.

Si en un fichero FXML seleccionas un elemento a través del lateral izquierdo (Document → Hierarchy) y luego miras el panel Properties que hay a la derecha del programa (ver figura), dentro encontrarás un apartado llamado JavaFX CSS. Este apartado

permite indicar estilos directamente (apartado Style), asignarle una clase de un fichero CSS y añadir un fichero CSS. Justo en el apartado Stylesheets, hemos puesto app.css que es el nombre del fichero CSS de la aplicación. En el caso del color y el tamaño de un texto (i.e. label) o un botón (i.e. button), SceneBuilder ya trae un apartado para ellos dentro de Properties sin necesidad de usar propiedades CSS.



6.2.5. Vistas en JavaFX (parte interactiva)

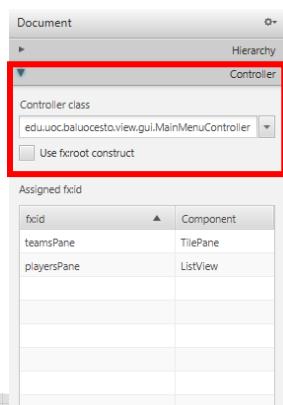
Hasta ahora sólo hemos visto cómo crear la parte visual de una interfaz/vista/pantalla pero, ¿cómo le damos vida? ¿Cómo interactúa el usuario con ella? Pues bien, esto se hace mediante eventos (tanto en JavaFX como en Swing y AWT). Esto implica una nueva forma de programar que es denominada programación orientada a eventos (POE). En la POE, el flujo de ejecución del programa viene definido por las acciones/eventos (p.ej. clic, doble clic, presionar una tecla, etc.) realizados por un agente externo al programa, p.ej. un usuario, otro programa, etc. La POE no es incompatible con la POO. De hecho, con Java, seguiremos trabajando con objetos que son capaces de responder a eventos. Así pues, POO y POE se complementan. Por ejemplo, tendremos un objeto de tipo Button que estará escuchando (con lo que se denomina un *Listener*) que ocurra un evento. Cuando ocurra un evento concreto como puede ser hacer clic en el botón, si tiene codificado un método asociado a “hacer clic”, dicho método se ejecutará al producirse el evento. Es decir el *Listener* invocará al método asociado.

Como ya hemos dicho, una pantalla/interfaz/vista tendrá, por un lado, su fichero que define la parte visual (i.e. un fichero FXML) y, por otro, un fichero Java que será el encargado de gestionar los eventos producidos sobre la vista (i.e. fichero FXML) y seguir la lógica del programa según estos eventos. Como habrás visto en el proyecto Eclipse, dentro del paquete view.gui hay unos ficheros Java llamados igual que los FXML pero cuyos nombres acaban con la coletilla Controller. Ésta es una norma no escrita que se utiliza para denominar a estos ficheros Java que se encargan de la parte interactiva de los ficheros FXML. Así sabemos que el fichero Java encargado de la parte interactiva de un fichero FXML se llama igual que el fichero FXML pero con la coletilla Controller. A partir de ahora llamaremos controladores a estos ficheros.

Es importante destacar que **SceneBuilder sólo detecta/encuentra las clases (i.e. ficheros Java) que están en el mismo paquete que el fichero FXML que se está editando.**

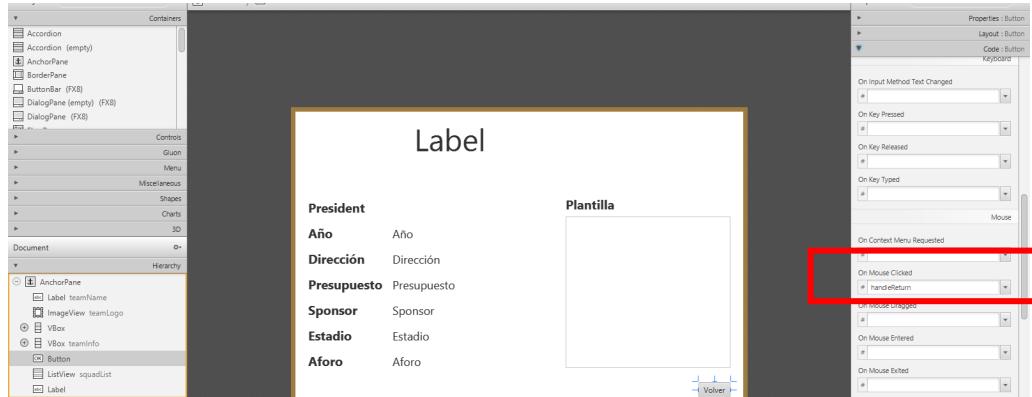
Como ya habrás podido intuir, es esencial vincular un controlador/parte interactiva (fichero Java) a una vista (fichero FXML). Esto se puede hacer de dos formas:

- Editando directamente el código XML del fichero FXML: abre el fichero FXML con Eclipse. En el tag/etiqueta raíz se puede añadir un campo/atributo llamado `fx:controller` cuyo valor es el nombre de la clase controladora (incluyendo el paquete en el que está ubicado).
- Usando SceneBuilder: abre el fichero FXML con SceneBuilder. Verás que hay una opción llamada `Controller` dentro de la opción `Document` del menú izquierdo (ver figura). En ella hay un campo denominado `Controller class` en el que puedes asignarle la clase controladora. SceneBuilder detectará todas aquellas clases que estén en el mismo paquete que el fichero FXML que se está editando. Para cada FXML deberás escoger del desplegable la opción (i.e. controlador) que corresponda.

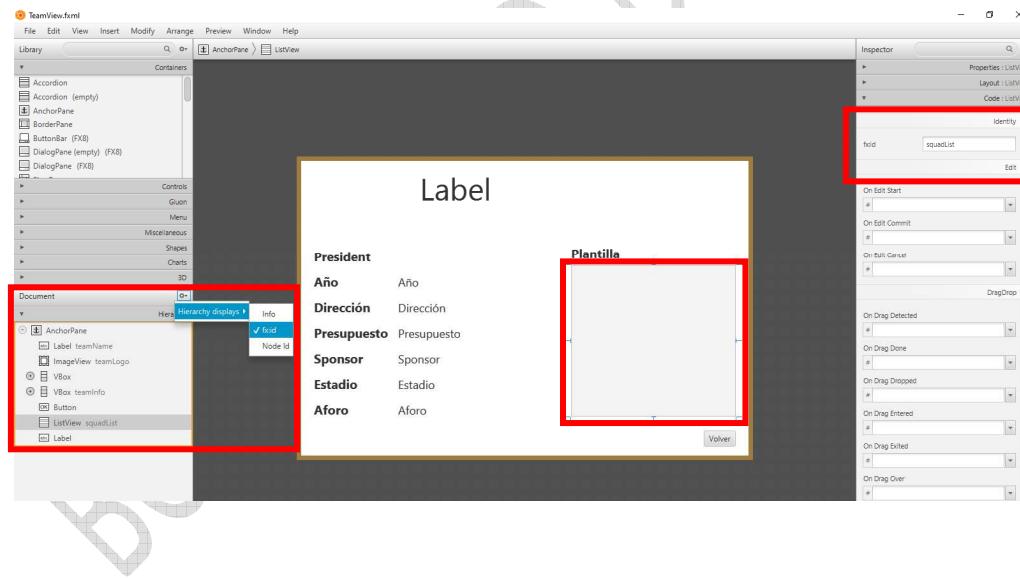


En la clase controladora crearemos todos los métodos y atributos que queremos que interactúen con la parte visual de la vista. Es una buena práctica preceder a todos los atributos y métodos del controlador que queramos que sean accesibles desde las vistas con la anotación `@FXML`. Los atributos y métodos precedidos con `@FXML`, aunque sean privados, serán visibles para la vista donde usemos el controlador que los contiene. Si un atributo o método es público, no es necesario usar `@FXML`, pero sí recomendable.

Imagina que creamos un método llamado `handleReturn` que sirve para volver a la vista/pantalla anterior a la que estamos. En este momento debemos vincular un elemento de la interfaz con el método `handleReturn`. Para hacer esta asignación/vinculación, lo más cómodo es usar SceneBuilder. Abrimos el fichero FXML con SceneBuilder, seleccionamos un `Button` que hayas puesto en la interfaz y en el apartado `Code` del lateral derecho buscamos la opción `On Mouse Released` (ver figura). Allí hemos escrito/seleccionado el nombre de método a ejecutar, en este caso, `handleReturn`. Esta asignación también la puedes ver reflejada, obviamente, en el código FXML del fichero `TeamView.fxml` (puedes verlo con Eclipse).



Finalmente, si queremos que un elemento de la interfaz sea accesible desde un controlador (i.e. código Java), entonces debemos asignarle un identificador. Para ello En SceneBuilder seleccionamos uno de los elementos de la interfaz gráfica y, en el menú derecho, en la opción Code, encontraremos un campo llamado `fx:id`. Lo habitual es ponerle como nombre uno que también usemos como atributo en el controlador (precedido de `@FXML`). De esta forma, el controlador puede acceder a este componente para modificarlo o consultararlo, y la vista a la información del atributo de la clase de tipo controladora. O dicho de otro modo, el atributo de la clase controladora hace referencia al componente de la vista porque tienen el mismo nombre y porque en la clase controladora hemos puesto `@FXML` delante del atributo.



En el ejemplo que podemos ver en la imagen anterior, el nombre que le hemos dado a la ListView es `squadList`. En el código Java del controlador tendremos un atributo llamado `squadList` de tipo `ListView` y precedido por la anotación `@FXML`.

6.3. JUnit 5

La librería JUnit permite a los desarrolladores hacer tests unitarios. En este apartado veremos qué es un test unitario y cómo usar JUnit con Eclipse.

6.3.1. Test unitario

Antes de nada, vamos a definir qué es un test unitario:

Definición tests unitarios (*unit tests*)

Son las pruebas de más bajo nivel en la programación. Con ellas se comprueba el funcionamiento de las unidades lógicas independientes, normalmente los métodos. Son pruebas muy rápidas de llevar a cabo y son las que los desarrolladores están constantemente pasando para verificar que su software funciona adecuadamente.

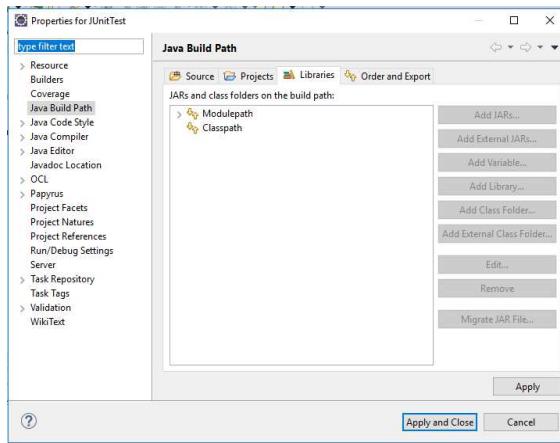
Estos son los tests más importantes. Cada test unitario es un paso que andamos en el camino de la implementación correcta del software. Los desarrolladores utilizan los tests unitarios para asegurarse de que el código funciona como esperan que funcione, al igual que el cliente usa los tests de aceptación/cliente para asegurarse de que los requisitos de negocio se cumplan con el software como se espera que lo haga. Todo test unitario debe ser:

- **Atómico:** significa que el test unitario prueba la mínima cantidad de funcionalidad posible. Esto es, probará un único comportamiento de un método de una clase. Como ya sabes, un método puede presentar distintas respuestas ante distintas entradas o distinto contexto. El test unitario se ocupará exclusivamente de uno de esos comportamientos, es decir, de un único camino de ejecución.
- **Independiente:** significa que un test no puede depender de otros para producir un resultado satisfactorio. No puede ser parte de una secuencia de tests que se deba ejecutar en un determinado orden. Debe funcionar siempre igual independientemente de que se ejecuten otros tests o no.
- **Inocuo:** esta propiedad significa que no altera el estado del sistema. Al ejecutarlo una vez, produce exactamente el mismo resultado que al ejecutarlo veinte veces. Además, no altera la base de datos, ni envía e-mails ni borra ficheros, ni los crea (y si los tiene que crear, los debe borrar). Un test, una vez ejecutado, debe comportarse como si no se hubiera ejecutado.
- **Rápido:** puesto que ejecutamos un gran número de tests cada pocos minutos, resultaría muy poco productivo tener que esperar unos cuantos segundos cada vez. Un único test tiene que ejecutarse en una pequeña fracción de segundo.

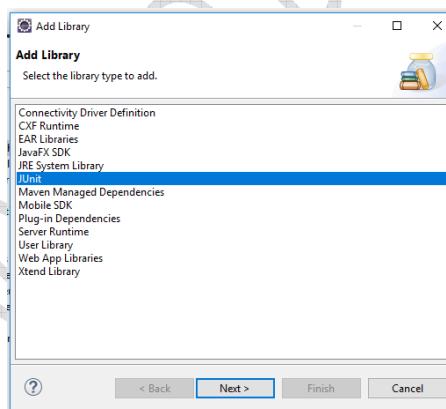
6.3.2. Configuración del plugin JUnit

Al instalar Eclipse IDE se realiza la instalación de JUnit, una biblioteca que viene por defecto en Eclipse. No obstante, a continuación se ilustran los pasos necesarios para configurar un entorno de trabajo con pruebas. Algunos aspectos o nombres pueden cambiar entre lo que se explica en este tutorial y lo que te puedes encontrar en tu entorno de trabajo. En tal caso, debes ser capaz de hacer las analogías pertinentes.

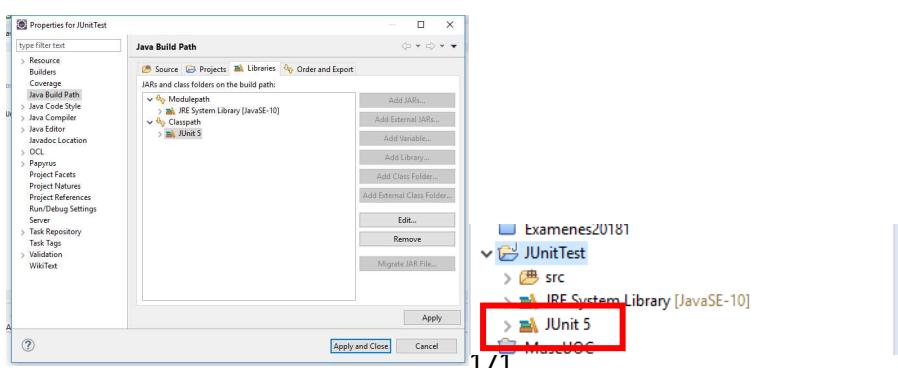
1. Imaginemos que tenemos un proyecto ya creado llamado JUnitTest que está vacío, es decir, no contiene ningún fichero .java. Si ahora queremos indicar que este proyecto debe permitir JUnit, entonces debemos hacer clic derecho sobre el nombre del proyecto en el *Package Explorer* y escoger *Properties*. En la ventana que nos aparecerá debemos ir a la opción *Java Build Path* del menú de la izquierda y, seguidamente, escoger la pestaña *Libraries* en la zona central y seleccionar *Classpath*. A continuación hacemos clic en el botón *Add Library...* (ver siguiente figura).



2. En la ventana que nos aparecerá seleccionamos *JUnit* del listado y pulsamos en *Next* (ver siguiente figura).



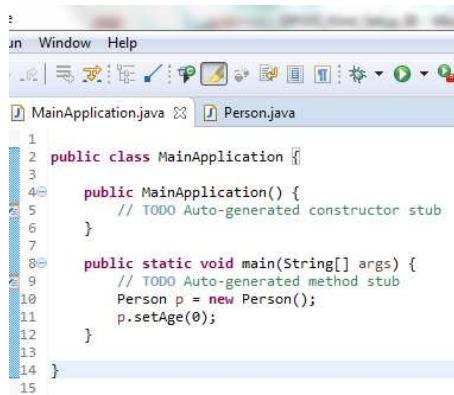
3. En la siguiente pantalla escogemos la versión más alta de JUnit (i.e. versión 5) y decimos *Finish* (ver siguiente figura). A continuación vemos la pantalla de primera imagen con JUnit añadida. Pulsamos en *Apply and Close*. Ahora en el *Package Explorer* vemos que se ha añadido la librería JUnit 5. A partir de aquí ya podemos usar JUnit para hacer tests unitarios.



6.3.3. Usando JUnit

Vamos a ver un ejemplo dentro del proyecto JUnitTest mencionado en el apartado anterior.

- Ahora añadimos una nueva clase llamada MainApplication que contendrá un método main. El código del fichero MainApplication.java es el que se muestra en la figura siguiente.

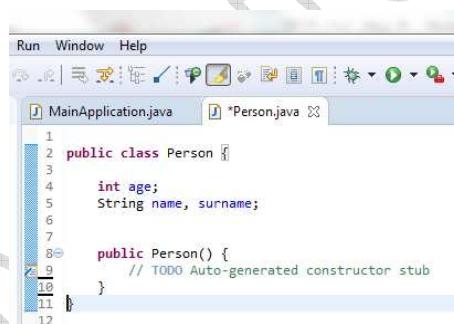


```

Main Application.java Person.java
1 public class MainApplication {
2     public MainApplication() {
3         // TODO Auto-generated constructor stub
4     }
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         Person p = new Person();
8         p.setAge(0);
9     }
10 }
11
12
13
14
15

```

- Ahora añadimos una nueva clase llamada Person. Una vez creado el fichero Person.java, escribimos los atributos, como se muestra en la imagen siguiente.

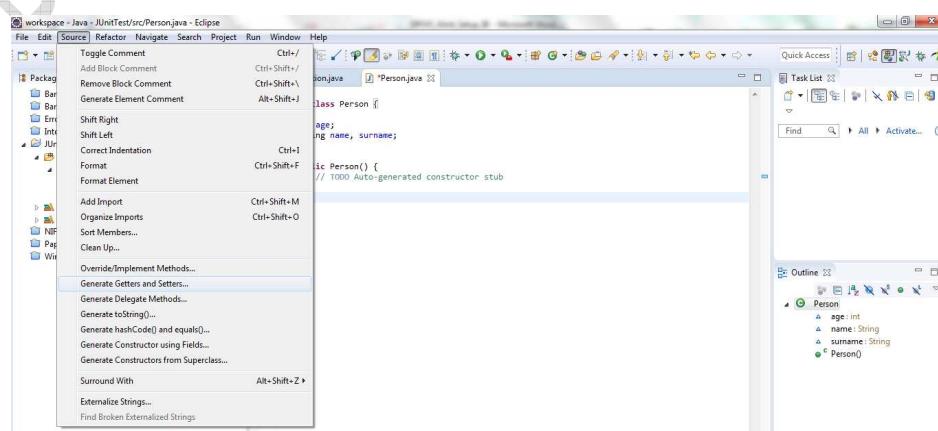


```

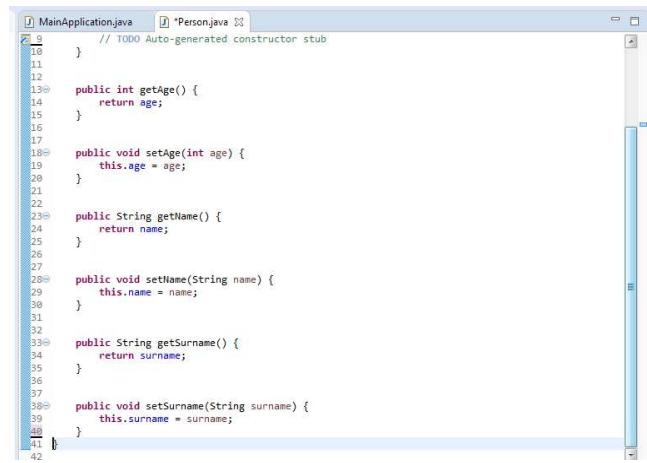
Main Application.java *Person.java
1 public class Person {
2     int age;
3     String name, surname;
4
5     public Person() {
6         // TODO Auto-generated constructor stub
7     }
8
9
10
11
12

```

- Para generar los *getters* y *setters* de la clase Person, usaremos un atajo de Eclipse. Estando activado el fichero Person.java en el editor de código, vamos a la opción Source del menú superior de Eclipse y seleccionamos la opción *Generate Getters and Setters...* (ver siguiente figura). En la ventana que nos aparecerá marcamos todos los atributos y pulsamos en OK. ¿Rápido, verdad?



4. El resultado es el que se muestra en la siguiente imagen.

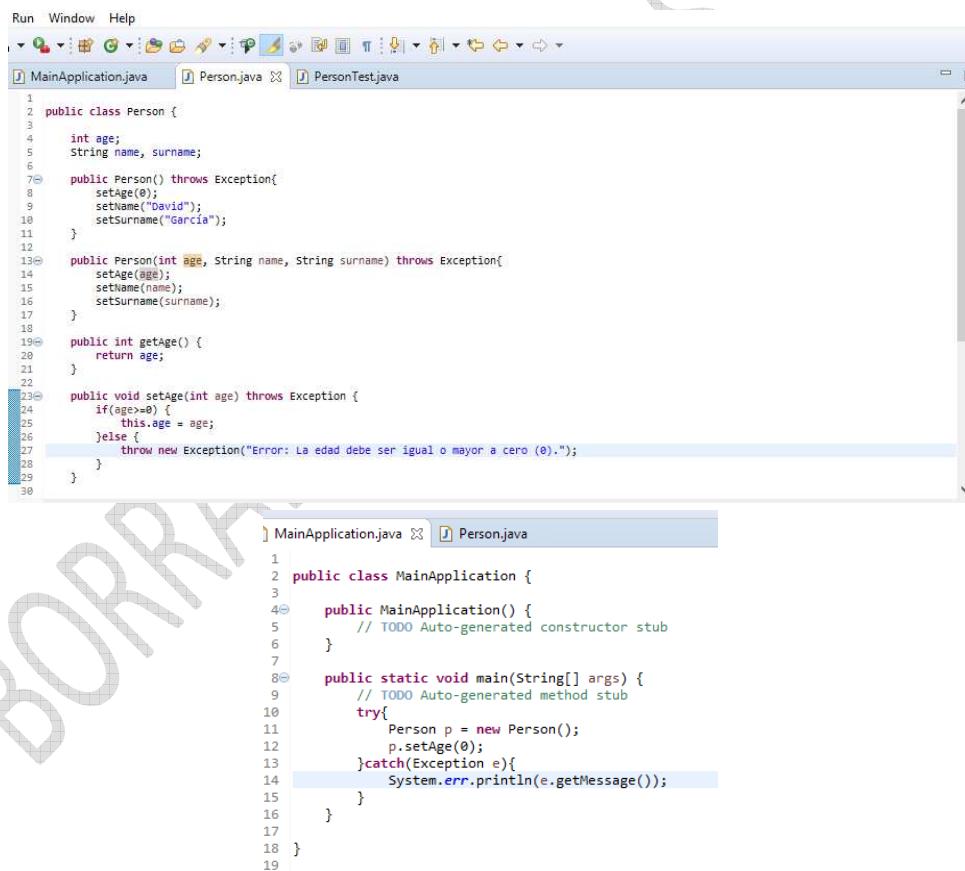


```

1  package com.uoc;
2
3  public class Person {
4
5      int age;
6      String name, surname;
7
8      public Person() throws Exception{
9          setAge(0);
10         setName("David");
11         setSurname("García");
12     }
13
14     public Person(int age, String name, String surname) throws Exception{
15         setAge(age);
16         setName(name);
17         setSurname(surname);
18     }
19
20     public int getAge() {
21         return age;
22     }
23
24     public void setAge(int age) throws Exception {
25         if(age>=0) {
26             this.age = age;
27         }else {
28             throw new Exception("Error: La edad debe ser igual o mayor a cero (0).");
29         }
30     }
31
32     public String getName() {
33         return name;
34     }
35
36     public void setName(String name) {
37         this.name = name;
38     }
39
40     public String getSurname() {
41         return surname;
42     }
43
44     public void setSurname(String surname) {
45         this.surname = surname;
46     }
47
48 }

```

5. Añadimos código en el constructor por defecto, añadimos un constructor con argumentos y cambiamos el código de setAge tal y como se muestra en la siguiente figura. También cambiamos el código de MainApplication para que capture la excepción de Person.



MainApplication.java

```

1  package com.uoc;
2
3  public class MainApplication {
4
5      public MainApplication() {
6          // TODO Auto-generated constructor stub
7      }
8
9      public static void main(String[] args) {
10         // TODO Auto-generated method stub
11         try{
12             Person p = new Person();
13             p.setAge(0);
14         }catch(Exception e){
15             System.err.println(e.getMessage());
16         }
17     }
18 }

```

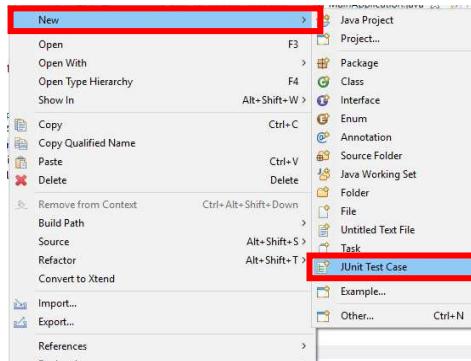
Person.java

```

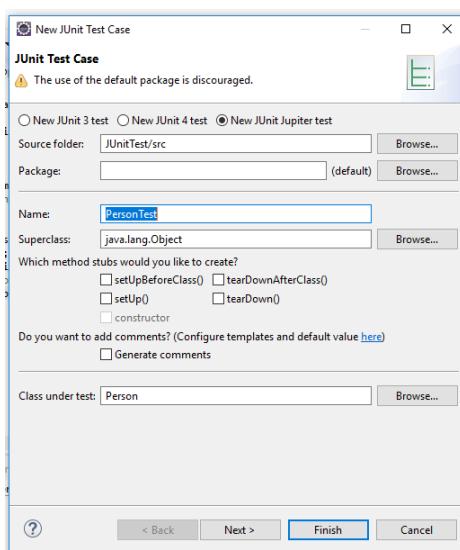
1  package com.uoc;
2
3  public class Person {
4
5      int age;
6      String name, surname;
7
8      public Person() throws Exception{
9          setAge(0);
10         setName("David");
11         setSurname("García");
12     }
13
14     public Person(int age, String name, String surname) throws Exception{
15         setAge(age);
16         setName(name);
17         setSurname(surname);
18     }
19
20     public int getAge() {
21         return age;
22     }
23
24     public void setAge(int age) throws Exception {
25         if(age>=0) {
26             this.age = age;
27         }else {
28             throw new Exception("Error: La edad debe ser igual o mayor a cero (0).");
29         }
30     }
31
32     public String getName() {
33         return name;
34     }
35
36     public void setName(String name) {
37         this.name = name;
38     }
39
40     public String getSurname() {
41         return surname;
42     }
43
44     public void setSurname(String surname) {
45         this.surname = surname;
46     }
47
48 }

```

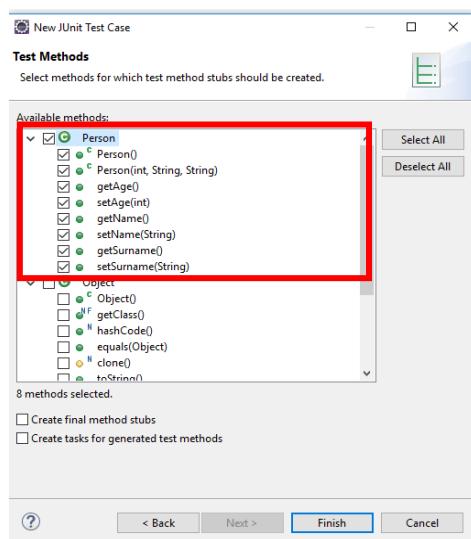
6. Ahora, en la ventana *Package Explorer*, hacemos botón derecho con el *mouse* encima de Person.java y hacemos *New → JUnit Test Case* (ver siguiente imagen).



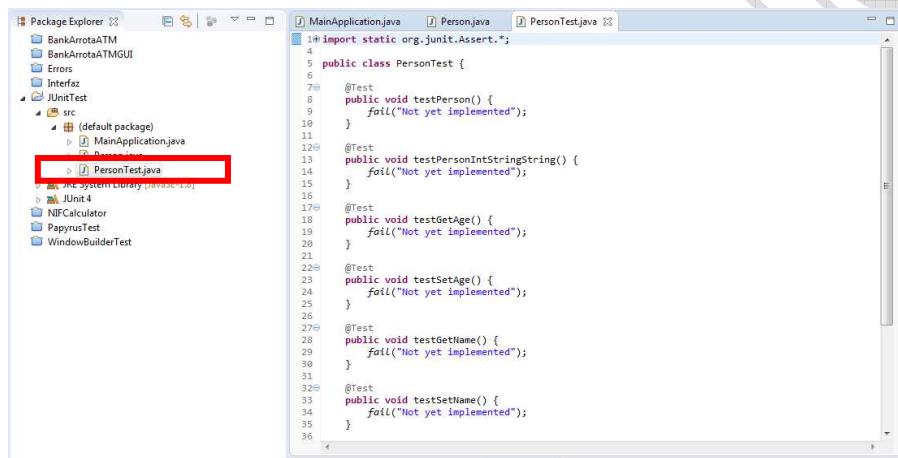
7. De este modo, nos solicita los métodos del código para los cuales se quiere generar la plantilla de pruebas (ver siguiente figura). En esta ventana hacemos clic en *Next* (la opción *New JUnit Jupiter Test* es lo mismo que decir *JUnit 5*)



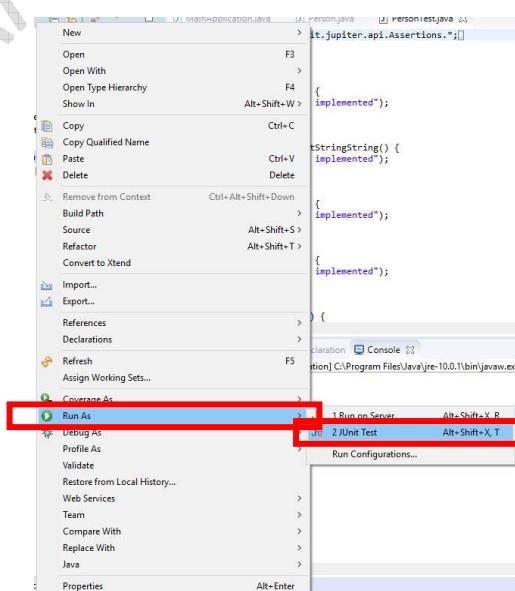
8. En la ventana que nos aparecerá seleccionamos los métodos que queremos testear. Por ejemplo, seleccionamos la clase Person para así seleccionar, de una vez, todos sus métodos (ver siguiente figura). Una vez lo tengamos, hacemos clic en el botón *Finish*.



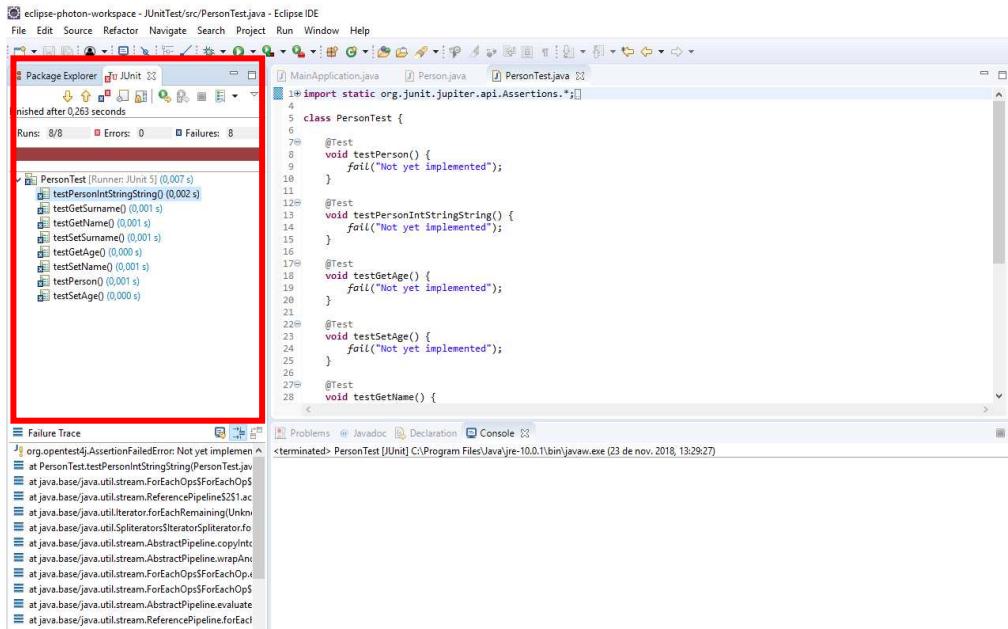
9. Ahora en el *Package Explorer* nos ha aparecido una nueva clase llamada PersonTest.java (ver Figura 12) con el aviso de error. Una forma de quitar los errores, es eliminando el fichero module-info.java (en la PAC 2 se explicará qué es este fichero). Si miramos el código de PersonTest.java, veremos que son pruebas relacionadas 1:1 con los métodos de la clase Person y que inicialmente serán fallidas para recordarnos que deben ser implementadas. De hecho, sale el texto "Not yet implemented!". Podremos añadir más métodos de prueba anteponiendo la anotación @Test delante de la firma de dicho método. Gracias a @Test ese método es considerado prueba unitaria. Fíjate que si un método está sobrecargado, como puede ser el constructor, para diferenciarlo le añade como coletilla los tipos de los argumentos que recibe, p.ej. testPersonIntStringString es el método de testeo que crea JUnit para el constructor con argumentos, i.e. Person(int age, String name, String surname).



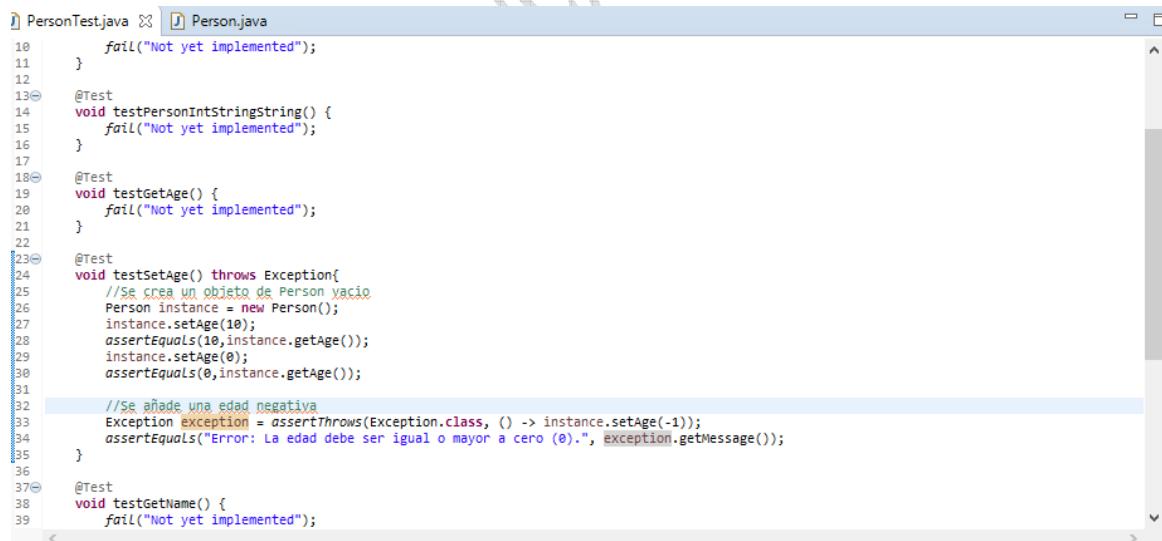
10. A continuación se deben ejecutar las pruebas para comprobar que NO se han superado (lo cual es lógico ahora mismo). En este caso se debe marcar el fichero de Test (en nuestro ejemplo PersonTest.java) y con el botón derecho del mouse pulsar *Run As→JUnit Test* (ver siguiente figura).



11. Ahora nos aparece una pantalla en la que se muestra los tests que NO se han superado, y el número de tests que han fallado. En este caso hemos configurado 8 tests para 8 métodos que queríamos comprobar su funcionalidad y todas son fallidas (ver Figura 14). Es posible que en Windows 10 os salte una pantalla de aviso del Firewall que deberéis aceptar.



12. Ahora vamos a codificar el método `testSetAge` (ver imagen siguiente).



```

10     fail("Not yet implemented");
11 }
12
13@Test
14 void testPersonIntStringString() {
15     fail("Not yet implemented");
16 }
17
18@Test
19 void testGetAge() {
20     fail("Not yet implemented");
21 }
22
23@Test
24 void testSetAge() throws Exception{
25     //Se crea un objeto de Person vacío
26     Person instance = new Person();
27     instance.setAge(10);
28     assertEquals(10,instance.getAge());
29     instance.setAge(0);
30     assertEquals(0,instance.getAge());
31
32     //Se añade una edad negativa
33     Exception exception = assertThrows(Exception.class, () -> instance.setAge(-1));
34     assertEquals("Error: La edad debe ser igual o mayor a cero (0).", exception.getMessage());
35 }
36
37@Test
38 void testGetName() {
39     fail("Not yet implemented");

```

13. Mirando el código de la imagen anterior, se puede ver que el primer paso consiste en cambiar la firma del método para que lance/propague las excepciones que recibe su código, en este caso, el método `setAge` lanza `Exception`. Ya en el cuerpo del método, lo primero es crear un objeto que disponga del método que se quiere comprobar. Por consiguiente, se crea un objeto de la clase `Person`.

```
Person instance = new Person();
```

Una vez se dispone de un objeto Person válido, se comienza a invocar los métodos que se quieren comprobar para chequear que los valores que dispone son correctos. Observamos que sobre un objeto (`instance`) de la clase Person se invoca el método `setAge` (el cual se quiere testear). En este caso se ha invocado con un valor de 10 años.

```
instance.setAge(10);
```

Nuestra persona por consiguiente debería tener una edad de 10 años. Para comprobar que es así, el *framework* JUnit aporta métodos especiales denominados *asserts* que nos deben devolver valores booleanos (true o false) para poder comprobar que los métodos funcionan adecuadamente. Por ejemplo, se hace uso del método `assertEquals`, el cual está comprobando que el valor 10 es igual que el valor obtenido por `instance.getAge()`.

Más adelante se quiere comprobar si el método `setAge` lanza la excepción que tiene programada cuando el valor introducido es negativo. Para especificar dicha operación en JUnit 5 se hace uso del nuevo método `assertThrows` que devuelve un objeto de tipo `Exception` cuyo mensaje se puede comparar con el método `assertEquals` que ya hemos visto.

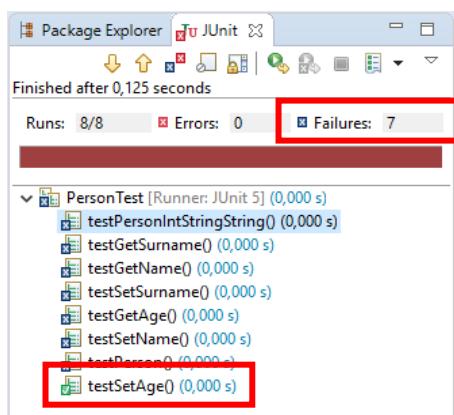
```
Exception exception = assertThrows(Exception.class, () -> instance.setAge(-1));

assertEquals("Error: La edad debe ser igual o mayor a cero (0).",
             exception.getMessage());
```

Observamos que el primer parámetro de `assertThrows` es el tipo de excepción que se espera, en este ejemplo, esperamos que la clase `Exception.class`. En caso de que se disponga de una clase específica para excepciones (p.ej. `PersonException`) bastaría con especificar ahí la clase de la excepción que se espera (p.ej. `PersonException.class`). El segundo parámetro es una expresión lambda (se verá qué es en la PEC 2). De momento, sólo tenéis que poner la llamada al método después de `->`. Finalmente, en la segunda sentencia, la excepción devuelta por `assertThrows` es comparada mediante su mensaje (`getMessage()`) con el mensaje esperado. Para que esto funcione hay que añadir el correspondiente `import`:

```
import static org.junit.jupiter.api.Assertions.*;
```

14. Si volvemos a ejecutar *Run As →JUnit Test*, veremos que ahora sólo tenemos 7 tests fallidos, en vez de 8. En el listado `testSetAge` aparece con un visto/check verde, lo que significa que ha pasado la prueba correctamente.



15. Si hacemos clic en alguno de los métodos fallidos en la ventana JUnit que se muestra en la Figura 16, en la parte inferior llamada *Failure Trace*, nos indica dónde ha fallado dicho método/test. Cuando todos los métodos de test de la clase PersonTest superen las pruebas, la franja roja se volverá verde y todos los métodos tendrán el visto/check verde al lado. Además pondrá Runs: 8/8, Errors: 0, Failures: 0. Gracias al uso de los tests sabemos que las clases se comportan como se espera que lo hagan. Si vamos codificando y pasamos los tests y uno que no fallaba ahora falla, podremos detectarlo a tiempo.

BORRADOR@DGS-UOC

7. Bibliografía

Java

Tutorials

Leanring

Paths:

<https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>

Patrick Niemeyer, Jonathan Knudsen. "Learning Java (Java Series), 2nd Edition". O'Reilly Media, Inc. 2002. ISBN: 978-0596002855.

Patrick Niemeyer, Daniel Leuck. "Learning Java (Java Series), 4th Edition". O'Reilly Media, Inc. 2013. ISBN: 978-1449372477.

Kathy Sierra, Bert Bates. "Head First Java, 2nd Edition. A Brain-Friendly Guide". O'Reilly Media, Inc. 2009. ISBN: 978-0596009205.

Peter van der Linden. "Just Java 2 (6th Edition)". Prentice Hall PTG, 2004. ISBN: 978-0137009909.