Guia de Java

PID_00273928

David García Solórzano

Temps mínim de dedicació recomanat: 16 hores

Universitat Oberta de Catalunya

David García Solórzano

És graduat superior en Enginyeria en Multimèdia i enginyer en Informàtica per la Universitat Ramon Llull des del 2007 i el 2008, respectivament. També és doctor per la Universitat Oberta de Catalunya des del 2013, on va realitzar una tesi doctoral relacionada amb l'àmbit de l'aprenentatge en línia (*e-learning*). Des del 2008 és professor de la Universitat Oberta de Catalunya en els Estudis d'Informàtica, Multimèdia i Telecomunicació.

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: David García Solórzano

Primera edició: setembre 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Autoria: David García Solórzano
Producció: FUOC

Índex

Introducció

1) Breu història

El llenguatge de programació Java va aparèixer el 1995, creat per James Gosling de l'empresa Sun Microsystems, que va ser adquirida per Oracle el 2010. Diuen que originalment s'havia d'anomenar Oak, a causa d'un roure[1] que hi havia al jardí al qual donava el despatx de Gosling. Més tard va ser batejat com a Green i al final es va anomenar Java. Diuen que el nom de Java va ser inspirat pel cafè de tipus Java que servien a la cafeteria on solia anar Gosling amb els seus companys. Per aquest motiu, el logo de Java és una tassa de cafè fumejant.

(1)En anglès, *oak*.

2) Edicions de Java

Java és tant un llenguatge de programació com una plataforma.

> Per **plataforma** entenem un entorn concret en el qual s'executen programes desenvolupats en llenguatge Java.

En l'actualitat hi ha quatre plataformes o edicions de Java que estan destinades a construir diferents tipus d'aplicacions. Cada plataforma Java està composta per una màquina virtual, anomenada *Java Virtual Machine* (JVM), i un conjunt de llibreries o API.[2]

(2)Acrònim d'*application programming interface.*

- **Java Virtual Machine (JVM)**: és un programa, per a un sistema operatiu concret, que executa programes desenvolupats en Java.

- **API**: és una col·lecció de llibreries que un programador pot fer servir per desenvolupar un nou programari.

Així doncs, un programa és desenvolupat per a una plataforma concreta de Java. Les quatre plataformes o edicions de Java que hi ha en l'actualitat són:

a) Standard Edition (SE). Aquesta edició és el nucli del llenguatge i la part que veurem en aquesta guia. La seva API inclou tot allò relacionat amb els tipus bàsics, l'orientació a objectes, classes per a xarxes, seguretat, accés a bases de dades, parseig d'XML, etc.

b) Enterprise Edition (EE). Aquesta plataforma és un superconjunt de l'Standard Edition (SE), és a dir, inclou l'SE i proporciona llibreries addicionals. Es fa servir per desenvolupar aplicacions d'una mida gran, distribuïdes, escalables, fiables i segures.

c) Micro Edition (ME). A diferència de l'Enterprise Edition, aquesta edició és un subconjunt de l'Standard Edition i està dissenyada per desenvolupar aplicacions en dispositius mòbils. Per aquest motiu, inclou llibreries especials útils per al desenvolupament d'aplicacions en dispositius mòbils i la seva màquina virtual està adaptada a les capacitats de telèfons intel·ligents, rellotges intel·ligents, etc.

d) Java Card. Aquesta edició és un subconjunt de l'Standard Edition i es fa servir per executar aplicacions de manera segura en targetes intel·ligents[3] o dispositius amb poca capacitat d'emmagatzematge. Aquesta tecnologia s'utilitza per exemple en targetes moneder.

[(3)] En anglès, *smart cards*.

3) Cicle de vida d'un programa Java

Un programador escriu un programa en Java. Per simplificar, imaginem que el seu programa només utilitza un fitxer anomenat `Program`. Aquest fitxer tindrà codi escrit en Java i la seva extensió serà `.java`; així doncs, el nom complet del fitxer serà `Program.java`. Una vegada escrit el fitxer `Program.java`, el programador l'ha de compilar. Per a això, crida el **Java Compiler (javac)** de la manera següent mitjançant la línia d'ordres:

```
javac Program.java
```

Si el programa estigués format per més d'un fitxer `.java`, aleshores podria haver compilat tots els arxius alhora amb l'ordre següent:

```
javac *.java
```

Amb aquest pas s'aconsegueix convertir el codi escrit en Java en un codi escrit en un llenguatge intermedi anomenat ***bytecode***. De fet, Java Compiler crea un fitxer nou que s'anomena igual que el fitxer `.java` original, però amb extensió `.class`. Gràficament seria:

Figura 1

Es podria dir que el codi *bytecode* és similar al **codi màquina** que els compiladors d'altres llenguatges de programació, com C/C++, generen. La diferència rau en el fet que el codi *bytecode* no està destinat per executar-se en una arquitectura de maquinari concreta (per exemple, Intel és diferent de Macintosh), sinó que està pensat per executar-se en una màquina virtual Java concreta (Java Virtual Machine, JVM). **Java Virtual Machine** (JVM) no és més que un intèrpret de *bytecode*. Això comporta que durant la fase d'interpretació hi hagi un alentiment en el rendiment del programa respecte al mateix programa escrit en llenguatge de màquina que generen els compiladors de llenguatges com C/C++.

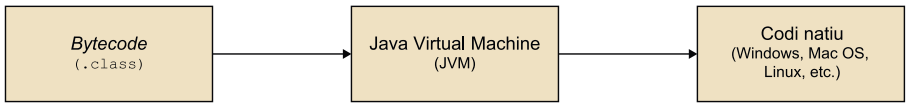
Bytecode

El *bytecode* no és entès per cap arquitectura de maquinari, sinó per la JVM. És menys complex (i més ràpid) per a la JVM interpretar *bytecode* que directament codi Java.

La JVM només depèn del sistema operatiu, però no de la màquina/ordinador en què s'executi. Així doncs, l'única cosa que s'ha de fer és tenir instal·lada la JVM del sistema operatiu corresponent. Gràcies a això, es desenvolupa una vegada en qualsevol dispositiu (fitxers `.java`), en aquest dispositiu es compila/transforma el codi Java en un codi estàndard i intermedi anomenat *bytecode* (fitxers `.class`) i, finalment, aquest codi s'executa en qualsevol dispositiu equipat amb la màquina virtual (JVM). Per aquest motiu, l'eslògan de Java és: «*Write once, run anywhere*».[4] Gràficament aquest procés seria:

(4)En català, «Escriu una vegada, executa a qualsevol lloc».

Figura 2



Així doncs, si s'executa l'arxiu `.class` en Windows, la JVM interpretarà el *bytecode* del fitxer `.class` i utilitzarà codi natiu que Windows entén per finalment executar el programa de manera adequada.

Java és compilat o interpretat?

Sempre trobem la discussió de si Java és compilat o interpretat, ja que hi ha totes dues fases. En termes generals, es diu que és interpretat per la manera en què la JVM executa el codi *bytecode*.

Perquè la JVM executi el *bytecode* d'un programa, cal cridar el fitxer `.class` l'homònim `.java` del qual tingués el mètode `main`. Aquesta crida es fa de la manera següent:

Mètode `main`

Més endavant, veurem què és el mètode especial `main` d'un programa.

```
java Program
```

4) JRE enfront de JDK

Independentment de la plataforma de Java, hi ha dos paquets diferents: JRE i JDK. Quina és la diferència?

El **JRE (Java Runtime Enviroment)** és un programari que proporciona el mínim indispensable per poder executar una aplicació Java en el dispositiu en el qual s'instal·la. Està format per la JVM i un conjunt de components (per exemple, llibreries d'integració, *toolkits* d'interfície d'usuari, etc.). El JRE ha de ser instal·lat per tothom que vulgui executar programes en Java, per exemple, un usuari final.

Per la seva banda, el **JDK (Java Development Kit)** és un programari de desenvolupament que permet crear i executar programes en Java. Inclou dos elements:

1) Un conjunt d'eines necessàries per al desenvolupament.
2) El JRE, el qual permet executar un programa Java.

El JDK ha de ser instal·lat per tothom que vulgui desenvolupar programes en Java.

1. Configuració de l'entorn

1.1. Instal·lant JDK

Com hem vist, quan volem programar en Java, hem d'instal·lar el JDK al nostre ordinador. En el moment d'escriure aquesta guia, la versió actual era la 13. Que la versió que hi hagi quan llegeixis aquesta guia en sigui una de més avançada no hauria de ser cap problema, ja que des de fa temps la manera d'instal·lar el JDK continua essent la mateixa.

Abans de veure els passos generals necessaris per instal·lar el JDK per a Windows, Linux i macOS, cal emfatitzar que des de la versió 11 del JDK, la llicència d'ús va canviar. Si abans era completament gratuït, des del JDK 11 la seva utilització només és gratuïta per a ús personal, però no per desenvolupar aplicacions per a tercers. Així mateix, el suport d'Oracle –com poden ser actualitzacions–, tampoc no és gratuït. És per això que des de la versió 11, moltes persones prefereixen instal·lar-se la versió lliure anomenada OpenJDK.

1.1.1. OpenJDK enfront d'Oracle JDK

La taula 1 destaca les similituds i les diferències entre OpenJDK i Oracle JDK.

Taula 1. Similituds i diferències entre OpenJDK i Oracle JDK

	OpenJDK	**Oracle JDK**
Oficialitat	És una implementació de l'especificació oficial de Java. Des del Java SE 7 és la implementació de referència utilitzada com a base per desenvolupar Oracle JDK.	És la versió oficial *de facto*.
Llançament de nova *release* (versió)	Cada sis mesos.	Cada tres anys es llança una versió LTS (*long term support*), per exemple, JDK 8 i JDK 11, que afegeix grans canvis i rep suport (per exemple, actualitzacions, correcció de *bugs*, etc.) durant vuit anys. Cada sis mesos apareix una versió no LTS que afegeix millores sobre l'última versió LTS, però aquestes millores no impacten en el *core*. Cada versió no LTS deixa de tenir suport als sis mesos, és a dir, quan n'apareix una nova versió.
Llicència	Lliure/gratuïta.	Gratuïta per a ús personal (per exemple, no comercial). De pagament per a ús comercial (per a tercers).
Actualitzacions (per exemple, correcció de *bugs*)	Cada sis mesos.	Immediat si es paga una llicència. En cas contrari, cada sis mesos.
Encarregat del manteniment	Comunitat Java, Oracle, Red Hat, IBM, etc.	Oracle Corporation.

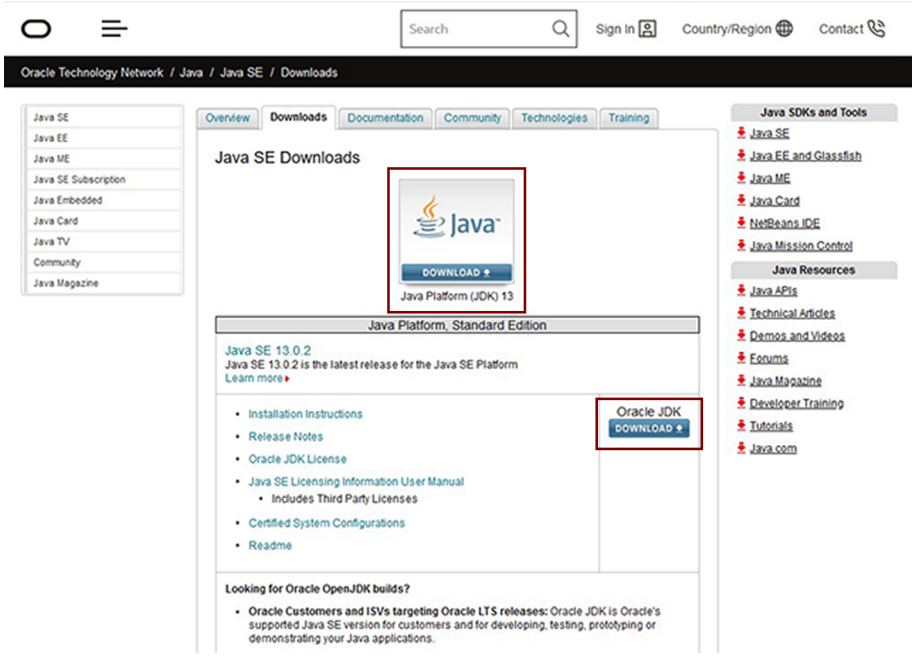
	OpenJDK	Oracle JDK
Instal·lació	Basat en un `.tar.gz` o `.zip`.	Té instal·ladors per a diferents sistemes operatius (msi, rpm, deb, etc.).
Rendiment	No hi ha diferències significatives.	

1.1.2. Oracle JDK per a Windows

A continuació, explicarem com instal·lar JDK 13 en Windows 10:

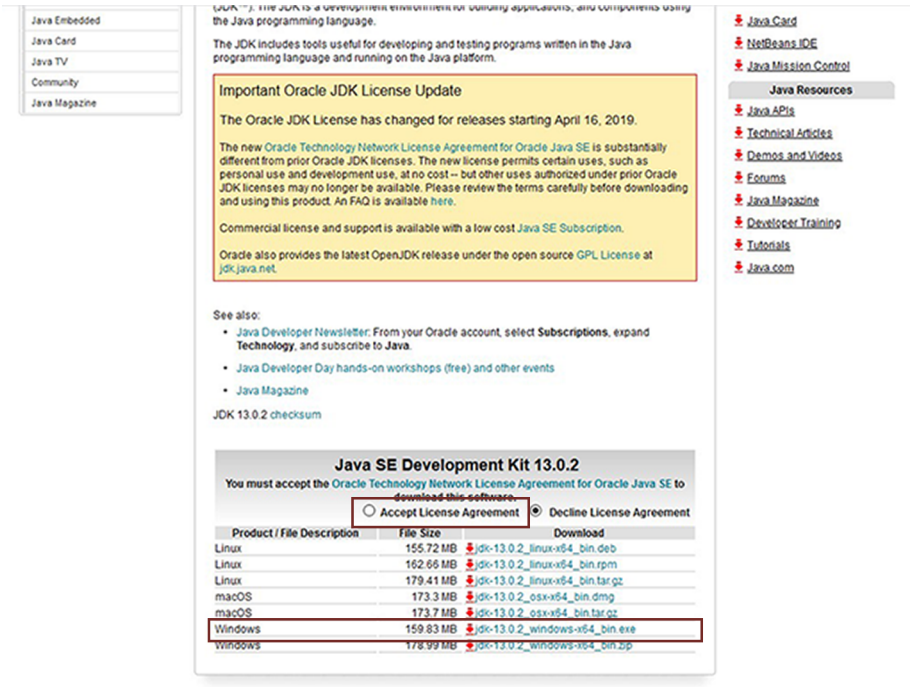
1) Descarregueu la versió de JDK 13.0.2 o superior. Sobretot, cal fixar-se que es tracta del JDK i no del JRE. Accediu per mitjà del navegador a l'adreça següent: http://java.sun.com/javase/downloads/index.jsp.

Figura 3



Feu clic en un dels dos botons de «Download» (ressaltats amb un requadre a la figura 3). A continuació, accepteu la llicència i trieu l'opció de Windows que vulgueu. Us recomanem que trieu l'opció executable (*.exe).

Figura 4



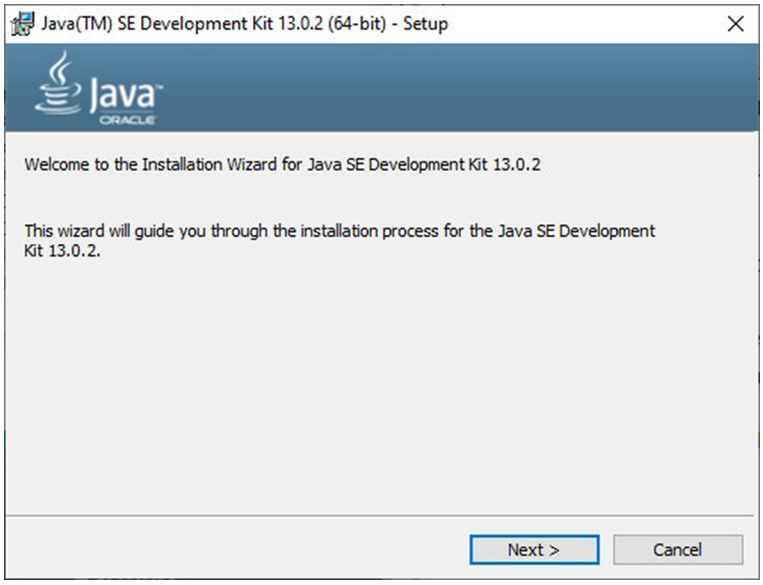
Nota

Si teniu Windows XP o un ordinador amb CPU de 32 bits, heu de descarregar la versió 8 (8u221), que té opció de 32 bits (x86): https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. En principi podreu fer totes les activitats inicials, però és important que aconseguiu un ordinador amb CPU de 64 bits i amb un sistema operatiu Windows 7 o superior.

2) Una vegada descarregat el programa d'instal·lació, **executeu-lo en mode administrador** (per exemple, botó dret del ratolí: «Executar com a administrador»). El procés d'instal·lació acaba de començar. Una pantalla de seguretat de control apareix per als usuaris de Windows 7 o superior. Si apareix, accepteu.

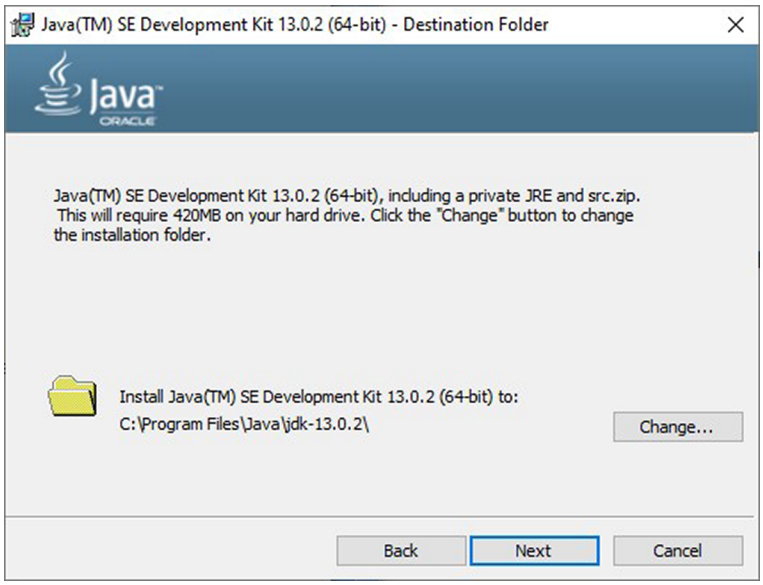
3) A continuació, us ha d'aparèixer la pantalla de benvinguda a la instal·lació; feu «Next».

Figura 5



4) Apareixerà una pantalla en la qual es demana la carpeta d'instal·lació. Deixeu la que us recomana l'instal·lador. Feu «Next».

Figura 6

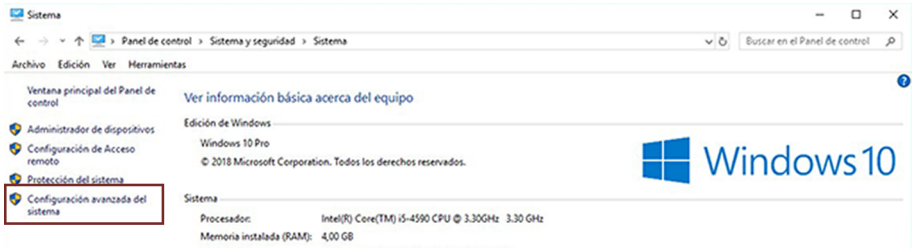


5) Una vegada instal·lat el JDK, apareixerà la pantalla següent, en la qual hi ha dos botons. Si premeu «Next Steps», s'obrirà la pàgina web https://docs.oracle.com/en/java/javase/13/ en la qual trobareu una gran quantitat de documentació. Si premeu «Close», es tancarà el Wizard d'instal·lació.

Figura 7

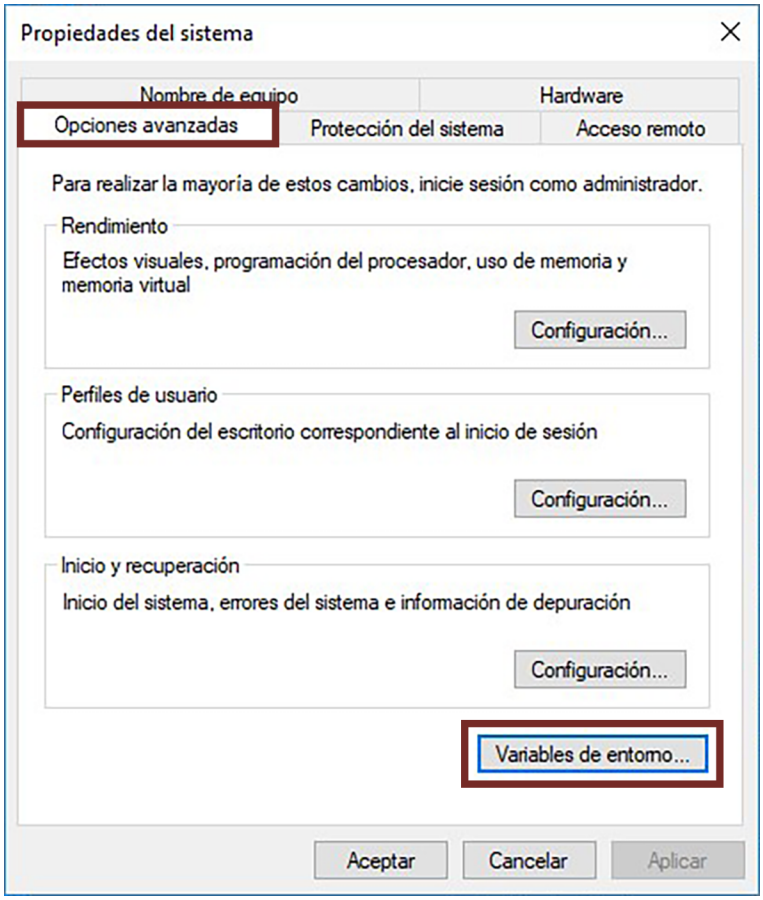
6) Després d'haver finalitzat la instal·lació, heu de configurar el PATH. Per a això, en Windows (qualsevol versió), cal que aneu al Tauler de control i trieu l'opció «Sistema i seguretat». A continuació, trieu «Sistema» i dins d'aquesta pantalla trieu «Configuració avançada del sistema» (està a la banda esquerra; vegeu requadre a la figura 8).

Figura 8



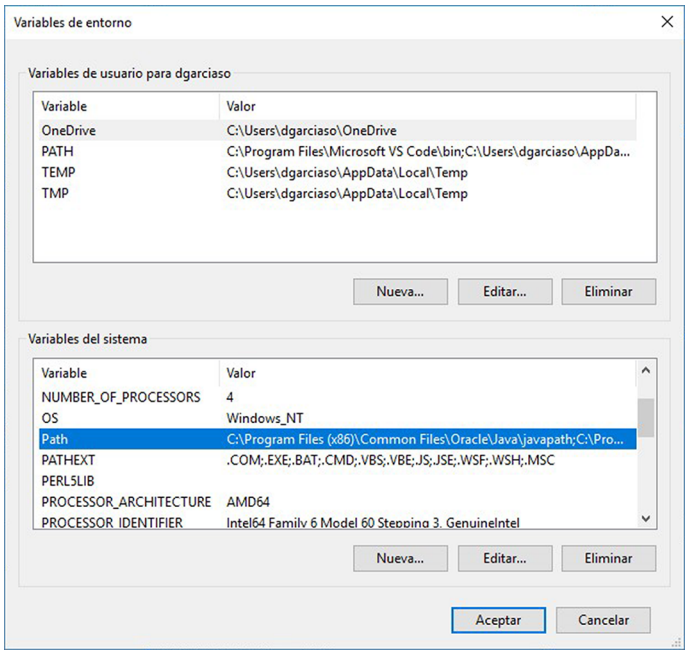
7) Dins d'aquesta finestra de configuració trieu la pestanya «Opcions avançades», en la qual trobareu un botó que diu «Variables d'entorn».

Figura 9



8) Si feu clic a «Variables d'entorn», apareixerà una finestra per modificar les diferents variables definides en el sistema. En la llista de sota busqueu i seleccioneu la variable *Path* (o PATH) i, a continuació, premeu el botó «Editar...».

Figura 10



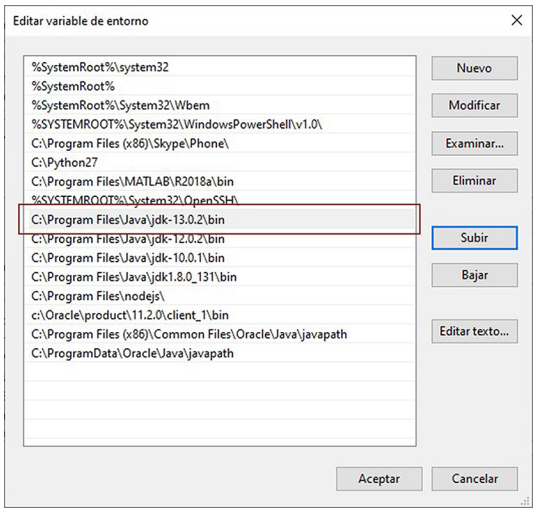
9) S'obrirà una finestra per modificar el valor de la variable *Path*. Seleccioneu la primera fila buida que hi hagi i escriviu:

```
JAVADIR\bin
```

`JAVADIR` és el directori on heu instal·lat el JDK; per exemple, si l'heu instal·lat a `'C:\Program Files\Java\jdk-13.0.2'`, llavors caldrà afegir:

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

Figura 11



Nota

Recordeu que el nom de la carpeta depèn de la versió instal·lada i la ruta d'instal·lació. Si teniu diverses versions de JDK instal·lades, és important que la versió que acabeu d'instal·lar sigui la primera de la llista. Per aconseguir que aparegui abans que qualsevol altra versió, utilitzeu els botons «Pujar» i «Baixar» del costat dret.

10) Finalment, per comprovar que JDK està instal·lat correctament, podeu consultar (per línia d'ordres, per exemple, `cmd`) les versions de `java` i `javac` mitjançant les instruccions:

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

Aquestes instruccions us haurien de retornar respectivament la versió de `java` (JRE) i `javac` (compilador) que es fan servir a l'ordinador. Ha de coincidir amb la que heu instal·lat. Si tot és correcte, veureu una pantalla semblant de la que es mostra a la figura 12.

Figura 12

Si tot i així no aconseguiu tenir instal·lada la versió 13 (o la que correspongui en el moment de llegir aquesta guia) de JDK, seguiu els passos que apareixen en el manual *Installation of the JDK on Microsoft Windows Platforms* (https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_OPENTOPIC_TOC_PROCESSING_d127e1987).

1.1.3. Oracle JDK per a Linux

Tenim dues maneres d'instal·lar JDK. La primera, per mitjà dels repositoris de paquets de Linux, i la segona, des del web d'Oracle.

Seguiu les instruccions que trobareu a la guia *Installation of the JDK on Linux Platforms* (https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_OPENTOPIC_TOC_PROCESSING_d127e1148).

1.1.4. Oracle JDK per a macOS

El sistema operatiu macOS ja porta per defecte la versió completa del JDK instal·lada i preparada per fer-la servir. En el cas que es tingui una versió vella del JDK i se'n vulgui instal·lar una de més nova, es pot aconseguir per mitjà de les actualitzacions de programari i el lloc web de suport d'Apple.

Si tot i així no aconseguiu tenir instal·lada l'última versió del JDK, seguiu els passos de la guia *Installation of the JDK on macOS* (https://docs.oracle.com/en/java/javase/13/install/installation-guide.pdf#_OPENTOPIC_TOC_PROCESSING_d127e1628).

1.2. Compilar i executar un programa per línia d'ordres

En primer lloc veurem com es compila un programa Java per línia d'ordres. És a dir, executarem el compilador del JDK perquè tradueixi de llenguatge Java a *bytecode*. O, el que és el mateix, crearem un fitxer `.class` a partir d'un fitxer `.java`.

Activitat

Feu la Tasca 1 de la PAC 1 que proporcionem amb aquesta assignatura.

1.3. Eclipse IDE

Si heu fet l'activitat proposada com a Tasca 1 de la PAC 1, us deveu haver adonat que podem escriure codi amb un editor de text senzill –per exemple, Notepad– i també compilar i executar programes Java per línia d'ordres fent servir `javac` i `java`, respectivament. No obstant això, quan els programes es fan més complexos, aquest entorn de desenvolupament no és pràctic. És per això que es van crear els IDE,[5] que faciliten el desenvolupament de programari. D'IDE, n'hi ha molts. Els més utilitzats en l'àmbit del llenguatge Java són Eclipse, NetBeans i IntelliJ. Tots són molt similars i fer-ne servir un o un altre dependrà de les preferències de cada desenvolupador. En aquesta guia optarem per Eclipse.

(5)Acrònim d'*integrated development environment*, entorn integrat de desenvolupament.

En primer lloc hem d'instal·lar Eclipse IDE. Per a això, hem de seguir els passos que detallem a l'apartat següent.

1.3.1. Instal·lació d'Eclipse IDE

La instal·lació d'Eclipse és bastant senzilla, només heu de seguir els passos que es detallen a continuació (per a sistemes operatius diferents de Windows, potser és més senzill baixar-se directament el fitxer comprimit de la modalitat d'*Eclipse IDE for Java Developers* i descomprimir-lo a l'ordinador). Alguns aspectes o noms poden canviar entre el que s'explica en aquest apartat i el que podeu trobar a l'entorn de treball. Si és així, heu de ser capaços de fer les analogies corresponents.

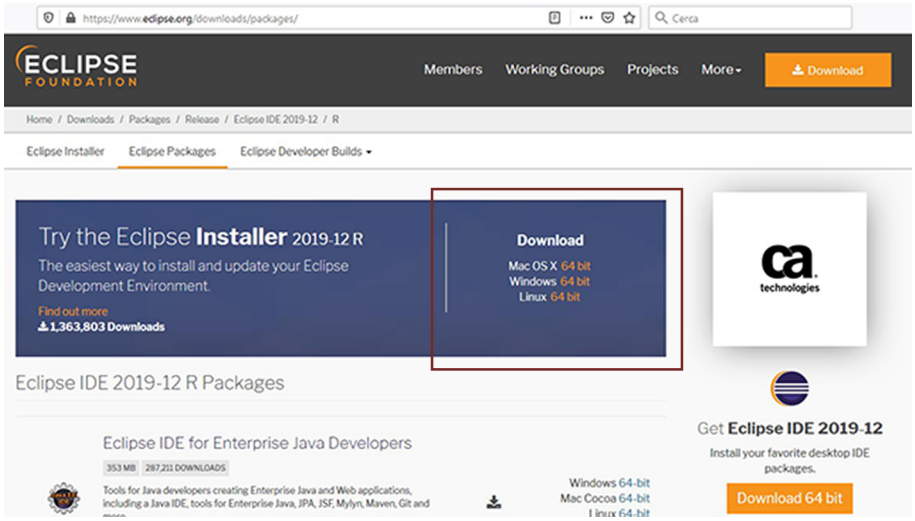
1) En primer lloc, heu de tenir instal·lada la versió JDK més actual que sigui possible.

2) Aneu al web de descàrregues d'Eclipse: https://www.eclipse.org/downloads/eclipse-packages/.

3) Un cop hi sigueu, farem servir la manera més senzilla d'instal·lar Eclipse. És a dir, mitjançant l'instal·lador Eclipse[6] (vegeu requadre de la figura 13). Fem clic a l'enllaç del nostre sistema operatiu.

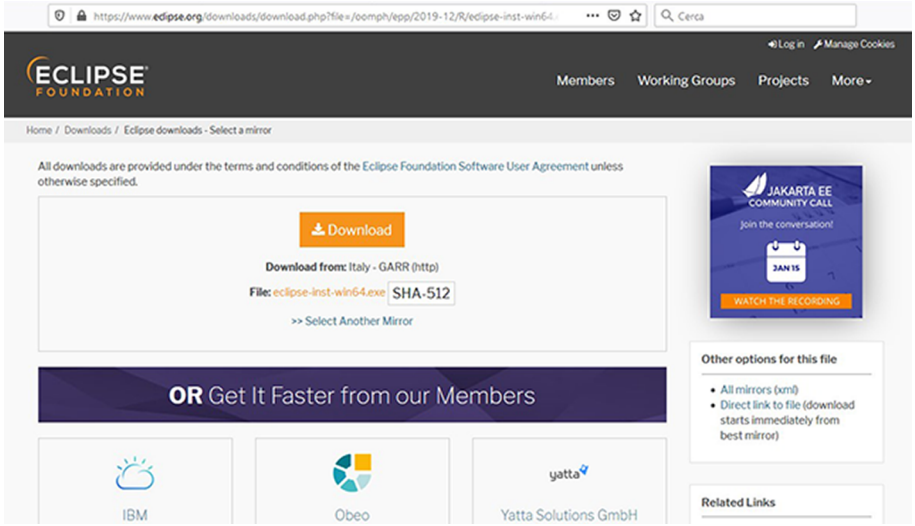
(6)En anglès, *Eclipse Installer*.

Figura 13



4) El web ens portarà a una altra pàgina amb un botó gran que diu «Download». Hi fem clic.

Figura 14



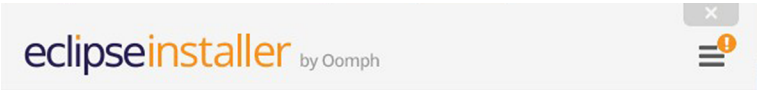
5) Desem l'executable que ens apareixerà.

6) Un cop tinguem l'executable en el nostre ordinador, hi fem doble clic per executar-lo. Recordeu executar-lo com a administrador (és a dir, en Windows, l'usuari amb el qual esteu autenticats en el sistema operatiu ha de tenir permisos d'administrador; si no, heu de clicar amb el botó dret l'executable i escollir l'opció «Executar com a administrador»).

7) Ens apareixerà una finestra amb diferents modalitats d'Eclipse. Abans de res és interessant (i important) actualitzar l'instal·lador, si és que cal fer-ho. Per saber si ens cal actualitzar-lo, mirem la icona amb dibuix de menú (tres ratlles) que hi ha a la part superior dreta. Si té una exclamació, el més segur és que

quan hi cliquem, ens aparegui l'opció «Update» amb exclamació. Si és així, fem clic a «Update» i esperem perquè s'actualitzi l'instal·lador (segurament es reiniciarà).

Figura 15



8) Quan tinguem l'instal·lador actualitzat i siguem a la pantalla de selecció de la modalitat d'Eclipse que cal instal·lar, escollim «Eclipse IDE for Java Developers». També ens serviria «Eclipse IDE for Java EE Developers», que és una versió més completa destinada per al desenvolupament d'aplicacions basades en Java EE. No obstant això, la versió Java EE ocupa més espai en el disc dur.

Funcionalitats d'Eclipse

Si necessiteu afegir alguna funcionalitat de Java EE al vostre «Eclipse IDE for Java Developers», podeu fer-ho instal·lant el *plugin* corresponent.

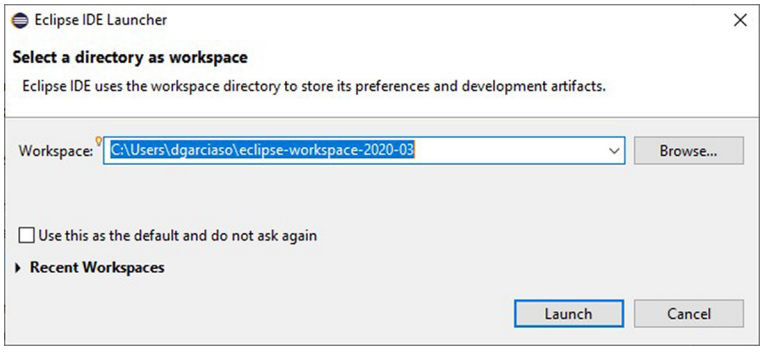
9) Una vegada triada la modalitat, ens apareixerà una finestra que ens demanarà on volem desar Eclipse, si volem una entrada en el menú d'inici i un accés directe a l'escriptori. A més, ens detecta la versió de Java que tenim per defecte en el sistema operatiu. En el camp «Product Version», escolliu la versió «Lastest». Quan tinguem les nostres preferències indicades, cliquem «Install».

10) Durant la instal·lació ens apareixerà una finestra per acceptar la llicència d'ús. Fem clic a «Accept». La instal·lació continuarà. També és possible que ens demani acceptar uns certificats. Els acceptem i la instal·lació seguirà el seu curs.

11) Si tot ha anat bé, ens apareixerà a la pantalla el botó «Launch» en verd. Fem clic en el botó «Launch» per executar Eclipse.

12) Després de carregar llibreries, ens apareixerà una finestra en la qual ens demana la ubicació (és a dir, la carpeta) que Eclipse ha de fer servir com a directori de treball, o *workspace*. És a dir, el lloc on desarà i buscarà els projectes (és a dir, els programes que creem), a més de desar les nostres preferències. Podem deixar la ubicació que ens suggereix o triar-ne una altra. També podem dir-li que la recordi i no ens la pregunti més (recomanem no marcar aquesta opció). Una vegada tinguem clar on situar el nostre directori de treball, fem clic en el botó «OK».

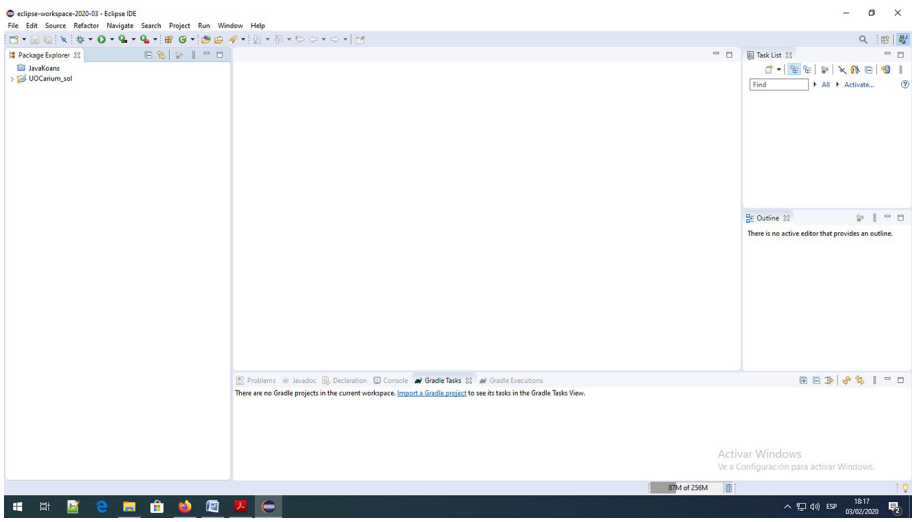
Figura 16



13) Eclipse continuarà carregant-se fins que, per fi, s'obrirà.

14) La primera vegada ens apareixerà una finestra de benvinguda. A continuació, tanquem aquesta finestra mitjançant la «X» que hi ha a dalt a l'esquerra al costat del nom «Welcome» i ens apareixerà la perspectiva Java.

Figura 17



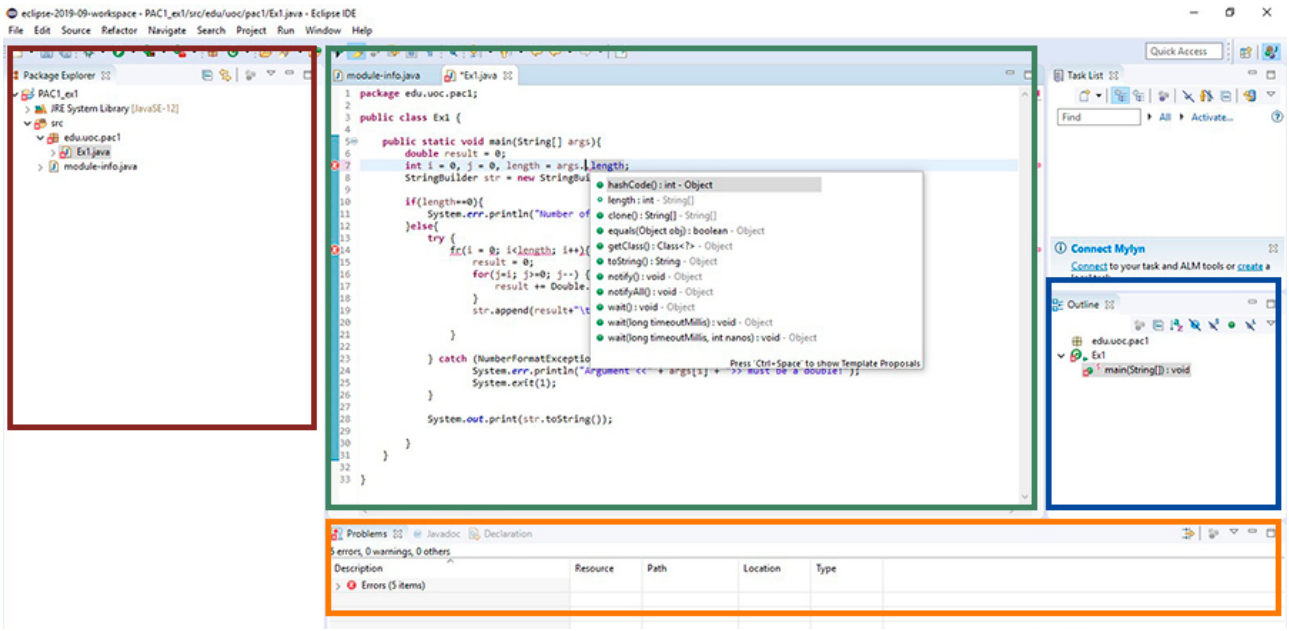
15) Si tanquem Eclipse, ens desarà la nostra configuració. D'aquesta manera, quan el tornem a obrir, Eclipse s'obrirà amb les mateixes finestres i perspectives que hi havia just abans de tancar-lo. Això serà especialment útil quan desenvolupem programes d'una certa mida i tinguem oberts diferents fitxers i perspectives.

A continuació, descriurem molt breument els elements més importants d'Eclipse.

1.3.2. Vistes

Cadascuna de les finestres/trossos que es veuen a la pantalla d'Eclipse és una vista. Cada vista està associada a una perspectiva; per tant, depenent de la perspectiva en què ens trobem, hi haurà vistes que no estaran disponibles. Totes les vistes tenen uns botons a la part dreta per maximitzar o minimitzar la finestra. Si fem doble clic sobre la finestra també es pot maximitzar. Quan maximitzem una vista, totes les altres vistes es minimitzen. Per tornar a la configuració normal, podem tornar a fer doble clic a la barra superior de la vista o bé clicar el botó situat a la part de més a la dreta de la barra superior de la vista, que té la forma de dues finestres sobreposades. En la perspectiva de Java, quan obrim per primera vegada un nou projecte, ens trobem amb les vistes següents:

Figura 18



Explorador de paquets (*package explorer*)

És la finestra que es troba a la part de més a l'esquerra (requadre granat de la figura 18). S'hi veuen tots els projectes que tenim en el nostre directori de treball. Per a cada projecte es veuen en forma d'arbre tots els fitxers que conté. Els petits triangles que apareixen al costat d'alguns dels ítems –com ara directoris, paquets, fitxers, etc.– indiquen que aquests elements es poden expandir. Si fem clic sobre el triangle, l'ítem s'expandeix i revela els elements que conté. En la perspectiva de Java es remarquen els subdirectoris que formen un paquet i els fitxers `.java` que conté cadascun dels paquets. Els fitxers Java es poden expandir i mostrar-nos les classes, els mètodes i els atributs que contenen. Si fem doble clic sobre algun fitxer, se'ns obrirà a l'editor, és a dir, a la part central de l'aplicatiu.

Editor

L'editor és la finestra/vista que apareix a la part central d'Eclipse (requadre verd de la figura 18). Tots els fitxers que s'obren, des de l'«Explorador de paquets» o des del menú «Fitxer», es mostren a l'editor. La part superior de la vista és una barra de pestanyes on se'ns mostren els noms dels fitxers oberts recentment. La pestanya activa correspon al fitxer que es visualitza en aquest moment a l'editor. En cas que obríssim molts fitxers, a la part dreta de la barra de pestanyes apareixeria el símbol >> amb el nombre de fitxers oberts i que no es veuen en aquesta barra. Si hi cliquem, se'ns desplegaran els altres fitxers que estan oberts.

La vista de l'editor és la que farem servir més durant la implementació de codi; per tant, sol ser una bona idea maximitzar la finestra de l'editor mentre treballem.

Quan ens trobem en la perspectiva de Java i editem un fitxer `.java`, en el codi se'ns ressalten les paraules clau amb color lila. Altres elements com els noms dels atributs o les cadenes de text se'ns mostren amb color blau. Els mètodes es poden ocultar clicant en el símbol menys (-) que apareix a la part esquerra de la línia de declaració del mètode. Si volem tornar a veure'n el contingut, només hem de clicar en el símbol més (+) que apareixerà al costat de la definició del mètode.

L'editor també té la funció d'anar marcant els errors de compilació que trobi mentre escrivim. Els errors surten marcats a la part dreta com a quadrats vermells; a la part esquerra, com a creus, i a la part de codi incorrecte, subratllats amb vermell. Si ens situem damunt d'alguna marca d'error, ens sortirà un missatge explicatiu del tipus d'error.

Una altra funcionalitat de l'editor és l'autocompleció de codi. L'autocompleció de codi ens permet inspeccionar el codi que es pot incloure on escrivim. Per exemple, si escrivim el nom d'una instància d'un objecte, podem inspeccionar els mètodes i atributs d'aquest objecte tan sols escrivint el punt per invocar les crides (missatges). Per a això escrivim el punt i esperem que ens surti la llista de possibles mètodes i atributs. Es pot cridar l'autocompleció de codi en qualsevol moment prement la combinació de tecles *Ctrl* i *espai*. Estiguem on estiguem, se'ns mostrarà una llista de suggeriments amb els possibles objectes o variables que tinguem definits.

Outline

La vista d'*outline* (requadre blau de la figura 18) ens permet visualitzar els components del fitxer Java que tinguem obert a l'editor, de la mateixa manera en què es visualitzen a l'explorador de paquets. És a dir, se'ns mostren les classes, els atributs i els mètodes que hi hagi al fitxer `.java` actual. Si es clica en qualsevol d'aquests elements en la vista d'*outline*, immediatament es ressalta a la finestra de l'editor. Els botons que hi ha al damunt de la vista ens permeten ocultar o mostrar diferents elements, com mètodes no públics, mètodes estàtics, atributs, etc. També ens permet ordenar els elements en la vista per ordre alfabètic.

Problems

Aquesta vista (requadre taronja de la figura 18) se sol trobar a la part inferior de l'editor i ens mostra els errors i avisos que tinguem en qualsevol fitxer dels projectes que tinguem oberts. Per a cada problema ens mostra una descripció i el lloc on es troba (fitxer i línia).

Console

Aquesta vista se sol trobar també a la part inferior de l'editor, en una altra pestanya al costat de la de problemes (requadre taronja de la figura 18). Aquesta vista s'activa automàticament quan s'executa una aplicació Java i serveix per mostrar la sortida del programa, i per demanar l'entrada de dades a l'aplicació si cal.

1.3.3. Menús

Les funcionalitats d'Eclipse estan disponibles per mitjà de la barra de menú general de l'aplicació situada a la part superior. El menú està dividit en diferents submenús: «File», «Edit», «Source», «Refactor», «Navigate», «Search», «Project», «Run», «Window» i «Help». Segons la perspectiva, aquests menús i el seu contingut poden variar.

També hi ha disponible un menú secundari al qual es pot accedir prement el botó dret del ratolí damunt de qualsevol selecció que fem sobre les vistes d'explorador de paquets o editor. Aquest menú contextual mostra les opcions de menú més comunes i adequades a la selecció que fem.

Menú «File»

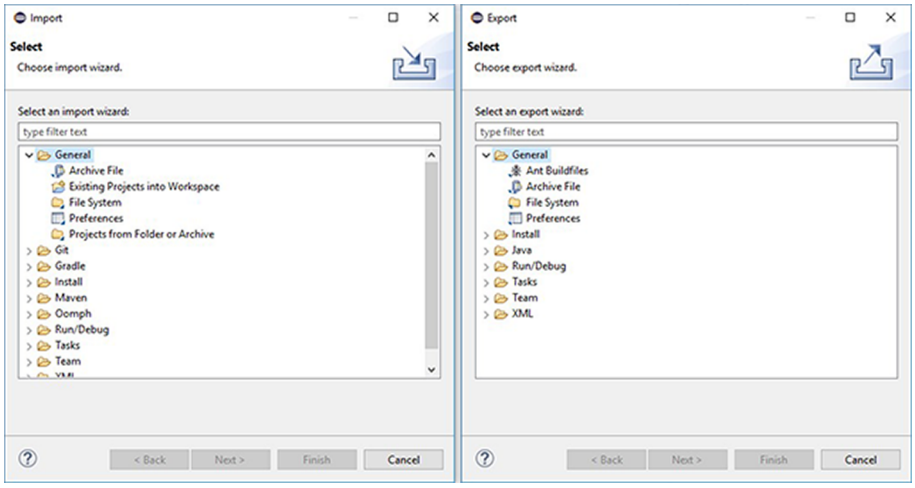
Des d'aquest menú podem accedir als assistents de creació de noves classes, projectes, paquets, interfícies i qualsevol altre element relacionat amb Java. Per a això, només cal obrir el submenú «New» i des d'allí seleccionar l'assistent que es vulgui.

Des del menú «File» tenim accés a totes les operacions sobre els fitxers del nostre projecte. Si estem situats sobre algun fitxer en la vista d'explorador de paquets, podrem moure el fitxer o canviar-li el nom. També hi ha l'opció de refrescar («**Refresh**») les carpetes de l'explorador. Aquesta funcionalitat es pot aplicar també per mitjà de la drecera de teclat *F5*. Tant si estem en aquesta vista o sobre l'editor, podrem tancar el fitxer seleccionat o desar-lo amb un altre nom.

D'aquest menú cal destacar també la possibilitat de canviar de directori de treball, seleccionant «Switch Workspace»; això ens permet tenir diferents directoris de treball en els quals podem desar els nostres projectes. L'últim directori de treball seleccionat serà amb el qual s'obrirà Eclipse la propera vegada que l'engeguem.

Finalment, mereixen una atenció especial els apartats d'importar i exportar del menú «**File**». Aquests assistents són extremadament importants per poder emportar-nos els nostres projectes Eclipse a altres màquines. Quan obrim aquests assistents se'ns mostra una llista de possibles ubicacions des d'on importar o a on exportar.

Figura 19



En l'apartat «General» trobem les eines més utilitzades de tots dos assistents. En el cas d'«Assistent d'importació» destaquem:

- L'opció «Existing Projects into Workspace» ens permet crear un projecte en el nostre directori de treball a partir d'un fitxer comprimit que contingui un projecte Eclipse o des d'un altre directori de treball. El projecte se'ns obrirà dins del nostre directori de treball i, si ho volem, podrem marcar l'opció «Copy projects into workspace» per desar-lo.

- L'opció «Archive File» ens permet desar fitxers situats en un fitxer comprimit dins d'un projecte Eclipse que ja existeixi en el nostre directori de treball. Aquests fitxers passaran a formar part del projecte seleccionat.

- L'opció «File System» ens permet desar fitxers que tinguem en alguna carpeta del nostre sistema dins d'un projecte Eclipse que ja existeixi en el nostre directori de treball. Aquests fitxers passaran a formar part del projecte seleccionat.

En l'apartat «General» de l'«Assistent d'exportació» destaquem les opcions següents:

- L'opció «Archive File» exporta un projecte del nostre directori de treball com un fitxer comprimit.

- L'opció «File System» exporta un projecte des del nostre directori de treball a un altre directori local de la nostra màquina.

Dins de l'assistent d'exportació cal destacar també l'apartat «Java». Tenim tres opcions:

1) L'opció «JAR File» permet generar un fitxer JAR de qualsevol dels nostres projectes. Un fitxer JAR és un contenidor de classes Java que permet tenir classes Java comprimides i agrupades en un únic fitxer. Això facilita l'exportació i la transmissió de programes/API entre programadors.

2) L'opció «Javadoc» que permet generar la documentació *Javadoc* de qualsevol dels nostres projectes, especificant les opcions que vulguem i el directori on desar la documentació generada.

3) L'opció «Runnable JAR file» crea un fitxer JAR autoexecutable, és a dir, com si fos un *.exe*.

Menú «Project»

El menú «Project» ens mostra opcions que ens permeten especificar com volem que es dugui a terme la compilació del projecte. L'última opció de totes és la de propietats del projecte («Properties»). Des de «Properties» es pot configurar qualsevol aspecte del projecte. Les opcions més destacades són:

1) «**Java Build Path**». Des d'aquest formulari podem especificar llibreries (per exemple, fitxers JAR) i codi que volem afegir al nostre projecte. Aquest formulari és el mateix que aquell al qual es pot accedir des de l'assistent de projecte nou. El «Build Path» és la manera que té Eclipse de configurar el *classpath* de la màquina virtual de Java; és, per tant, molt important que el «Build Path» estigui molt ben configurat, ja que si no, tindrem errors de compilació.

En l'apartat de llibreries sempre surten per defecte com a mínim les llibreries de sistema de la màquina virtual que fem servir. Si s'edita aquesta llibreria tindrem l'opció de canviar-la per qualsevol altra de les màquines virtuals de Java que tinguem en el nostre ordinador. Es poden afegir altres llibreries que facin falta en el nostre projecte des de fitxers JAR o des de directoris. A més, hi ha les opcions d'editar i esborrar llibreries ja incloses. En l'apartat «Source» surt per defecte el directori on hi ha el codi del projecte.

2) «**Java Compiler**». Des d'aquest formulari podem escollir el grau de compatibilitat del nostre codi. El grau de compatibilitat (*JDK Compliance*) significa fins a quina màquina virtual ha de ser vàlid aquest codi. Per exemple, si posem compatibilitat fins a la màquina virtual 1.4, tots els elements sintàctics propis de la màquina virtual 1.5, com els genèrics o les enumeracions, que hi hagi en el nostre codi donaran error. Per defecte s'agafen les propietats generals especificades en el nostre directori de treball. Es pot canviar la configuració per a un projecte en concret si activem la casella «Enable project specific settings».

Menú «Run»

Des d'aquest menú podem executar el nostre codi, gestionar diferents configuracions d'execució del projecte i cridar el *debugger*.

L'opció «Run» executarà el projecte que tinguem seleccionat en l'editor o en l'explorador de paquets. Buscarà el mètode `main` i l'executarà. Els possibles resultats de sortida per consola es veuran en la vista «Console». En cas que no hi hagi cap mètode `main`, sortirà un avís que ens informarà del fet que la selecció no conté cap mètode que es pugui executar.

Altres menús

En el menú «Window» trobem opcions per configurar l'entorn de finestres com vulguem. Podem obrir vistes, canviar de perspectiva i configurar-la. També tenim l'opció de restaurar una perspectiva en el seu estat per defecte, així com desar-la o tancar-la.

En el menú «Search» hi ha les opcions per buscar cadenes de text dins del nostre codi. A més, es poden fer altres tipus de cerques, com trobar les referències o declaracions d'una variable. Només cal seleccionar en l'editor la variable que vulguem buscar i triar l'opció adequada en el menú de cerques.

En els menús «Source» i «Refactor» hi ha tot un seguit d'opcions per tal d'automatitzar certes tasques repetitives a l'hora de desenvolupar una aplicació. En el menú «Source» podem trobar utilitats com afegir blocs de comentaris, generar *setters* i *getters* automàticament, afegir de manera automàtica els *imports* que facin falta en el fitxer, crear la indentació del codi de manera automàtica, etc. En el menú «Refactor» tenim utilitats per propagar canvis en el codi sobre tots els fitxers `.java` implicats, com, per exemple, canviar el nom d'una classe o d'un mètode, i canviar totes les referències i declaracions que hi hagi en el codi. Des d'aquí també podem moure una classe d'un paquet a un altre i propagar els canvis que això produeixi en les altres classes del projecte.

2. Fonaments de Java (com a llenguatge imperatiu)

2.1. Comentaris dins del codi

Els comentaris s'utilitzen per documentar i explicar el codi que realitzem. Els comentaris són ignorats pel compilador, però són importants per explicar la lògica del programa (i algunes decisions preses) a tercers (i a un mateix en el futur).

Cal dir que en Java hi ha dues maneres d'escriure comentaris en el codi:

Comentari d'una línia

```
//Comentari d'una línia
```

Comentari de múltiples línies

```
/* Comentari
de múltiples
línies
*/
```

A més de servir per documentar, els comentaris també serveixen per «capar» o «inhabilitar» una part del nostre codi. Així doncs, en comptes d'esborrar un tros de codi per sempre, podem comentar-lo i, si calgués, recuperar-lo més tard. Una vegada sapiguem que aquest codi és descartable, podem esborrar-lo per sempre.

Això és útil, per exemple, quan provem coses. També és útil si estem encallats escrivint un algorisme complex, però tenim un codi base creat que sabem que funciona. Aquest codi base podria ser comentat per recuperar-lo més tard si en escriure l'algorisme arribem a un atzucac i hem de començar de nou. D'aquesta manera, el codi base serveix com a punt de partida des d'on iniciar de nou l'algorisme.

2.2. Estructura bàsica d'un programa

Com ja sabem, Java és un llenguatge pensat per al paradigma de la programació orientada a objectes (POO). De moment no entrarem en conceptes de POO, però irremeiablement, per la naturalesa de Java, apareixeran des del començament de les explicacions.

Com a conseqüència d'això, alguns elements de POO apareixen quan analitzem el codi mínim per crear un programa en Java.

ClassName.java

```
/*
Aquest codi es pot considerar una plantilla per començar a programar en Java.
Com podeu veure, utilitzem un comentari de múltiples línies per documentar.
*/
public class ClassName{ // Aquest és el nom de la classe, trieu-ne un de representatiu.
    //El main és un mètode/funció especial que fa de punt d'entrada del programa
    public static void main(String[] args){
        //TODO: Aquí, hi va el codi!!
    }
    //TODO: Aquí, hi podria anar més codi en forma de funcions (també anomenats mètodes)
}
```

Analitzem el codi anterior:

1) Atès que Java està orientat a objectes, els programes en Java estan organitzats per classes. **Cada classe ha d'anar en un fitxer el nom del qual ha de ser el mateix que el de la classe**. Per una qüestió de convenció de noms, el més habitual és que la primera lletra de cada paraula que compon el nom de la classe comenci amb majúscula. En l'exemple, la classe es diu `ClassName` i el fitxer, `ClassName.java`.

2) Un programa d'una certa envergadura estarà format per més d'una classe (per exemple, més d'un fitxer `.java`). Això obliga al fet que, com a mínim en una d'aquestes classes, hi hagi un mètode especial anomenat `main`. Aquest mètode serveix de punt d'entrada al programa. És a dir, el programa s'iniciarà en cridar aquest mètode. El mètode `main` té una signatura concreta:

Ús de *varargs* en el `main`

Com veurem més endavant, `String[] args` es pot canviar per `String...args`, gràcies a la nova sintaxi de les *varargs*.

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

3) El mètode `main` no retorna res (per això el seu tipus de retorn és `void`) i rep un *array* de `String` (és a dir, una cadena de caràcters) per paràmetres anomenat `args` (el nom es pot canviar, però el més habitual és anomenar-lo així, procedent de l'abreviatura d'*arguments*). Aquest paràmetre `args` emmagatzema els valors passats per línia d'ordres (és a dir, arguments) quan el programa s'executa. Com en el codi anterior el mètode `main` es troba a la classe `ClassName` i farem:

```
java ClassName David 36
```

4) En aquest cas, `args` tindria dues caselles amb els valors `"David"` i `"36"`, tots dos serien text (és a dir, `String`). El primer valor estaria en `args[0]` i el segon, en `args[1]`.

5) Ara com ara oblidem-nos de la paraula reservada `static`. En veurem el significat tant en aquesta guia com en els materials teòrics de l'assignatura. Això sí, si creem més funcions dins del fitxer `ClassName`, **ara com ara** (i només ara com ara), en no treballar orientat a objectes (que és per a la qual cosa

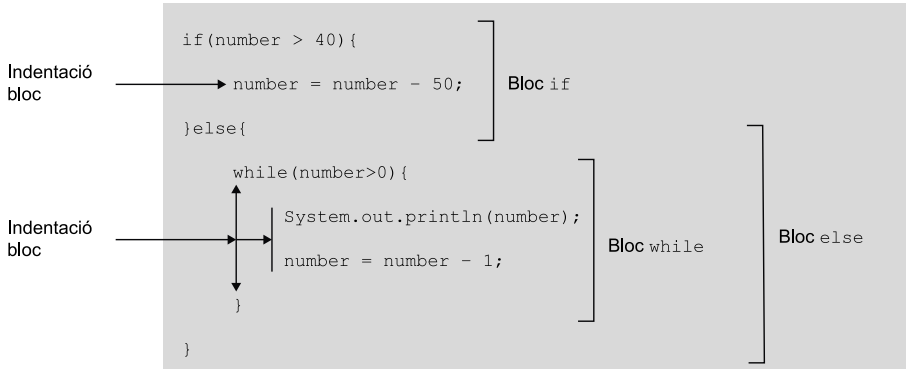
està concebut Java), **les signatures de totes les funcions hauran de tenir la paraula `static`**. Quan passem a programar orientat a objectes, veurem que aquest requisit ja no serà necessari.

6) A continuació dels dos comentaris, on posa `TODO` (de l'anglès, *to do*, és a dir, «per fer», «pendent», etc.) podríem escriure el codi del nostre programa. El codi del primer `TODO` estarà format per instruccions/sentències (*statements*) i blocs (*blocks*).

- **Instrucció/sentència**: és la unitat mínima de programació encarregada de realitzar una acció. Cada sentència ha d'acabar amb un punt i coma (`;`).

```
int number = 50; //declara una variable de tipus int anomenada
//number i la inicialitza amb el valor 50
number = number * 3; //multiplica per 3 el valor de number i
//l'assigna a number
System.out.println(number); //imprimeix per pantalla el valor de
//number
```

- **Bloc**: és un grup de sentències agrupades entre dues claus `{ }`. Totes les sentències situades entre dues claus són tractades com una única unitat. Dins de dues claus pot ser que no hi hagi cap instrucció.



7) A l'hora d'escriure el nostre codi hem de tenir present algunes convencions d'estil de format:

- No importa afegir gaires espais, tabuladors, *enters* (és a dir, noves línies en blanc) entre dues paraules, ja que Java els tracta com un únic espai/tabulador/nova línia.

- És important indentar (és a dir, sagnar) el codi amb tabuladors i espais en blanc, i també fer servir de manera adequada els espais en blanc entre línies per tal de facilitar la lectura del codi.

- En el cas de les claus, el més habitual en Java és posar la primera clau (`{`) al final de la primera línia. En el cas de l'última clau (`}`), aquesta es posa en

una línia després de l'última sentència i alineada amb la primera sentència del bloc (mireu el codi anterior).

2.3. Entrada per teclat i sortida per pantalla (I/O)

Java té tres *streams* (fluxos) d'entrada i sortida:

1) Entrada estàndard (*standard input stream*). Per defecte es refereix al teclat, encara que es pot redirigir a una altra font d'entrada, per exemple, un fitxer de text. Per accedir a aquest flux farem servir `System.in`.

2) Sortida estàndard (*standard output stream*). Per defecte, Java utilitza la pantalla de la consola com a sortida, però es pot canviar, per exemple, a un fitxer de text. Per accedir a aquest flux farem servir `System.out`.

3) Sortida d'error (*error output stream*). Aquest *stream* s'utilitza per mostrar missatges d'error o amb informació important. Per defecte fa servir la pantalla de la consola, encara que és molt freqüent redirigir-lo cap a un fitxer de *log*. Per accedir a aquest flux utilitzarem `System.err`.

La classe `System` de l'API Java ens proporciona accés a tres atributs públics i estàtics anomenats `in`, `out` i `err`. Tant `System.in` com `System.out` i `System.err` són instanciats i inicialitzats automàticament per la JVM quan aquesta engega, així que no cal fer res per tenir-los disponibles.

Nota

En aquest apartat només veurem els mètodes bàsics de l'entrada i la sortida estàndards, i obviarem la sortida d'error.

2.3.1. Mètodes de sortida

La classe `System` de l'API Java ens proporciona accés a un atribut públic i estàtic anomenat `out` de tipus `PrintStream` (una altra classe de l'API de Java). Aquest atribut, que és una referència a un objecte de la classe `PrintStream`, ens permet accedir als mètodes públics d'aquesta classe, la qual s'encarrega d'escriure dades en el flux de sortida que s'indiqui. Entre ells, destaca: `print`, `println` i `printf`.

Mètodes `print` i `println`

Aquests dos mètodes són els més freqüents. La diferència entre ells és que `println` imprimeix el contingut que se li passi i afegeix un salt de línia (`\n`). És a dir, avança el cursor per escriure a la línia següent de la pantalla. Sabent això, el codi següent:

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

Imprimiria:

```
58
5
```

Com veiem, el `8` s'imprimiria a la mateixa línia que el primer `5`, perquè el `5` ha estat imprès amb un `print`.

Mètode `printf`

Els dos mètodes anteriors no permeten especificar un format per controlar aspectes com, per exemple, el nombre de decimals que cal mostrar per a un valor de tipus `double`. El mètode `printf` rep, com a mínim, una cadena de text que indica el text juntament amb el format que cal mostrar. A continuació, si és necessari, rep tants valors (en forma de literals o variables) com calgui per emplenar el text. Vegem diversos exemples per entendre'n el funcionament:

```
System.out.printf("Hola! %d %.2f", 53, 89.7); //Hola! 53 89.70 -> afegeix un 0
//perquè 87.7 tingui dos decimals
System.out.printf("Hola! %03d", 53); //Hola! 053 -> afegeix un 0 perquè 53
//tingui 3 dígits, si no posem el 0 en %03d imprimiria un espai en blanc
```

La manera d'indicar en el text de sortida on volem un format especial és mitjançant l'ús d'un comodí que té un format determinat: `%[flags][amplada]codiConversió`. Tant els *flags* com l'amplada són dos valors opcionals. Els *flags* serveixen per indicar aspectes com l'alineació del text i l'element que cal afegir com a *padding* (farciment), entre altres. Per la seva banda, l'amplada és un valor que indica quant d'espai ha d'ocupar l'element com a mínim. Per acabar, el codi de conversió serveix per indicar el tipus del valor:

- `d` per a enter,
- `f` per a `float` o `double`,
- `c` per a `char` i
- `s` per a `String` (cadena de text).

Vegem-ne alguns exemples:

```
System.out.printf("%s|%6d|%.2f","eps",53,89.7); //eps|   53|89.70 -> el número
//53 ocupa una amplada de 6 caràcters.
System.out.printf("Hola|%-4d|", 53); //Hola|53 | -> 53 ocupa 4 posicions i està
//alineat a l'esquerra (ho indiquem amb un - ).
```

Respecte al salt de línia, aquest mètode funciona exactament igual que `print`, és a dir, no avança el cursor a la línia següent. Si volem fer un salt, hem d'afegir `\n` al final del text.

2.3.2. Mètodes d'entrada

Com ja hem dit, per defecte, el flux d'entrada ve del teclat. En JDK 5 es va introduir una classe anomenada `Scanner` que facilita la lectura amb format del teclat. D'aquesta manera, amb `Scanner` podrem controlar, de manera senzilla, si volem llegir un `int`, un `double`, etc.

```
Scanner in = new Scanner(System.in); //Objecte Scanner que llegeix de teclat
int number = in.nextInt(); //Espera rebre un int, en cas contrari llança una
//excepció
double decimal = in.nextDouble(); //Espera rebre un double, en cas contrari
//llança una excepció
String word = in.next(); //Llegeix un text fins a arribar al primer espai
String text = in.nextLine(); //Llegeix un text fins a arribar al primer salt de
//línia (inclou els espais en blanc)
in.close(); //Tanquem el flux (stream)
```

2.4. Variables

Una **variable** es defineix com una unitat bàsica d'emmagatzematge. Es pot veure com una etiqueta que permet desar i recuperar una dada de memòria. Aquesta dada serà d'un tipus. Així doncs, una variable tindrà un nom i un tipus, i desarà un valor.

S'anomena *variable* perquè el valor que emmagatzema pot variar (és a dir, canviar) durant l'execució del programa.

2.4.1. Nom

Els noms de les variables han de ser representatius d'allò que emmagatzemen.

Imagineu que tenim tres variables amb els tres noms següents: `n`, `p1` i `vC`. Sabeu què emmagatzemen? Ni idea, oi? Així doncs, aquests noms tan críptics no són adequats, encara que sí que són correctes sintàcticament (és a dir, Java no donarà error). No obstant això, si canviem els noms de les variables anteriors per aquests altres: `numberPeople`, `person1` i `viewsCount`, ara sí que podríeu saber què desen. Com podeu veure, escollir un nom que sigui descriptiu és una bona pràctica.

A l'hora d'assignar un nom a una variable hem de tenir present que:

- Els noms de les variables són *case-sensitive*. És a dir, Java distingeix entre minúscules i majúscules, per la qual cosa `numPeople` i `numpeople` no són la mateixa variable.

- El nom d'una variable ha de començar amb una lletra, o amb el símbol `$`, o amb el símbol *underscore* _. No obstant això, s'ha generalitzat la bona pràctica que el nom de les variables comencin per una lletra, no per `$` ni _.

- El nom d'una variable pot tenir qualsevol longitud i, a partir de la primera lletra, pot estar format per qualsevol combinació de lletres Unicode, dígits, i els símbols `$` i *underscore* _.

- Per escriure el nom d'una variable en Java se segueix la convenció anomenada ***lower camel case*** (o *dromedary case*). Aquesta convenció indica que la primera paraula s'escriu en minúscula i, si el nom de la variable està compost per dues o més paraules, a partir de la segona paraula, la primera lletra de cadascuna d'elles s'escriu en majúscula, p. ex., `lastName` (està composta de `last` + `name`), `resourceNumber`, `yTopLeft`, `xMin`, `height`, etc.

- El nom que s'assigni a una variable no pot coincidir amb un dels literals següents: `true`, `false` i `null`.

- Tampoc no es poden fer servir com a noms de variable algunes de les paraules següents reservades de Java (també anomenades *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+)
```

2.4.2. Tipus

Com hem comentat, les variables seran d'un tipus. En Java, hi ha dos tipus: primitius i referències. Els **primitius** (o bàsics) serveixen per emmagatzemar valors senzills, mentre que les **referències** s'utilitzen per desar elements complexos, com ara els objectes d'una classe.

Primitius

En Java hi ha vuit tipus primitius o bàsics.

Taula 2. Primitius de Java

Tipus	*Bytes* necessaris per desar el valor	Interval/rang (valors possibles)	Valor per defecte	Utilitat	Exemples
boolean	1 bit	`true o false`	`false`	Desar valors dicotòmics «cert» i «fals».	`true`
byte	1 (8 bits)	[-128, 127]	`0`	Desar dades petites.	56, 83
short	2 (16 bits)	[-32768, 32767]	`0`	Mateix ús que *byte*, però la seva mida és el doble.	32769, 0xffea
char	2 (16 bits)	`['\u0000', '\uffff']` (internament es desa com un enter [0, 65535])	`'\u0000'`	Desar un símbol de la codificació Unicode.	'a', '?', '\101', '\n'
int	4 (32 bits)	[-2147483648, 2147483647]	`0`	Desar valors enters.	-2, -1, 0, 1, 234, 500
long	8 (64 bits)	$[-2^{63}, 2^{63}-1]$	`0`	Desar valors enters que no caben dins del rang de l'`int`.	-2L, -1L, 0l; 1L, 2L, -64323
float	4 (32 bits)	Estàndard per a aritmètica en coma flotant de 32 bits (IEEE 754).	`0.0`	Desar valors decimals amb 6-7 decimals significatius.	1.23e100f, -0.23e-100f, .5f
double	8 (64 bits)	Estàndard per a aritmètica en coma flotant de 64 bits (IEEE 754).	`0.0`	Desar valors decimals amb 15 decimals significatius.	1.23456e300d, 1e1d

Unicode

El codi Unicode és un estàndard de la informàtica per representar text. Inclou alfabets, símbols (per exemple, $, &, ?, @) i elements especials (per exemple, £). Unicode inclou més de 110.000 caràcters.

Per declarar una variable seguirem un dels patrons següents (i combinacions d'ells):

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

Per exemple:

```
float grade;
int numberPeople = 120360;
byte age = 36; //es podria declarar int, però l'edat d'una persona no serà
//més gran que 127. Amb un byte ocupem menys espai en memòria (1 byte) que amb
//un int (4 bytes).

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

Com es pot veure en l'exemple anterior, en el moment de la declaració es pot aprofitar per inicialitzar la variable amb un valor. Si no s'indica un valor, el compilador li assigna el valor per defecte del tipus amb el qual s'ha declarat. En el cas de la variable `grade`, aquesta serà inicialitzada automàticament amb el valor `0.0`, mentre que la variable `numStudents` serà inicialitzada amb el

valor `0`. Així mateix, es pot veure com podem declarar més d'una variable d'un mateix tipus alhora, combinant al mateix temps la inicialització o no d'aquestes variables. Hi ha alguns programadors que desaconsellen declarar les variables d'aquesta manera i prefereixen que cada variable sigui declarada individualment. És a dir, canviaríem l'última línia de l'exemple anterior per:

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

D'altra banda, en el cas de dades numèriques, per facilitar-ne la lectura, Java permet separar-les mitjançant el símbol _ (*underscore*), per exemple, `123_456`, `12_345`, `1_3` (seria el número 13), etc. És justament el que s'ha fet en la variable `viewsCount`.

Així mateix, els literals de nombres enters es poden expressar en binari (base 2), octal (base 8), decimal (base 10) o hexadecimal (base 16). Per exemple:

```
int number = 0b01001011; //número 75 en decimal. S'ha d'anteposar 0b o 0B
//(zero b) al número binari.
int number = 01144; //número 612 en decimal. S'ha d'anteposar un 0 (zero) al
//número en octal.
int number = 1983; //número 1983 en decimal. És la manera habitual de
//representar els números
int number = 0xFFFF; //número 65535 en decimal. S'ha d'anteposar 0x o 0X
//(zero X) al número en hexadecimal.
int number = 0xFFF_F; //amb els literals numèrics podem utilitzar sempre el
//underscore _ independentment del seu format.
```

En aquest punt, cal tenir present una apreciació amb els tipus `long` i `float`. Vegem-ne l'exemple següent:

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

Les dues declaracions anteriors llançaran un error que indicarà que el valor que es vol assignar a `year` i a `price` és incompatible. Com és això possible si estan dins del rang/interval de valors? Senzillament perquè el compilador de Java considera els valors `1983` i `9.99` com a valors de tipus `int` i `double`, respectivament. Així doncs, per forçar que els consideri `long` i `float`, respectivament, hem d'afegir la lletra `L` en el primer cas, i la lletra `F`, en el segon. En tots dos casos, la lletra pot ser majúscula o minúscula:

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

A continuació, vegem un cas de variable de tipus `char` i un altre `boolean`.

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

Les variables de tipus `char` desen un únic caràcter. Aquests valors es delimiten amb una cometa simple. En el cas de les variables de tipus `boolean`, els valors que poden acceptar són `true` o `false`.

JDK 10 i posterior

Des de JDK 10 és possible no indicar el tipus (primitiu o referència) d'una variable local mitjançant la paraula reservada `var`. Una restricció de fer servir `var` és que la variable ha de ser inicialitzada perquè el compilador pugui inferir el tipus de la variable, per exemple: `var v1 = 5;`.

Quan declarem una variable de tipus primitiu, allò que fa el compilador és assignar una adreça de memòria de l'*stack* a aquesta variable. Nosaltres, com a programadors, no tenim control de l'adreça de memòria en què es crea aquesta variable.

Contingut	Adreça de memòria
36	1000
	1004
	1008
	1012

```
int age = 36;
```

En l'exemple anterior, la variable `age` es troba a l'adreça de memòria `1000`. En el nivell de memòria, el nom de la variable desapareix i passa a ser l'adreça `1000`. És a dir, a baix nivell no hi ha noms de variables, sinó adreces de memòria. Per tant, gràcies als llenguatges d'alt nivell com és el cas de Java, el programador no treballa directament amb adreces de memòria, la qual cosa facilita el desenvolupament de programari. Així doncs, quan un programador escriu el codi següent després de la declaració anterior:

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

El compilador tradueix aquesta instrucció de manera que va a l'adreça de memòria `1000`, recupera el valor que hi ha en aquesta posició (és a dir, `36`), li suma `2` i desa a l'adreça de memòria `1000` el valor `38`.

Referències

Totes les variables que no són declarades com un dels vuit tipus primitius anteriors, són referències. En l'exemple següent, declararem un *array* d'`int`.

```
int [] grades; //també es pot fer així: int grades[]; però és preferible
//l'opció int[].
```

La manera de declarar un *array* és mitjançant `[]`. La manera d'inicialitzar una referència no és assignant un valor com en el cas dels tipus primitius, sinó per mitjà de l'operador (i paraula reservada) `new`.

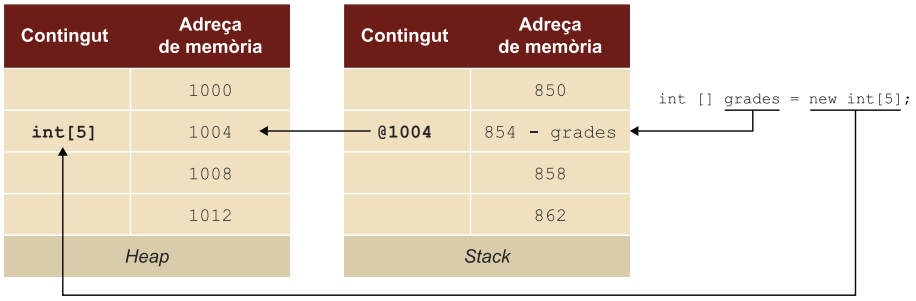
```
int [] numbers = 5; //error: no és un tipus primitiu, sinó una referència
int [] grades = new int[5]; //OK: dins de [] indiquem la mida de l'array.
/Les 5 caselles de l'array són inicialitzades amb el valor per defecte del tipus
//primitiu, en aquest cas, 0.
```

A l'última línia del codi anterior declarem un *array* anomenat `grades` amb cinc caselles de tipus primitiu `int`.

L'operador `new` crea un objecte en una adreça de memòria concreta del *heap* i aquesta adreça de memòria l'assigna a la referència, la qual es desa en una zona de memòria de l'*stack*. Així doncs, el valor d'una referència serà una adreça de memòria.

Stack i heap

Els conceptes de *stack* i *heap* s'estudien en profunditat en assignatures relacionades amb els sistemes operatius.



Així doncs, `grades` és una variable de tipus referència situada en la posició `854` de l'*stack* i el valor de la qual és l'adreça de memòria (valor `1004`) d'un objecte *array* de cinc caselles de tipus `int` que està en el *heap*.

Stack enfront de *Heap*

Quan un programa s'executa, la JVM crea un fil d'execució (en anglès, *thread*) amb un *stack* privat. En l'*stack* es desen variables locals, variables de referència, el control de les invocacions a mètodes, els valors dels paràmetres i de retorn d'un mètode, resultats parcials, etc. Aquest espai de memòria es regeix per un comportament de pila (*stack*), és a dir, LIFO (acrònim de *Last-In, First-Out*).

En canvi, el *heap* és un espai de memòria dinàmica que la JVM adquireix tot just engegar-se la JVM i és únic. Per tant, el *heap* és comú a tots els *threads* d'execució. En el *heap* es desen els objectes.

Per tot això que hem explicat, la mida de l'*stack* és molt menor que la del *heap*.

Així doncs, una vegada creat l'objecte de tipus *array* d'`int`, podríem assignar un valor a una de les seves caselles:

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

Hem assignat el valor `10` a la quarta casella de l'*array* `grades`. Tal com passa en altres llenguatges (per exemple, C/C++), la primera casella d'un *array* en Java es correspon amb l'índex `0`.

Per imprimir per pantalla, hem de fer servir l'ordre `System.out.print()` o `System.out.prinltn()`. Com ja sabem, la segona afegeix un salt de línia, és a dir, un `\n`. Dit això, analitzem el codi següent:

```
int number = 8;
int [] grades = new int[5];
grades[3] = 10;
System.out.println(number); //Imprimeix 8
System.out.println(grades[0]); //Imprimeix 0 (valor per defecte)
System.out.println(grades[3]); //Imprimeix 10
System.out.println(grades); //Imprimeix I@98af3857 --> adreça de memòria
```

Si mirem l'última línia del codi anterior, veiem que el que s'imprimeix per pantalla no és cap valor de l'*array*, sinó un «text estrany». Aquest «text estrany» no és més que l'adreça de memòria del *heap* on està allotjat l'objecte, que és un *array* d'`int`.

> Les variables que com `grades` desen una adreça de memòria, en Java són anomenades *variables de tipus referència* o, simplement, *referències*.

A diferència de llenguatges com C/C++, en Java no hi ha el tipus punter (o apuntador) com a tal:

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

El símbol `*` ens diu que `agePointer` és una variable que emmagatzema l'adreça de memòria –és a dir, un punter– d'una altra variable que és de tipus `int`. Com que `agePointer` és una variable, el compilador li assignarà una adreça de memòria de l'*stack*:

Contingut	Adreça de memòria
36	1000
@1000	1004
	1008
	1012

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

En declarar la variable `agePointer` de tipus punter d'`int`, el compilador li ha assignat la posició de memòria `1004`. En l'última instrucció del codi anterior, li hem assignat l'adreça de memòria de la variable `age`. Com podem veure, l'operador `&` en C/C++ retorna l'adreça de memòria d'una variable.

A partir d'aquí, puc fer servir tant `age` com `agePointer` per modificar el valor d'`age`, ja que `agePointer` sap en quina adreça de memòria està desat el valor de la variable `age`. Així doncs:

```
//Codi C/C++
age = 80;
*agePointer = 80; //El símbol * és l'Operador de desreferència o també
//anomenat d'indirecció
```

Les dues instruccions anteriors farien exactament el mateix, és a dir, assignarien el valor `80` a la variable `age`. Així doncs, amb qualsevol de les dues instruccions anteriors, la memòria en el nostre exemple quedaria així:

Contingut	Adreça de memòria
80	1000
@1000	1004
	1008
	1012

Quins avantatges té fer servir punters? L'avantatge rau a poder comunicar-se a un nivell més baix amb el maquinari, la qual cosa fa que l'execució del codi sigui més ràpida. Per exemple, treballar amb estructures de dades complexes és més senzill si s'utilitzen punters que variables. Així doncs, passar adreces de memòria a una funció és més ràpid que passar una còpia sencera de la variable (per exemple, una estructura de dades). A més, l'ús de punters permet aplicar operacions aritmètiques (per exemple, sumar i restar) sobre adreces de memòria per arribar a altres dades que estan en altres zones de memòria. Tot això és útil en àmbits com la creació de sistemes operatius.

No obstant això, l'ús de punters també provoca problemes, com ara:

- Pèrdua de robustesa i seguretat en intentar accedir a zones de memòria prohibides pel programa.
- Dificulta l'alliberament automàtic de memòria.
- Dificulta la desfragmentació de memòria.

- Dificulta la programació distribuïda de manera clara i eficient.

Un cop arribats a aquest punt, com es relacionen les referències i els punters en Java? Doncs bé, de manera molt informal i superficial, podríem dir que:

> Una **referència en Java** és com un punter de C/C++ amb la sintaxi d'un tipus primitiu i amb unes restriccions més fortes.

Punters en Java

Java no té punters explícitament, però sí implícitament. És a dir, les referències estan programades internament com a punters, però aquests punters no són accessibles per als programadors.

A partir de la definició anterior, per declarar una referència declararem una variable de la mateixa manera que fèiem amb els tipus primitius, però ara el tipus serà un *array* o una classe (pròpia de l'API de Java o creada per nosaltres).

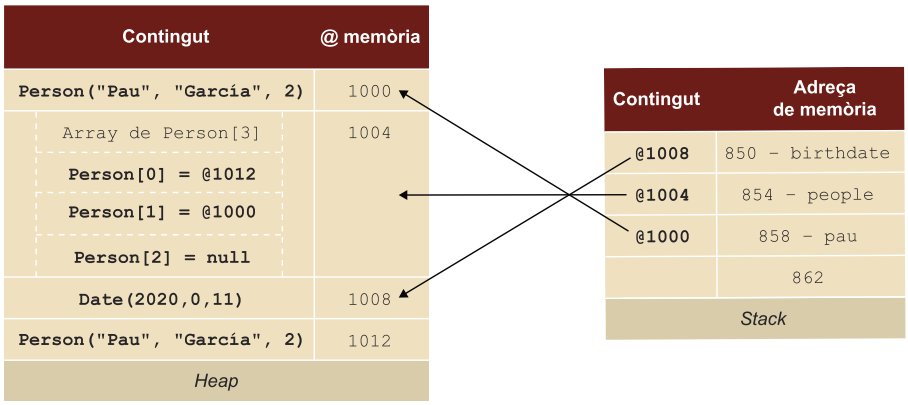
```
double[] grades; //grades és una referència a un array de double el valor del qual
//és null.

Date birthdate; //birthdate és una referència a un objecte de tipus Date.
//Date és una classe que existeix a l'API de Java.

Person david; //david és una referència a un objecte de tipus Person (classe
//creada per nosaltres)

Person [] people = new Person[3]; //array d'objectes Person, cada casella és
//una referència a un objecte Person amb valor null. Al seu torn, people és una
//referència a l'array d'objectes Person
```

El fet que Java no faci servir el terme *adreça de memòria* o *punter*, sinó *referència*, és perquè el terme *referència* afegeix un significat més genèric i més estricte. De fet, pròpiament dit, una variable de referència emmagatzema l'adreça de memòria d'un objecte.

Quan es declara una variable de tipus referència sense utilitzar l'operador `new`, el valor per defecte és `null`. Així doncs, en l'exemple anterior, tant `grades` com `birthdate` i `david` són tres referències que no apunten cap zona de memòria del *heap*, és a dir, són `null`. En canvi, la JVM sí que ha creat un objecte de tipus *array* amb `3` elements `Person`. L'adreça de memòria del *heap* d'aquest objecte li ha assignat com a valor a la referència `people`. Per tant, `people` és una referència a una zona de memòria del *heap*, per exemple, `I@78ab8876`. No obstant això, cada casella de l'*array* `people` és una referència que no apunta cap zona del *heap*, és a dir, són `null`. En el cas de les variables `grades`, `birthdate` i `david`, com també per a cada casella de `people`, encara no se'ls ha assignat un objecte amb l'operador `new` (o assignat una altra referència) i apunten `null`. No obstant això, si fem:

```
Date birthdate = new Date(2020,0,11); //birthdate és una referència a un objecte
//de tipus Date. Date és una classe que existeix a l'API de Java.

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

Person [] people = new Person[3];

people[0] = new Person("Pau", "García", 2); //és una referència diferent a la
//de pau. És a dir, people[0] i pau són dos objectes diferents encara que s'inicialitzen
//amb els mateixos valors.

people[1] = pau; //la casella 1 de people apunta la mateixa posició de memòria
//del heap que pau. És a dir, people[1] i pau són la «mateixa» referència.
```

El codi anterior de manera gràfica seria:



Java proporciona al programador referències i no punters per evitar els perills dels punters, per exemple, aplicar operacions aritmètiques sobre els punters o deixar memòria reservada dinàmicament (per exemple, mitjançant `malloc`, `calloc` o `realloc`) sense alliberar (sense haver fet servir `free`). Com ja hem comentat, Java és molt estricte amb les referències en comparació a com ho és C/C++ amb els punters. Algunes d'aquestes restriccions que imposa Java a les referències són:

- Un programador no pot canviar explícitament el valor d'una variable de tipus referència. El codi següent seria incorrecte:

```
Person pau = 10458;
```

- L'única manera en què una variable de tipus referència obté el seu valor és mitjançant l'assignació d'un objecte. Les dues instruccions següents són correctes:

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

- Com a conseqüència de les dues restriccions anteriors, un programador no pot aplicar operadors aritmètics sobre les referències.

- L'únic que és capaç de manipular directament les adreces de memòria és la JVM.

2.5. Operadors

Una vegada ja sabem com declarar i inicialitzar variables, és hora de fer «alguna cosa» amb elles. En aquest apartat veurem els principals operadors de Java. Però abans, definim què és un operador.

De la mateixa manera que en les matemàtiques, els **operadors** són símbols del llenguatge que indiquen que ha de realitzar-se una operació específica sobre un cert nombre d'operands i retornar un resultat que és fruit d'aplicar aquesta operació als operands. Per la seva banda, un operand és una entrada d'un operador.

Abans de continuar, cal definir el concepte *expressió*.

Expressió és una combinació de valors, funcions que retornen un valor, operadors i altres expressions que, mitjançant regles de precedència dels operadors que les formen, generen un resultat. Aquest resultat és una altra expressió.

Així doncs, tres exemples d'expressió podrien ser:

$$2 + 7 - add(8,2)$$
$$-1$$
$$(3 * 2) + 8$$

Un aspecte clau quan es tracta el tema dels operadors és la seva precedència. Quan les expressions són complexes és important saber quina és la precedència que té cada operador dins de l'expressió –és a dir, quin operador s'aplica primer– amb l'objectiu d'aconseguir el resultat esperat. A continuació, es mostren els operadors de Java ordenats de major a menor precedència. És a dir, l'operador que està més amunt en la taula s'aplica abans que un operador que està més a baix.

Taula 3. Operadors de Java ordenats de major a menor precedència

Operador	Precedència
postfix	`expr++ expr--`
unari	`++expr --expr +expr -expr ~ !`
multiplicatiu	`* / %`
addició	`+ -`
shiftat	`< >> >>>`

Operador	Precedència
relacional	`< > <= >= instanceof`
igualtat	`== !=`
bitwise AND	`&`
bitwise OR exclusiva	`^`
bitwise OR inclusiva	`|`
AND condicional	`&&`
OR condiconal	`||`
ternària	`? :`
assignació	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Els operadors que estan en la mateixa fila en la taula 3 tenen la mateixa precedència. En aquest cas, la regla que se segueix és que tots els operadors binaris (és a dir, que tenen dos operands), excepte els d'assignació, són avaluats d'esquerra a dreta, mentre que els operadors d'assignació (última fila de la taula) són avaluats de dreta a esquerra. Així mateix, els parèntesis, de la mateixa manera que en les matemàtiques, tenen la màxima precedència i se solen fer servir per canviar l'ordre d'avaluació. Vegem alguns exemples d'expressions per entendre les regles de precedència:

Exemple	Equivalent	Resultat
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 continuació veurem els diferents tipus d'operadors segons la freqüència d'ús.

2.5.1. Operadors d'assignació, aritmètics i unaris

En aquest apartat veurem:

- L'assignació simple.
- Els operadors aritmètics.
- Assignacions augmentades o compostes.
- Els operadors unaris.

Assignació simple

És un dels operadors més comuns en qualsevol programa. Com el seu propi nom indica, aquest operador consisteix a assignar una expressió (operand dret) a una variable (operand esquerre), de tipus primitiva o referència. Algun exemple podria ser:

```
int age = getAge()-(2*3); //assignem a "age" el resultat de restar sis (2*3) al
//valor retornat pel mètode/funció getAge().

int speed = 10; //assignem a la variable "speed" una expressió que és un
//literal numèric, p. ex., l'enter 10. Després de l'assignació "speed"
//emmagatzemarà el valor 10.

Person pau = new Person("Pau", "García", 2); //assignem un objecte, és a dir,
//"pau" és una referència (emmagatzema la adreça de memòria del heap de l'objecte
//Person creat).
```

Operadors aritmètics

Java proporciona els mateixos operadors aritmètics que hi ha en el llenguatge matemàtic més un de nou, molt comú en els llenguatges de programació, anomenat *mòdul* i el símbol del qual és `%`. Aquest nou operador divideix de manera entera (és a dir, sense decimals) l'operand de l'esquerra entre l'operand de la dreta i retorna el residu d'aquesta divisió.

Taula 4. Operadors aritmètics de Java

Operador	Descripció	Exemple	Resultat	Tipus de resultat
+	Suma	2+3	5	Numèric
-	Resta	5-6.5	-1.5	Numèric
*	Multiplicació	7*3	21	Numèric
/	Divisió	10/2	5	Numèric
%	Mòdul	10%7	3	Numèric

> Els operadors anteriors es poden aplicar entre diferents tipus de dades primitives que siguin numèriques, és a dir, queda exclòs el tipus de dada `boolean`.

Per exemple, en la taula anterior per a l'operador -, hem realitzat la resta d'un `int (5)` amb un `float (6.5)`, i el resultat de l'operació és un valor numèric de tipus `float`.

Un cop arribats a aquest punt cal tenir present el valor resultant d'aplicar un operador. En aquest cas, hem de distingir diverses casuístiques:

1) Operands del mateix tipus (casos `int`, `long`, `float` o `double`). Si els dos operands que intervenen en un operador aritmètic són del mateix tipus, el resultat serà d'aquest mateix tipus. Així doncs, si mirem la suma: `int + int = int`, `long + long = long`, `float + float = float` i `double + double = double`. El mateix succeiria amb la resta, la multiplicació, la divisió i el mòdul.

La regla anterior afecta la divisió entre enters (`int`), ja que el resultat real podria ser un nombre decimal (és a dir, `float` o `double`), però el resultat obtingut aplicant la regla anterior és que serà un enter (és a dir, sense decimals). Així doncs, 1/2 = 0. No obstant això, `1.0/2.0 = 0.5`. En aquest cas, si volem forçar que el resultat sigui un `float` o un `double`, hem de fer un *cast* explícit, és a dir, una conversió explícita del tipus, per exemple, `(float) (1/2) = 0.5`.

2) Operands del mateix tipus (casos `byte`, `short` i `char`). Si els dos operands són `byte` o `short` o `char`, les operacions aritmètiques són realitzades com si els operands fossin `int` (*cast* implícit perquè no hi ha pèrdua de precisió) i el resultat és un `int`. Per exemple, `byte + byte = int + int = int`. Si es vol recuperar el tipus original, cal fer una conversió explícita (és a dir, un *cast* explícit) del resultat:

Cast explícit

Normalment el *cast* explícit consisteix a indicar al compilador que volem que es realitzi una pèrdua de precisió (truncatge) en passar d'un tipus a un altre de més petit i que som completament conscients d'aquesta possible pèrdua durant la conversió, per exemple, `int a = (int)3.5; //a valdrà 3`. Si no fem el *cast* explícit quan hi ha una possible pèrdua de precisió (per exemple, de `double`, `float` i `long` cap a `int`), el compilador ens donarà un error.

```
byte b1 = 1, b2 = 3, b3;
b3 = (byte)(b1 + b2); //si haguéssim escrit b3 = b1 + b2; hauríem
//obtingut un error perquè el resultat de b1 + b2 és un int i l'estem
//assignant a una variable de tipus byte.
```

En el cas del tipus `char`, el valor `int` assignat a un caràcter serà el seu corresponent nombre Unicode dins del rang `[0, 65535]`.

Per a aquests tres tipus –`byte`, `short` i `char`–, Java sempre fa les operacions aritmètiques com si fossin `int`. És a dir, els promociona (els aplica un càsting implícit) a `int`.

3) Operands de tipus diferents. En aquest cas, l'operand de tipus més petit és promocionat automàticament al tipus més gran –és a dir, es fa un càsting implícit–, i es duu a terme l'operació en el tipus més gran, i el resultat és del tipus més gran. L'ordre de promoció en Java és el següent: `int → long → float → double`. Així doncs: `1/2.0 → 1.0/2.0 = 0.5 (double)`.

En aquest tipus d'operacions hem de tenir present que si volem assignar a una variable el resultat d'una expressió com `5-6.5`, aquesta variable ha de ser del tipus major, en aquest cas `float` o `double`.

Un cop arribats a aquest punt, cal destacar la necessitat de ser conscients que en fer una operació aritmètica sobre dos nombres enters (`int`) molt grans, pot ser que el resultat no es pugui expressar com a `int` (perquè se surt del rang vàlid de dades), per la qual cosa el resultat serà un `long`. Fins i tot pot donar-se el cas de sortir-se dels límits inferior o superior d'un `double`.

Finalment, cal indicar que l'operador `+` es pot fer servir per concatenar (és a dir, unir) cadenes de caràcters, que en Java són objectes de la classe `String` (ho veurem més endavant). Per exemple:

```
String texto1 = "Hola ";
String texto2 = "David!";
String texto1 = texto1 + texto2; //després de l'assignació el valor de texto1 és
//"Hola David!"
```

Assignacions augmentades o compostes

D'altra banda, els operadors aritmètics de la taula 4 permeten combinar-se amb l'operador d'assignació, i crear nous operadors anomenats *assignacions augmentades o compostes*. Aquests operadors assignen a la variable que hi ha al costat esquerre el resultat d'aplicar l'operador aritmètic al valor de la variable (operand esquerre) i a l'expressió de la dreta (operand dret). Vegem en la taula 5 aquests operadors suposant que el valor de la variable `number` és `10`.

Taula 5

Assignació augmentada	Exemple	Equivalent	Nou valor de `number`
+=	`number+=3`	`number = number + 3`	`13`
-=	`number-=6`	`number = number - 6`	`4`
***=**	`number*=3`	`number = number * 3`	`30`
/=	`number/=2`	`number = number / 2`	`5`
%=	`number%=7`	`number = number % 7`	`3`

Sucre sintàctic

El sucre sintàctic (en anglès, *syntactic sugar*) és una nova sintaxi que s'afegeix a un llenguatge de programació per fer algunes operacions ja existents de manera més senzilla i abreujada. Aquesta nova sintaxi fa «més dolç» el llenguatge. Un exemple és l'operador `+=`.

Hi ha una petita diferència entre els operadors simples (`+`, `-`, `*`, `/`, `%`) i els compostos que acabem de veure:

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

b2 = (b1 + b2); //error: el resultat és un int, hi ha pèrdua de precisió en
//intentar-lo assignar a una variable de tipus byte. Cal fer càsting
//explícit.

b2 += b1; //correcte, l'assignació composta fa el cast per nosaltres
```

Operadors unaris

Aquest tipus d'operadors només utilitzen un operand. Els operadors unaris de Java es mostren en la taula 6.

Taula 6. Operadors unaris de Java

Operador	Descripció	Exemple	Descripció
+	Signe positiu	`number = +3;`	El + indica que el número 3 és positiu.
–	Signe negatiu	`number = -3;`	El – indica que el número 3 és negatiu.

++ (sufix)	Increment	`int number = 3;`
		`int a = number++;`
			El seu codi equivalent és:
			`int number = 3;`
			`a = number; //a val 3`
			`number = number + 1; //number val 4`

++ (prefix)	Increment	`int number = 3;`
		`int a = ++number;`
			El seu codi equivalent és:
			`int number = 3;`
			`number = number + 1; //number val 4`
			`a = number; //a val 4`

-- (sufix)	Decrement	`int number = 3;`
		`int a = number--;`
			El seu codi equivalent és:
			`int number = 3;`
			`a = number; //a val 3`
			`number = number - 1; // number val 2`

-- (prefix)	Decrement	`int number = 3;`
		`int a = --number;`
			El seu codi equivalent és:
			`int number = 3;`
			`number = number - 1; //number val 2`
			`a = number; // a val 2`

!	Negació	`boolean a = false;` `boolean b = !a;`	La variable `b` després de l'assignació serà `true` (el valor contrari al de `a`).

Com podem veure en la taula 6, una de les particularitats que tenen els operadors ++ i -- és que poden aplicar-se com a prefix o sufix. Vegem un exemple més extens de les diferències:

```
int a = 0;
int b = a++; //Després d'aquesta sentència, b val 0 i a val 1
int c = ++a; //Després d'aquesta sentència, c val 2 i a val 2
int d = (a += 1); //Després d'aquesta sentència, d val 3 i a val 3
```

2.5.2. Operadors d'igualtat, relacionals i condicionals

Veurem els operadors d'aquest apartat en tres grups:

- Operadors d'igualtat i relacionals.
- Operadors condicionals.
- Operador `instanceof` (pertinença a una classe o interfície).

Operadors d'igualtat i relacionals

Els operadors d'aquestes dues tipologies, també anomenats amb el terme *operadors comparatius*, permeten comparar dos operands. El resultat d'aquests dos operands serà un `boolean`, és a dir, `true` o `false`.

Taula 7. Operadors d'igualtat relacionals

Operador	Descripció	Exemple	Resultat
==	Igual que	3 == 3.0 true == false	true false

!=	Diferent que	int b = 5;
		b != 10;
		b != 5;
			
			true
			false

>	Més gran que	15 > 10 7 > 10	true false
>=	Més gran o igual que	8 >= 7 7 >= 8	true false
<	Menor que	-6.5 < 6.5 15 < (13.75+1)	true false
<=	Menor o igual que	3.2 < 3.3 10 <= 5	true false

Com es pot veure, es poden comparar expressions els resultats de les quals siguin de tipus numèrics diferents. El compilador s'encarrega de fer les conversions oportunes abans de fer la comparació. Alhora, els operadors == i != accepten també valors de tipus `boolean`.

Operadors condicionals

En aquesta categoria només hi ha dos operadors: `AND` condicional i `OR` condicional. Aquests actuen sobre dues expressions/operands `boolean`.

En el cas de l'`AND` condicional, aquest retorna `true` si tots dos operands són `true`. Mentre que l'operador `OR` condicional retorna `true` quan almenys un dels dos operands és `true`. Aquests dos operadors es relacionen amb els operadors lògics · i + de l'àlgebra de Boole, respectivament. També se'n pot fer l'equivalència amb les portes lògiques `AND` i `OR`, respectivament. Per tot això que acabem d'explicar, una manera senzilla d'obtenir el resultat d'un `AND` i un `OR` és convertir mentalment els operands `false` en `0`, els `true` en `1`, l'`AND` per una multiplicació i l'`OR` per una suma. Si el resultat de fer la multiplicació o la suma és `1`, llavors el resultat definitiu és `true` i, en cas contrari, és `false`. Cal tenir present que en l'àlgebra de Boole `1+1 = 1`.

Taula 8. Operadors condicionals

Operador	Descripció	Exemple	Resultat
&&	AND condicional	(3 == 3.0) && (50<100) false && true	true false

Operador	Descripció	Exemple	Resultat

`||`	`OR` condicional	`(3 == 3.0) || (50<100)`
		`false || true`
		`(3>5) || (5<4)`
			`true`
			`true`
			`false`

Els dos operadors anteriors tenen, a més, un comportament que s'anomena de *curtcircuit*, ja que el segon operand només és avaluat si és necessari. Quan cal avaluar l'expressió del segon operand? Quan amb el resultat del primer operand no es pot determinar amb seguretat el resultat de l'operació `AND` o `OR`. Per exemple:

```
(3 == 3.0) || (50<100) // No avaluarà (50<100) perquè (3==3.0) és true i
//l'operador || és true quan almenys un dels dos operands és true. Així
//doncs, és igual si el segon operand, és a dir, (50<100) és true o no,
//perquè el resultat serà true perquè el primer operand és true.

false && (50<100) // No avaluarà (50<100) perquè false és false i l'operador
///&& només és true quan els dos operands són true.
```

Potser us pregunteu: i per què és important saber que `AND` i `OR` es comporten com un curtcircuit? Amb l'exemple següent esperem que ho vegeu clar:

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

Imagineu que la funció/mètode `doSomething()` fa una sèrie d'operacions i retorna `true` o `false` en funció d'aquestes operacions. A més, dins d'aquestes operacions que realitza, n'hi ha una que és comptar les vegades que ha estat cridada aquesta funció. Doncs bé, en l'exemple anterior, aquest comptador seria sempre zero perquè mai no es cridaria `doSomething()`.

Operador `instanceof`

Aquest operador compara un objecte amb un tipus de classe concreta. És a dir, es fa servir per saber si una instància/objecte pertany a una classe concreta (pare directe o no) o implementa una determinada interfície.

Classe i interfície

Veurem aquests dos conceptes més endavant tant en aquesta guia com en els mòduls teòrics de l'assignatura.

Un exemple d'ús d'aquest operador podria ser:

```
Person pau = new Person("Pau", "García", 2);
pau instanceof Person; //retornaria true ja que la referència pau emmagatzema un
//objecte de la classe Person
pau instanceof Ball; //retornaria false perquè pau no és una referència a un
//objecte de la classe Ball.
```

2.5.3. Operadors bitwise i shiftat

En aquest apartat veurem operadors que treballen directament amb els bits. Aquests operadors són molt menys utilitzats que els vistos fins ara i es classifiquen en:

- **Operadors bitwise** (també coneguts com a *operadors a nivell de bits*).
- **Operadors de shiftat** (o *operadors de desplaçament de bits*).

Operadors bitwise

Aquests operadors manipulen bits, és a dir, 0 i 1.

Taula 9. Operadors bitwise

Operador	Descripció	Tipus operands	Exemple	Resultat
&	AND de bits	byte, short, int, long, char (tractat com a int) i boolean	(3 == 3.0) & (50<100) false & true 60 & 13	true false 12
|	OR inclusiva de bits	byte, short, int, long, char (tractat com a int) i boolean	(3 == 3.0) & (50<100) true | false (3>5) | (5<4) 60 & 13	true true false 61
^	XOR (o OR exclusiva) de bits	byte, short, int, long, char (tractat com a int) i boolean	5^3 (3 == 3.0) ^ (50<100) true ^ false	6 false true
~	Complement a 1	byte, short, int, long i char	~00000000 (és un byte) ~5 (en què 5 és un int: 0101)	11111111 (byte) -6 (int)

Els operadors AND (&) i OR (|) de bits quan els operands són boolean actuen com hem comentat amb els seus homònims condicionals, però sense el comportament de «curtcircuit». És a dir, independentment del valor de l'expressió del primer operand, el segon operand sempre s'avaluarà. Així doncs:

```
false & doSomething(); //sempre es cridarà doSomething();
```

El que fa l'operador XOR (o OR exclusiva) és retornar 1 quan dos bits de la mateixa posició són diferents entre els dos operands. En cas contrari retorna 0. De fet, el seu equivalent amb AND i OR lògics seria: (A AND !B) OR (!A AND B). Vegem l'exemple de la taula:

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

Aquest operador també es pot aplicar sobre operands el resultat dels quals sigui un `boolean`. En aquest cas, quan tots dos operands coincideixen en valor, el resultat és `false` i, en cas contrari, el resultat és `true`. Així doncs: `true^true` és `false`, `false^false` és `false`, `true^false` és `true` i `false^true` és `true`.

L'operador complement a 1 (~), per la seva banda, converteix la representació binària de l'operand i canvia els zeros per uns i els uns per zero. En l'exemple de la taula 9, els passos interns són:

`int`, `short`, `long` i `byte`

Els tipus `int`, `short`, `long` i `byte` en Java són nombres en complement a 2.

```
~5 = ~(00000101) = 11111010 = -6 /*a 5 li hem aplicat amb ~ el complement a 1, el resultat
del qual és 11111010, però Java, en desar-lo com a int utilitza per a la seva
interpretació el complement a 2; per això, el resultat és -6*/
```

Operadors de shiftat

Aquests operadors manipulen bits desplaçant-los cap a la dreta o cap a l'esquerra.

Taula 10. Operadors de shiftat

Operador	Descripció	Exemple	Resultat
<<	Desplaça l'operand de l'esquerra cap a l'esquerra tantes vegades com indica l'operand de la dreta, i les posicions shiftades són omplertes amb zeros.	`60 << 2`	`240` `(00000000 00000000 00000000 01110000)`
>>	Desplaça l'operand de l'esquerra cap a la dreta tantes vegades com indica l'operand de la dreta. Preserva el bit de signe original (primer bit per l'esquerra) en les posicions shiftades. Així manté el signe del valor original.	`60 >> 2`	`15` `(00000000 00000000 00000000 00001111)`
>>>	Desplaça l'operand de l'esquerra cap a la dreta tantes vegades com indica l'operand de la dreta, i les posicions shiftades són omplertes amb zeros. A diferència de >>, aquest operador no té en compte el signe del valor original.	`60 >>> 2`	`15` `(00000000 00000000 00000000 00001111)`

2.6. Blocs de flux d'execució

Com ja hem comentat a l'apartat «Estructura bàsica d'un programa», els blocs són una agrupació d'instruccions que són tractades com una única unitat. En Java podem distingir tres tipus de blocs que afecten el flux d'execució del programa, és a dir, determinen quines instruccions s'executen i quines no: bloc seqüencial, bloc condicional i bloc iteratiu.

2.6.1. Bloc seqüencial

Un programa, per si mateix, és una seqüència d'instruccions. És a dir, les instruccions s'executen en l'ordre en el qual estan escrites (sempre de dalt cap avall). Aquest és el bloc més comú.

```
int number = 10; //1a instrucció en executar-se
number += 3; //2a instrucció en executar-se, s'executa després d'int number = 10;
System.out.println(number); //3a instrucció i última en executar-se
```

2.6.2. Bloc condicional

Un bloc condicional és el que serveix per executar o no una sèrie d'instruccions segons una condició. Dins d'aquest tipus de bloc, també anomenat *de decisió* o *selecció*, hi ha diverses alternatives.

1) Bloc `if`.

Aquest bloc serveix per indicar un conjunt d'instruccions que s'han d'executar si es compleix una expressió lògica, és a dir, si aquesta expressió és `true`. Aquesta expressió lògica se sol anomenar *condició*.

Patró del bloc	Exemple

`if(condició){`
`  //Codi: s'executa si la condició`
`  //és true`
`}`
 	
`//Pròxima instrucció`
	`int number = 10;`
	
	`if(number >= 10){`
	`  number += 6;`
	`}`
	
	`number -= 5; //resultat 11`

Encara que no és recomanable, si el codi que hi ha dins d'un bloc `if` només és una instrucció, es pot obviar l'ús de les claus. El codi següent seria equivalent a l'anterior:

```
int number = 10;
if(number >= 10)
  number += 6;
number -= 5; //resultat 11
```

2) Bloc `if-else`.

Aquest bloc serveix per indicar dos blocs d'instruccions que són excloents entre si. El primer bloc (bloc dins de l'`if`) s'executarà si es compleix una expressió lògica, és a dir, si aquesta expressió és `true`. En cas contrari, s'executarà el segon bloc (bloc dins de l'`else`).

Patró del bloc	Exemple

`if(condició){`
`  //Codi: s'executa si la condició`
`  //és true`
`}else{`
`  //Codi: s'executa si la condició`
`  //és false`
`}`
 	
`//Pròxima instrucció`
	`int number = 10;`
	
	`if(number > 10){`
	`  number += 6;`
	`}else{`
	`  number *= 5;`
	`}`
	
	`number -= 5; //resultat 45`

3) Bloc `if` imbricat o `else-if`.

Dins d'un `if` o un `else` es pot escriure/imbricar un altre bloc condicional, però també hi ha una opció més curta –anomenada `if` imbricat– per al cas en què vulguem imbricar dins de l'`else`.

Patró del bloc	Exemple

`if(condició1){`
`  //Codi: s'executa si la condició1`
`  //és true`
`}else if(condició2){`
`  //Codi: s'executa si la condició1`
`  //és false i la condició 2 és true`
`}else{`
`  //Codi: s'executa si la condició1`
`  //i la condició 2 són false`
`}`
 	
`//Pròxima instrucció`
	`int number = 8;`
	
	`if(number > 10 && number < 15){`
	`  number += 6;`
	`}else if(number > 8){`
	`  number *= 5;`
	`}else{`
	`  number += 3;`
	`}`
	
	`number -= 5; //resultat 6`

Si ens hi fixem, aquest bloc no deixa de ser un `if-else` en el qual l'`else` no té la clau `{` sinó un `if`. No cal posar l'últim `else` si no ho requereix la lògica del programa. És a dir, si no volem fer res quan totes les condicions són falses, no cal posar un bloc `else`, ni tan sols buit (és a dir, sense instruccions).

4) Operador ternari.

Hi ha un operador condicional anomenat *ternari* que compacta un `if-else` més la devolució d'un valor en una única instrucció.

Patró del bloc	Exemple

`condició ? exprTrue : exprFalse;`
`/*El resultat és exprTrue si la condició`
`és true. En cas contrari, el resultat és`
`exprFalse. */`
	`int number = (8>10) ? 5 : 3;`
	
	`number -= 2; //resultat 1`

Aquest operador és molt utilitzat per assignar valors a una variable en funció d'una condició.

5) Bloc `switch`.

Aquest bloc és una alternativa a la concatenació de blocs `if` imbricats per a valors fixos (no rangs) d'una expressió que fa de selector. Els valors del selector poden ser `byte`, `short`, `int`, `char` o `String` (és a dir, text/cadena de caràcters; des de JDK 1.7).

Patró del bloc	Exemple

`switch(selector){`
`  case valor1:`
`  //Bloc de codi;`
`  break;`
`  case valor2:`
`  //Bloc de codi;`
`  break;`
`  …`
`  case valorN:`
`  //Bloc de codi;`
`  break;`
`  default:`
`  //Bloc de codi;`
`}`
 	
`//Pròxima instrucció`
	`int number = 2, age = 30;`
	
	`switch(number){`
	`  case 0:`
	`  age = 50;`
	`  break;`
	`  case 1:`
	`  case 2:`
	`  age = 25;`
	`  break;`
	`  default:`
	`  age = 10;`
	`}`
	
	`age -= 5; //resultat 20`

Els casos s'avaluen de manera seqüencial fins que el seu valor coincideix amb el del selector i llavors entra en el seu bloc. Així doncs, segons el nostre programa, no serà el mateix posar el `case 0` primer que posar-lo després del `case 2`, per exemple. Una vegada s'entra en un cas, s'aniran executant tots els blocs fins que un d'ells contingui un `break`. Quan es troba el `break`, el programa surt del `switch`. És a dir, en l'exemple anterior, si `number` hagués estat `1`, s'executaria el bloc del `case 1`, però en no haver-hi `break`, s'executaria també el bloc del `case 2`, i s'assigna el valor `25` a la variable `age`. Com que el `case 2` té un `break`, l'execució surt del `switch` i passa a la instrucció `age -= 5;`. Així doncs, el programa es comporta de manera idèntica tant si el valor de `number` és `1` com si és `2`.

Així mateix, no cal posar el cas `default` si no se'n requereix l'ús, el codi del qual s'executa si cap cas no coincideix amb el valor del selector.

Des de JDK 13 en endavant es pot fer servir la paraula reservada `yield` per retornar un valor en acabar un cas i, per tant, acabar el bloc `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 aquest punt, number tindrà el valor 1 i s'haurà imprès per pantalla el
//text "Hola!".
```

2.6.3. Bloc iteratiu

Aquest tipus de bloc, també anomenat *bloc de bucle*, serveix per executar un conjunt d'instruccions repetidament fins que deixa de complir-se una condició. Hi ha diverses alternatives.

1) Bloc `while`

Executa el codi que hi ha dins de les seves claus mentre la seva condició sigui `true`. Quan la condició sigui `false`, finalitza el bucle i es passen a executar les instruccions que hi ha després del bloc `while`. Cal destacar que la condició s'avalua en cada iteració.

Patró del bloc	Exemple

`while(condició){`
`  //Codi que cal repetir`
`}`
 	
`//Pròxima instrucció`
	`int index = 3, number = 0;`
	`boolean end = false;`
	
	`while(index>0){`
	`  number++;`
	`  index--;`
	`}`
	
	`number += 2; //resultat 5`
	
	`while(!end){`
	`  number++;`
	`  if(number>=7){`
	`  end = true;`
	`  }`
	`}`
	
	`number += 2; //resultat 9`

2) Bloc `do-while`

Aquest bloc és similar a l'anterior, però la condició s'avalua per primera vegada quan s'han executat les instruccions que hi ha dins del bloc. No obstant això, a cada iteració s'avalua la condició.

Patró del bloc	Exemple

`do{`
`  //Codi que cal repetir`
`}while(condició);`
 	
`//Pròxima instrucció`
	`int index = 0, number = 0;`
	
	`do{`
	`  number = 3;`
	`}while(index>0);`
	
	`number += 2; //resultat 5`

En l'exemple anterior, s'executa una vegada el codi que hi ha dins del bloc `do-while`, ja que la condició no s'ha avaluat fins que es passa per l'assignació `number = 3;`. Una vegada feta l'assignació es comprova si la condició és `true`. En aquest cas, es veu que és `false` i s'acaba el bloc `do-while`.

A diferència del bloc `while`, el codi que hi ha dins de les claus del `do-while` sempre s'executa com a mínim una vegada, mentre que en el cas del `while` pot ser fins i tot que no s'executi mai.

Per acabar, és important adonar-se que després de la condició del `do-while` s'ha d'escriure un punt i coma (`;`).

3) Bloc `for`

Aquest bloc proporciona una manera compacta d'iterar sobre un interval de valors.

Patró del bloc	Exemple

`for(inicialització; condició; increment){`
`  //Codi que cal repetir`
`}`
 	
`//Pròxima instrucció`
	`for(int i=1; i<10; i++){`
	`  System.out.println(i);`
	`}`
	
	`//imprimeix de l'1 al 9`

Quan es fa servir aquest bloc, cal tenir en compte que:

- L'expressió d'`inicialització` s'executa només la primera vegada que s'executa el bloc. És possible declarar una o diverses variables dins de l'expressió d'`inicialització`. L'abast (*scope*) o la visibilitat d'aquestes variables està limitat al bloc `for` on han estat declarades. És a dir, aquestes variables només existeixen dins del bloc `for`. Si les variables que inicalitzem dins de la `inicialització` del `for` no es necessiten fora del `for`, la millor pràctica és declarar-les en l'expressió d'`inicialització` del mateix `for` i no fora d'ell. D'aquesta manera, es limita el temps de

vida d'aquestes variables. Els noms típics per a aquestes variables que es fan servir per controlar el bloc `for` solen ser `i`, `j` i `k`.

- El bloc (és a dir, la iteració) acaba quan la `condició` és `false`.
- L'expressió d'increment és invocada després d'executar el codi que hi ha dins de les claus. En realitat, aquesta expressió pot ser d'increment o de decrement.
- Tant en l'expressió d'inicialització com en la d'increment es pot posar més d'una instrucció, separant-les amb comes.

```
for(int row = 0, col = 0; col < 10; row++, col++){
   //Codi que cal repetir
}
```

4) Bloc `enhanced for`

El bloc `for` anterior té una altra sintaxi dissenyada per iterar sobre col·leccions anomenades *enhanced for*. Per col·lecció ens referim tant a un *array* com a una estructura de dades com pot ser una pila, una cua, etc. que l'API de Java proporciona.

Patró del bloc	Exemple

`for(tipus nomVariable : nomColeccio)`
`//Codi que cal repetir per a tots els elements`
`//que hi ha dins de nomColeccio`
`}`
 	
`//Pròxima instrucció`
	`int[] numbers = {1, 2, 3};`
	
	`for(int value : numbers){`
	`  System.out.println(value);`
	`}`
	
	`//imprimeix 1, 2 i 3 en`
	`//línies separades`

Si ens hi fixem, en la variable `value` s'emmagatzema el valor actual de l'*array* `numbers`. És a dir, per si mateix, l'`enhanced for` crea un índex intern inicialitzat a 0 i el va incrementant fins a arribar a la longitud (o mida) màxima de la col·lecció.

El codi anterior amb un `for` simple seria:

```
int[] numbers = {1, 2, 3};
for(int i = 0; i < numbers.length; i++){
    System.out.println(numbers[i]);
}
```

En aquest tipus de bucles és interessant i útil utilitzar la manera de declarar variables de JDK 10 (`var`) perquè sigui el compilador qui infereixi el tipus dels objectes de la col·lecció. Per exemple:

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

2.6.4. Instruccions de control de flux

En aquest apartat, veurem tres instruccions que permeten canviar el flux d'execució d'un programa.

1) Instrucció `break`

La instrucció `break` té dues modalitats: etiquetada i sense etiqueta. Aquesta última l'hem vist quan parlàvem del bloc `switch`. No obstant això, també la podem fer servir per acabar un bloc iteratiu (és a dir, `while`, `do-while` i `for`). Per exemple:

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

Per la seva banda, un `break` etiquetat permet finalitzar el bloc iteratiu que tingui l'etiqueta que s'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 l'exemple anterior, quan la variable `i` val `5` i la variable `j` val `3`, s'executa el `break` etiquetat. El que fa és finalitzar el bucle exterior que ha estat etiquetat amb el nom *etiqueta*, la qual cosa fa que el valor `5` només s'imprimeixi tres vegades i no s'imprimeixin els valors del `6` al `9` (és a dir, acaba l'execució d'aquest codi).

Si haguéssim escrit un `break` sense etiqueta, el bucle que inicialitza la variable `j` seria el que hagués acabat i imprimiria el valor de la `i` en aquell moment (és a dir, `5`) tres vegades i continuaria amb el bucle exterior per continuar imprimint `i` igual a `6` deu vegades fins a arribar a imprimir deu vegades el valor `9`.

2) Instrucció `continue`

La instrucció `continue` s'utilitza per saltar la iteració actual d'un bloc iteratiu (és a dir, `while`, `do-while` o `for`). De manera homòloga a la instrucció `break`, la instrucció `continue` també té la seva forma etiquetada i no etiquetada. Un exemple de versió no etiquetada seria:

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

for(int i = 0; i < numbers.length; i++){
  if(i == 2){
        continue; //Quan i sigui 2, l'execució anirà a l'inici del bucle,
                  //obviant el que hi ha a continuació per a la iteració i = 2
  }
  System.out.println(i); //Imprimirà 1 i 3 en línies separades
}
```

De la mateixa manera que amb `break`, un `continue` etiquetat permet saltar la iteració d'un bucle extern.

```
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 l'exemple anterior, només s'imprimiria `5` tres vegades, en comptes de deu vegades.

3) Instrucció `return`

La instrucció `return` s'utilitza dins de funcions/mètodes i el que fa és sortir de la funció per seguir el flux d'execució a partir de la instrucció que hi ha just després de la crida de la funció/mètode. El `return` pot retornar un valor o no. Si retorna un valor, el tipus del valor retornat ha de coincidir amb l'indicat en la signatura de la funció/mètode.

```
return 10;
return;
```

2.7. `String`

En aquest apartat veurem un tipus de referència molt utilitzat en Java: `String`. `String` és una classe que proporciona l'API de Java i que ens serveix per emmagatzemar cadenes/seqüencies de caràcters, és a dir, textos. Atès que la manipulació de textos és vital en qualsevol programa, la classe `String` s'ha convertit en un tipus primitiu *de facto* (encara que no ho és; és una classe, és a dir, un tipus referència). Qualsevol cadena de caràcters està composta per zero o més caràcters, tots ells delimitats per cometes dobles ".

```
String text = "Hola UOC!"; //"Hola UOC!" és un literal que s'assigna a una referència de
//tipus String
char letter = 'A'; //Els String s'escriuen entre " i els char entre '
String emptyMsg = ""; //Text buit
```

Excepcionalment, en Java, per instanciar un objecte de tipus `String` no cal cridar-ne el constructor, només cal assignar-li una cadena de caràcters (és a dir, un text). Veurem que també es pot cridar el constructor amb la paraula reservada `new`, però això ho comentarem més endavant, quan vegem la diferència entre les classes `String`, `StringBuilder` i `StringBuffer`.

2.7.1. *Escape sequences*

Hi ha diferents seqüències de fuita[7] que serveixen per indicar alguna instrucció de format de text. per exemple:

[7] En anglès, *escape sequences*.

- `\n` afegeix un salt de línia al final del text en posar el cursor en la línia següent.
- `\t` afegeix un tabulador on apareix.

```
String text = "Hola UOC!\tHola\nEps!";
System.out.print(text);
/* El print imprimirà per pantalla:
Hola UOC!      Hola
Eps!

*/
```

Què succeeix si el text té dins unes cometes dobles o una barra obliqua inversa? Doncs que el compilador es confondrà perquè no sabrà quines cometes dobles són les del text i quines les que limiten la cadena de caràcters. El mateix passa amb la barra obliqua inversa: no sabrà si es tracta d'una ordre o no. Com solucionar aquest conflicte? Afegint una doble inversa davant de les cometes o de la barra obliqua inversa que volem que formi part del text.

```
String text = "Hola \"UOC\"!\\tHola";
System.out.print(text);
/* El print imprimirà per pantalla:
Hola "UOC"!\tHola
Eps!

*/
```

2.7.2. Concatenar amb +

Quan vam veure els operadors aritmètics vam dir que l'operador + tenia un comportament especial amb els `String`. En concret vam comentar que serveix per concatenar una cadena de caràcters –és a dir, `String`–, amb un altre element (per exemple, un altre `String` o un tipus primitiu).

```
String text = "My name is ";
System.out.print(text+"David"); //Escriu per 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 cas de concatenar un `String` amb qualsevol altre tipus de valor, per exemple, un `int`, aquest últim es converteix en `String` i tots dos `String` (és a dir, l'original i l'`int` convertit en `String`) es concatenen i donen com a resultat un nou `String`.

2.7.3. Alguns mètodes interessants

La classe `String` proporciona molts mètodes, els quals ens faciliten moltes tasques.

Enllaç recomanat

Podeu veure els mètodes disponibles a la documentació oficial de la classe `String` (https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html).

Taula 11. Alguns mètodes de la classe `String`

Signatura del mètode	Descripció	Exemple

`char charAt(int index)`	Retorna el caràcter que es troba en la posició `index`	`String text = "Hola!";`
		`char c = text.charAt(0);`
		`System.out.print(c); //H`

`boolean contains(CharSequence c)`.	Diu si una cadena de caràcters està dins de l'`String` o no.	`String text = "Hola UOC";`
		`boolean b = text.contains("UOC");`
		`System.out.println(b); //true`

`boolean equals(String another)`.	Comprova si un `String` és igual que un altre.	`String text = "UOC";`
		`System.out.print(text.equals("UOC")); //true`
		`System.out.print(text.equals("uoc")); //false`

Signatura del mètode	Descripció	Exemple

`boolean equalsIgnoreCase(String another)`.	Comprova si un `String` és igual que un altre ignorant majúscules i minúscules.	`String text = "UOC";`
		`System.out.print(text.equalsIgnoreCase("UOC"));`
		`//true`
		`System.out.print(text.equalsIgnoreCase("uoc"));`
		`//true`

`int indexOf(String substring)`.	Retorna l'índex en el qual comença el `substring` dins de l'`String`. Si no hi és, retorna `-1`.	`String text = "Patata";`
		`System.out.print(text.indexOf("t")); //2`
		`System.out.print(text.indexOf("ata")); //1`

`boolean isEmpty()`.	Comprova si l'`String` és buit ("").	`String text = "UOC";`
		`String empty = "";`
		`System.out.print(text.isEmpty()); //false`
		`System.out.print(empty.isEmpty());//true`

`int length()`.	Retorna el nombre de caràcters que té el text incloent-hi espais.	`String text = "Hola!";`
		`int n = text.length();`
		`System.out.print(n); //5`

`String replace(char old, char new)`.	Retorna l'`String` amb totes les aparicions del caràcter `old` reemplaçades pel caràcter `new`.	`String text = "Hola patata";`
		`text = text.replace("a","*");`
		`System.out.print(text); //Hol* p*t*t*`

`String[] split(String regex)`.	Separa l'`String` original en punts `String` que estiguin separats per l'`String regex`.	`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`

`String substring(int beginIndex)`.	Retorna el subtext que hi ha dins del text a partir de l'índex indicat.	`String text = "Hola UOC";`
		`text = text.substring(5);`
		`System.out.print(text); //UOC`

`String substring(int beginIndex, int endIndex)`.	Retorna el subtext que hi ha dins del text des de l'índex inicial fins a l'índex final (no inclòs).	`String text = "Hola UOC";`
		`text = text.substring(5,6);`
		`System.out.println(text); //U`

`String toLowerCase()`.	Retorna l'`String` escrit tot en minúscules.	`String text = "UOC";`
		`text = text.toLowerCase();`
		`System.out.print(text); //uoc`

`String toUpperCase()`.	Retorna l'`String` escrit tot en majúscules.	`String text = "uoc";`
		`text = text.toUpperCase();`
		`System.out.print(text); //UOC`

Signatura del mètode	Descripció	Exemple

`String trim()`.	Elimina els espais inicials i finals de l'`String`.	`String text = "  Hola patata ";`
		`text = text.trim();`
		`System.out.print("_"+text+"_"); //_Hola patata_`

2.7.4. Conversions

Sovint necessitem convertir un valor en format text (és a dir, `String`) en un tipus primitiu per fer alguna operació. D'altres vegades, tindrem un valor en format primitiu i necessitarem tenir-lo en format text. Tots dos casos són un tipus de conversió.

1) De tipus primitiu a `String`

Una manera senzilla i ràpida de convertir un valor primitiu en un text és concatenant-li un text (encara que sigui buit, per exemple, `""`). Com:

```
int num = 50;
String text = 50+""; //ara text és "50", aquest 50 és text.
```

Una altra manera de fer-ho, potser més explícita i clara a simple vista, és mitjançant un mètode estàtic[8] anomenat `valueOf` que serveix per a tots els primitius.

[8] Més endavant, veurem què és un mètode estàtic.

```
int num = 50;
String text = String.valueOf(num); //ara text és "50", aquest 50 és text.
text = String.valueOf(50.25); //ara text és "50.25", aquest 50.25 és text.
text = String.valueOf(false); //ara text és "false", aquest false és text.
```

Una última forma de convertir un tipus primitiu en un `String` és mitjançant el mètode estàtic `toString` de cada classe *wrapper*[9] dels tipus primitius.

[9] Més endavant, veurem què és una classe *wrapper*.

```
int num = 50;
String text = Integer.toString(num); //ara text és "50", aquest 50 és text.
text = Double.toString(50.25); //ara text és "50.25", aquest 50.25 és text.
text = Boolean.toString(false); //ara text és "false", aquest false és text.
text = Character.toString('c'); //ara text és "c", aquesta c és text.
```

2) De tipus `String` a tipus primitiu

Per obtenir el valor de tipus primitiu d'un text podem fer servir els mètodes estàtics `parseXXX` que inclouen les classes *wrappers* dels tipus primitius (excepte per a `char`).

```
int i = Integer.parseInt("50"); //ara i és 50.
double d = Double.parseDouble("50.25"); //ara d és 50.25.
boolean b = Boolean.parseInt(false); //ara b és false.
```

En el cas del tipus `char`, hem de fer servir el mètode `charAt` de la classe `String` que hem vist anteriorment.

```
String text = "A";
char c = text.charAt(0); //ara c és 'A' i no "A".
```

2.8. *Arrays*

Ja vam parlar molt superficialment dels *arrays* quan vam veure el tipus referència. En aquest apartat volem aprofundir-hi.

Un ***array*** és un tipus de dada estructurada que permet emmagatzemar de manera ordenada un conjunt fix d'elements homogenis (una col·lecció), és a dir, que són del mateix tipus i estan semànticament relacionats entre si. El tipus dels elements emmagatzemats pot ser primitiu o referència. Cada element dins de l'*array* té assignat un índex numèric (o posició) amb la finalitat de trobar-lo unívocament. La primera posició o casella de l'*array* és la número `0`. Per accedir a una casella cal fer servir l'operador `[]`.

Quan l'*array* té una única dimensió se sol anomenar *array*, *vector* o *arranjament*. Quan l'*array* té dues dimensions s'anomena *array de dues dimensions*, *matriu* o *taula*. Quan el nombre de dimensions és més gran s'anomena *array de N dimensions* (en què N és el nombre de dimensions), *array d'arrays* o, simplement, *array multidimensional*. També pot donar-se el cas que el nombre de columnes en un *array* multidimensional sigui desigual, cosa que produeix allò que en anglès es coneix com a *jagged array*.

2.8.1. Declaració i creació en memòria

Un *array* serà un objecte que estarà allotjat en el *heap*. Per poder accedir a aquest objecte tindrem una variable de tipus referència. El nom d'un *array* sol ser un substantiu en plural.

```
int[] grades; //array d'int
short[][] ages; //array de 2 dimensions, o matriu o taula
```

Amb la declaració anterior no es crea cap objecte en el *heap*. Les referències `grades` i `ages` apunten `null`, és a dir, cap adreça de memòria del *heap*. Per crear l'objecte de tipus *array* cal fer servir la paraula reservada `new`:

```
int[] grades = new int[10]; //array de 10 int
short[][] ages = new short[5][6]; //array de 5 files i 6 columnes (6 caselles
//cada fila).
Person[][] people = new Person[2][]; //jagged array de 2 files i amb un número
//de columnes per determinar
```

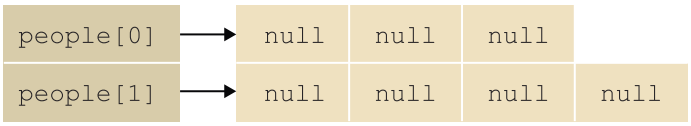
Amb el pas anterior, les referències `grades`, `ages` i `people` apunten l'adreça del *heap* dels seus respectius objectes *array*. No obstant això, cada casella dels tres *arrays* són inicialitzades amb el valor per defecte. En el cas dels tipus primitius, `int` i `short`, el valor per defecte és `0`, mentre que en el cas d'un tipus referència com ara un objecte de la classe `Person`, el valor per defecte és `null`.

Pel que fa a la referència `people` encara falten per inicialitzar les dues files:

```
Person[][] people = new Person[2][];
people[0] = new Person[3]; //accedim a la fila 0 i li assignem un array de
//3 caselles (columnes).
people[1] = new Person[4]; //accedim a la fila 1 i li assignem un array de
//4 caselles (columnes).
```

La referència `people` seria gràficament:

Figura 20



En Java, els *arrays* permeten ser inicialitzats d'una manera especial mitjançant l'ús de claus (`{}`). Per exemple:

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

Amb la línia anterior hem creat un *array* de cinc caselles de tipus `int`, en què el valor de la casella `0` és `5`, el de la casella `1` és `3`, el de la `2` és `6`, el de la `3` és `10` i el de la casella `4` és `7`. La línia anterior és equivalent al codi següent:

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

Per declarar i inicialitzar alhora una matriu podríem fer:

```
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},
{0,8,0,1,2,6}}; //El número d'elements de les files i columnes es dedueix amb
//el número d'elements passats
```

Com hem dit diverses vegades, un *array* en Java és un objecte d'una classe interna de Java. Un exemple de classe interna que Java utilitza per instanciar l'objecte per als *arrays* de tipus primitiu seria (el símbol `[` indica el nombre de dimensions):

Taula 12

Tipus d'*array*	Classe interna que se li assigna
`int[]`	`[I`
`double[]`	`[D`
`double[][]`	`[[D`
`short[]`	`[`
`byte[]`	`[B`
`boolean[]`	`[Z`

2.8.2. Mida d'un *array*

Com hem vist, per inicialitzar un *array* se n'ha d'indicar la mida (*length*) per situar-lo en memòria de manera correcta. Una vegada creada en memòria, la mida no es pot modificar, és a dir, és fixa. És un dels inconvenients més grans que tenen els *arrays*.

Com que un *array* és un objecte d'una classe interna de Java, aquest posa a la disposició dels programadors atributs públics, com, per exemple, `length`. Aquest atribut de només lectura (és `final`)[10] permet saber la longitud de l'*array*.

[10] Més endavant, veurem què és un atribut `final`.

```
int [] grades = new int[5]; //Declara i inicialitza grades amb 5 caselles.
int numGrades = grades.length; //numGrades és 5
```

El valor que pot prendre l'índex que es passa a un *array* va de `0` fins a la mida de `length-1`. En cas de fer servir un valor incorrecte per accedir a un element de l'*array* –per exemple, un enter negatiu o un de més gran que `length-1`–, Java llançarà una excepció de tipus `ArrayIndexOutOfBoundException`, és a dir, Java avisarà que s'ha produït un error.

Vegem un exemple de matriu:

```
int[][] grid = new int[10][8]; //Declara i inicialitza grid amb 10x8 int.
grid.length; //10
grid[0].length; //8
grid[1].length; //8
grid[9].length; //8
```

2.8.3. Copiar el contingut d'un *array* a un altre *array*

Hi ha diferents maneres de copiar el contingut d'un *array*. Totes elles fan una còpia superficial (*shallow copy*) del contingut. Què vol dir això? Doncs que es copia el contingut tal qual dels elements de l'*array* original. Per als tipus primitius, això no és cap problema, però per als de tipus referència sí, ja que el que es copia és l'adreça de memòria dels objectes. Quines conseqüències té això? Imaginem un programa en el qual participen dos *arrays* de `Person`. L'objecte de tipus `Person` que hi ha a la casella amb índex `2` de l'*array* `people1` hi té com a valor de l'atribut `name` el text `"Elena"`. A continuació, creem l'*array* `people2` fent una còpia de `people1` amb qualsevol dels mètodes que explicarem en aquest apartat. Si fem el següent:

```
people2[2].name = "David"; //El valor abans de l'assignació era "Elena"
people1[2].name; //serà "David"; s'ha modificat perquè tant people2[2] com
//people1[2] apunten el mateix objecte, són la mateixa referència.
```

En l'exemple anterior, no s'ha fet una còpia profunda (*deep copy*) del contingut. És a dir, en fer la còpia no s'ha creat un objecte nou de tipus `Person` per a `people2[2]` amb el mateix estat (és a dir, els mateixos valors per a tots els atributs) que l'objecte de `people1[2]`, sinó que directament s'ha copiat la referència, és a dir, l'adreça de memòria a l'objecte.

En aquest cas, si volem fer una còpia profunda, haurem de sobreescriure el mètode `clone` de la classe `Person` (ja ho veurem més endavant).

1) Mètode `clone`. Com que un *array* en Java és un objecte d'una classe interna i, com veurem, tota classe en Java hereta implícitament de la classe arrel `Object`, tot *array* proporciona el mètode `clone` que hereta d'`Object`, que, al seu torn, el sobreescriu.

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

2) Mètode `arraycopy`. La classe pròpia de Java anomenada `System` proporciona un mètode públic anomenat `arraycopy` que permet indicar l'*array* d'origen, l'element a partir del qual es fa la còpia, l'*array* de destinació, l'índex de l'*array* de destinació a partir del qual es copia i el nombre d'elements de l'*array* d'origen que volem copiar.

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int[] grades2 = new int[5];
System.arraycopy(grades1,0,grades2,0,5); //ara grades2 és {1, 2, 3, 4, 5}
```

3) Mètode `copyOf`. Java proporciona el mètode estàtic `copyOf` dins de la classe `Arrays`. Aquest utilitza internament el mètode anterior, `arraycopy`. Aquest mètode crea un *array* nou. Així doncs, podríem fer:

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8};
int[] grades2 = Arrays.copyOf(grades1,8); //ara grades2 és {1,2,3,4,5,6,7,8}
int[] grades3 = Arrays.copyOf(grades1,5); //ara grades3 és {1,2,3,4,5}
int[] grades4 = Arrays.copyOf(grades3,6); //ara grades4 és {1,2,3,4,5,0},
//afegeix un 0 perquè 6 és més gran que la longitud de grades3.
```

4) Mètode `copyOfRange`. La mateixa classe `Arrays` té un altre mètode que crea un nou *array* a partir d'un d'origen amb els elements situats dins d'un rang o interval.

```
int[] grades1 = {1, 2, 3, 4, 5, 6, 7, 8};
int[] grades2 = Arrays.copyOfRange(grades1,1,3); //ara grades2 és {2,3}
//l'últim índex no s'inclou, en aquest cas, l'índex 3 que té valor 4.
```

2.8.4. Ordenar un *array*

Per als tipus primitius, la classe `Arrays` ofereix el mètode `sort`, el qual ordena els elements de l'*array* en ordre ascendent, és a dir, de menor a major.

```
int[] grades = {8, 7, 3, 4, 5, 6, 1, 2};
Arrays.sort(grades); //ara grades és {1,2,3,4,5,6,7,8}
```

Si el tipus d'element desat a l'*array* és de tipus referència (és a dir, un objecte d'una classe), hem de fer servir una sobrecàrrega del mètode `sort` que té com a segon paràmetre un objecte de la classe `Comparator` que indica com s'ha de fer la comparació entre elements amb la finalitat d'ordenar-los correctament.

`Comparator`

`Comparator` és una interfície que proporciona l'API de Java que, a més, és parametritzada. Veurem aquests conceptes més endavant.

```
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 d'edat menor a major
   }
}); //ara people és { (Pau,2), (Marina, 5), (David, 36), (Elena, 41) }
```

2.9. Mètodes (funcions)

Quan un programa es fa complex és molt útil dividir-ne el codi en petites unitats lògiques, de manera que cadascuna d'aquestes unitats sigui capaç de fer una tasca molt concreta del programa. Cadascuna d'aquestes unitats contindrà part del codi del programa i les anomenarem *mètodes* (en llenguatges imperatius reben el nom de *funcions*). Aquesta organització del codi d'un programa en mètodes en facilita la llegibilitat i el manteniment. A més, molt sovint

hi ha trossos de codi que han de ser cridats diverses vegades, per la qual cosa és millor no repetir aquests fragments de codi, sinó posar-los en mètodes i cridar aquests mètodes quan calgui.

2.9.1. Definició d'un mètode

La sintaxi que defineix un mètode és:

```
modificadorDAcces modificadors tipusRetorn nomMetode (tipus param1, ..., tipus paramN){
//Codi
}
```

Com podem veure, el primer element que hem d'indicar és el modificador d'accés (hi aprofundirem més endavant). Serà `public`, `protected`, `private` o cap (cosa que significarà `package-private`).

El segon conjunt de modificadors serà: `final`, `abstract`, `static`, una combinació permesa dels tres (en veurem el significat més endavant) o cap dels tres.

El tercer element és el tipus del valor o element que retorna el mètode. Pot ser un tipus primitiu (`int`, `char`, etc.) o un tipus referència (per exemple, un *array*, un objecte d'una classe, etc.) o el tipus `void` (que significa que no retorna res).

A continuació, s'escriu el nom del mètode, el qual segueix el conveni de ser un verb o sintagma verbal escrit seguint l'estil *lower camel case* (és a dir, la primera paraula en minúscula i les paraules següents amb la inicial en majúscula).

Després ve el llistat de paràmetres que rep (pot ser buit). Per a cada paràmetre n'indicarem el tipus i el nom (en *lower camel case*).

Finalment, dins de les claus `{}` s'escriu el codi del mètode. Per retornar un valor (de tipus primitiu o referència) s'ha de fer servir la paraula reservada `return`. Quan un mètode troba un `return`, l'execució del mètode finalitza retornant el control al punt en què el mètode ha estat cridat/invocat. En els mètodes amb tipus de devolució `void` podem no escriure `return` o escriure-ho. El més habitual és no escriure `return`.

Alguns exemples de mètode són:

```
public int addTwoInt(int number1, int number2){
        return number1 + number2;
}
private static 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(){ //el seu modificador d'accés és package-private
        return new Person("Dummy", 10);
}
```

Estrictament parlant, en Java només el conjunt format pel nom del mètode més els tipus dels paràmetres en l'ordre en què apareixen és conegut com a **firma o signatura del mètode**. Així doncs, les signatures dels mètodes anteriors són:

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

2.9.2. Sobrecàrrega d'un mètode

El llenguatge Java suporta la sobrecàrrega (*overloading*) de mètodes. Això significa que podem tenir dins d'una mateixa classe dos mètodes anomenats exactament igual sempre que les seves signatures no coincideixin, és a dir, es diferenciïn en el nombre i/o en el tipus i/o en l'ordre dels paràmetres.

És molt habitual (segur que alguna vegada ho heu fet) trobar-se amb codi en el qual s'utilitzen noms diferents per a accions similars sobre tipus de paràmetres diferents. Per exemple:

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

La pràctica anterior és molt enutjosa i gens recomanable. Per això, el més recomanable i elegant en Java és fer una sobrecàrrega. Així doncs, l'exemple anterior amb sobrecàrrega seria:

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

Gràcies a la sobrecàrrega, en l'exemple anterior, tenim quatre mètodes anomenats igual –cosa que facilita la llegibilitat del programa– que reben paràmetres diferents i poden fins i tot tenir codis diferents. A més, és més fàcil recordar el nom `add` que quatre noms diferents.

És important adonar-se que Java mira les signatures dels mètodes, és a dir, no considera el tipus de retorn (perquè en Java estrictament no és part de la signatura). Així doncs, la sobrecàrrega següent seria incorrecta i el compilador donaria un error:

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

Per al compilador de Java els dos mètodes anteriors són el mateix mètode perquè tenen la mateixa signatura:

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

En canvi, la sobrecàrrega següent és correcta perquè l'ordre dels paràmetres és diferent. Així doncs, per a Java són dos mètodes diferents:

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

2.9.3. Utilitzar un mètode

El procés per fer servir un mètode és el següent:

1) Des d'un punt del programa es fa una crida o invocació al mètode. Per a això s'utilitza el nom del mètode i la llista de valors que faran servir els paràmetres definits en la signatura. Aquests valors que es fan en la crida/invocació s'anomenen *arguments*.

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

2) El mètode és cridat i rep els arguments en els seus corresponents paràmetres. Així doncs, el primer paràmetre val `8` i el segon val el resultat de `3*2`, és a dir, `6`.

3) El mètode executa el codi que hi ha en el seu cos.

4) Retorna un resultat o no (si és `void`). En aquest exemple, `add` retornarà un valor.

5) El valor és retornat al punt on s'ha cridat el mètode. És a dir, és com si en aquest moment se substituís el mètode pel valor retornat per ell. En l'exemple, la variable `sum` val `14`. A partir d'aquest punt, l'execució del programa continua.

2.9.4. Pas per valor i per referència

Quan es passa un argument de tipus primitiu a un mètode, se'n fa una còpia del valor i és aquesta còpia la que es passa realment al mètode. Així doncs, el mètode que ha estat invocat utilitza el valor copiat, la qual cosa implica que no pot modificar el valor original. Aquest procés és conegut com a **pas per valor**.

```
int value = 8;
int sum = add(value, 3*2);
System.out.print(value); //8
```

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

En l'exemple anterior, el valor de la variable `value`, en aquest cas `8`, es copia en el paràmetre `number1` del mètode `add`. Quan incrementem una unitat, el valor de `number1`, no modifiquem el valor de la variable `value`. Així doncs, quan imprimim per pantalla el valor de `value` en la tercera línia del primer codi, el seu valor continuarà essent `8`, mentre que el valor de `sum` serà `15` (és a dir, `9 + 6`). Si la variable `value` s'hagués dit `number1` –és a dir, igual que el primer paràmetre del mètode `add`–, tampoc no s'hauria modificat el valor de la variable `number1` del codi des del qual es crida el mètode `add`.

Així doncs, amb el pas per valor no podem modificar, des del mètode, el valor de l'element (és a dir, la variable) que ha estat passat com a argument al mètode.

En canvi, quan l'argument passat a un mètode és de tipus referència (per exemple, un *array* o un objecte d'una classe), el pas es fa per referència. Què vol dir això? Doncs que el que es passa al paràmetre del mètode és la referència a l'objecte, és a dir, l'adreça de memòria del *heap* d'aquest objecte. Així doncs, el paràmetre del mètode és exactament una referència idèntica a l'objecte que se li ha passat com a argument, per la qual cosa accedeix realment al mateix objecte, cosa que significa que pot modificar-lo. Aquest procés és conegut com a **pas per referència**.

```
int[] numbers = {1,2,8};
int sum = add(numbers);
System.out.print(numbers[2]); //3
```

```
public int add(int[] numbers){
    numbers[2] = 3;
    return numbers[0]+numbers[1]+numbers[2];
}
```

2.9.5. Nombre variable de paràmetres

Des de JDK 1.5 és possible definir un mètode que no tingui un nombre fix de paràmetres. És a dir, unes vegades rep dos arguments, altres vegades tres, etc. Fins a JDK 1.5, per simular això, el que es feia era passar un *array* (fixeu-vos que en els paràmetres no se n'indica la mida). Tanmateix, això obliga a fer certes tasques addicionals i el codi no expressa el que realment es vol indicar: la variabilitat d'arguments.

Així doncs, JDK 1.5 va introduir la possibilitat d'afegir arguments variables (també coneguts com a *varargs*) gràcies a la sintaxi: `type....`

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

Per utilitzar paràmetres variables hi ha una restricció: només l'últim paràmetre pot ser definit com a variable, és a dir, amb els tres punts (...).

Els tres punts indiquen que els arguments corresponents a l'últim paràmetre poden ser passats com un *array* o una seqüència d'arguments separats per comes. Sigui quin sigui el format, el compilador s'encarregarà d'empaquetar tots els *varargs* en un *array*. Així doncs, l'últim paràmetre l'haurem de tractar com un *array* per obtenir els diferents arguments.

```
public void print(String … texts){ //Versió varargs
        for(String text : texts){
                System.out.println(text);
        }
}
public void print(String text1, String text2){ //Sobrecàrrega amb 2 textos
        System.out.println(text1 +" "+text2);
}
public static void main(String ... args){ // equivalent a String[] args
        print("Hola", "UOC", "¿qué tal? "); //Crida la versió vargs
        print("Hola", "UOC"); //Crida la sobrecàrrega amb 2 textos
        print({"Hola", "UOC"}); //Crida la versió varargs
}
```

Amb l'exemple anterior veiem que:

- De les sobrecàrregues d'un mètode, la versió *varargs* serà l'última que serà cridada. Sempre es crida primer la versió més específica/concreta que la més genèrica (és a dir, *varargs*).

- Des de JDK 1.5, el paràmetre del mètode `main` es pot escriure `String[] args` o `String ... args`.

- En l'exemple anterior, no podríem definir una sobrecàrrega del mètode `print` amb la signatura següent perquè es confondria amb la versió *varargs*:

```
public void print(String[] texts){ //Coincideix amb la versió varargs
```

2.9.6. *Cast* implícit i explícit en els paràmetres

Un mètode pot rebre com a argument un tipus primitiu numèric que sigui de rang inferior al declarat per al paràmetre corresponent. És a dir, si el paràmetre és declarat com a `float`, en fer la crida se li pot passar un argument de tipus `short`, `int` o `float`.

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

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

No obstant això, el cas contrari no és acceptat pel compilador perquè hi ha una pèrdua de precisió. Si tot i així volem passar un tipus numèric superior com a argument a un paràmetre de tipus numèric inferior, haurem de fer un *cast* explícit perquè el compilador sàpiga que som conscients de la pèrdua de precisió que es produirà.

```
float number = 8.0;
int sum = add((int)number, 3*2);
```

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

2.10. Àmbit de les variables

Quan declarem les variables, aquestes existeixen o no –és a dir, poden ser utilitzades o no– en un punt d'un programa depenent d'on hagin estat declarades. Aquesta part, àrea o regió del programa en la qual la variable pot ser utilitzada rep el nom en anglès de *scope* (en català, àmbit, abast o, fins i tot, visibilitat d'una variable).

L'àmbit d'una variable és determinat pels blocs d'instruccions. Recordem que un bloc d'instruccions està definit per una clau inicial `{` i una de final `}`. Així doncs, una variable estarà disponible/visible/accessible en el bloc en el qual s'hagi declarat. És a dir, l'*scope* de la variable coincidirà amb el bloc en el qual s'hagi declarat.

```
public int operate(int a, int b, char op){ //a, b i op són accessibles en
//qualsevol part de la funció/mètode
        if(op == '+'){
                int result = a + b; //result només és accessible en el bloc if.
        }else if(op == '-'){
                return a - b;
        }
        return result; //error: result només és accessible dins de l'if. Ni
        //tan sols és accessible dins de l'else-if.
}
```

Una pràctica molt comuna és crear en els blocs d'iteració de tipus `for` variables locals l'*scope* de les quals sigui el cos del mateix bloc `for`. Vegem-ne un exemple:

```
public int add(int[] numbers){ //numbers és accessible en qualsevol part de la
//funció/mètode
        int result = 0; //result és accessible en qualsevol part del mètode add.
        for(int i = 0; i<numbers.length; i++){ //i només existeix dins del for.
                result += numbers[i];
        }
        System.out.println(i); //error: i no és accessible fora del bloc for.
        return result;
}
```

No obstant això, el codi següent sí que seria correcte:

```
public int add(int[] numbers){ //numbers és accessible en qualsevol part de
//la funció/mètode
        int result = 0, i = 0; //result i i són accessibles en qualsevol part
//del mètode add.
        for(i = 0; i<numbers.length; i++){
                result += numbers[i];
        }
        System.out.println(i); //correcte: i seria igual a numbers.length
        return result;
}
```

Tingueu en compte que l'*scope* d'un bloc afecta l'*scope* dels blocs que tingui imbricats. Així doncs, si declarem una variable en un bloc que té imbricat un segon bloc, aquest segon bloc no podrà declarar una variable anomenada exactament igual, ja que les variables del primer bloc existeixen dins del segon bloc.

```
public void printNumbers(int[] numbers){
        int i = 0;
        for(int i = 0; i<numbers.length; i++){ //error: el bloc for està
//imbricat dins del bloc del mètode printNumbers i aquest ja ha declarat una variable i
            System.out.println(numbers[i]);
        }
}
```

Intenteu endevinar què mostra per pantalla el codi següent:

```
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 codi anterior imprimirà els valors de la casella `0` a la `9` de l'*array* `numbers` a causa del bucle i, posteriorment, imprimirà el nombre `50`. Per què? Doncs, perquè es declara una variable `i` dins del `for` que només existeix dins del `for`

i *a posteriori*, una vegada s'ha acabat el `for` i, per tant, s'ha destruït la variable `i` del `for`, es declara en el bloc superior (és a dir, en l'*scope* del mètode/funció) una variable `i` que a partir d'aquest moment comença a existir (no abans).

3. Orientació a objectes en Java

Abans de continuar llegint és important que llegiu i entengueu, més o menys, els conceptes explicats en els mòduls de teoria, com a mínim, el titulat «Abstracció i encapsulació». En aquesta guia no explicarem què és una classe, ni un mètode, ni un modificador d'accés, etc. El que farem és veure com aquests conceptes es posen en pràctica mitjançant el llenguatge de programació Java. Òbviament, apareixeran alguns conceptes teòrics que detallarem per a aquest llenguatge. El fet de veure com els conceptes teòrics es posen en pràctica ajudarà a acabar d'entendre alguns dubtes que pugueu tenir sobre ells.

3.1. Definir una classe

En aquest apartat veurem quina és la sintaxi mínima per declarar una classe, i també els elements essencials que defineixen tota classe: els membres de la classe (és a dir, atributs i mètodes) i constructors.

3.1.1. Estructura mínima

Per definir una classe en Java hem de tenir present que el seu esquelet mínim és el següent:

```
modificadorDAcces class NomClasse{
        //TODO
}
```

En Java, `public` o `package-private` són els dos únics modificadors d'accés que es poden assignar a una classe principal. De moment, farem que totes les classes siguin `public`.

Per la seva banda, els programadors de Java han de respectar la convenció següent per nomenar les classes:

- El nom és un substantiu o un sintagma nominal en singular.
- Nom escrit en *camel case*, és a dir, totes les paraules que componen el nom amb la primera inicial en majúscula, per exemple: `Ball`, `Person`, `Player`, `MemberVip`, `BankAccount`, etc.

Així doncs, el mínim per definir la classe `Person` en Java seria:

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

3.1.2. Membres de la classe

Com ja sabeu, les classes tenen membres. Hi ha dos tipus de membres de la classe: els atributs i els mètodes. Així doncs, cal definir-los. El més habitual és definir primer els atributs i després, els mètodes (i al mig, els constructors). És a dir:

```
public class Person{
        //Atributs primer
        //Constructors després
        //i Mètodes al final
}
```

Atributs

Els atributs es defineixen igual que les variables (fins i tot respecten les convencions utilitzades en les variables per als noms), però cal indicar-ne a més el modificador d'accés. De fet, podeu pensar-hi com a variables globals dins de la classe. És a dir, tots els atributs d'una classe, independentment del modificador d'accés que se'ls assignin, són visibles en qualsevol part de la classe en la qual estan definits. L'única diferència amb les variables és, com hem dit, que als atributs cal assignar-los un modificador d'accés i a les variables no. Vegem-ne alguns exemples:

Nomenclatura per als atributs

A més del terme *atribut* (*attribute*), també s'empra el terme *camp* (*field*).

```
public class Person{
        private String name, surname;
        private int age = 0; //podria haver estat byte per ocupar menys espai
        private int height; //entenem que s'indica en centímetres
        private float weight;
}
```

Si ens hi fixem, abans de la declaració de l'atribut li indiquem el modificador d'accés que volem que s'apliqui (observeu les paraules amb vermell). El més habitual és declarar els atributs com a `private`. Si sabem que la classe participarà en una relació d'herència, també pot ser una bona opció `protected` (dependrà del nostre disseny).

Què succeeix si no s'indica un modificador d'accés? S'entén que l'atribut té un modificador d'accés `default`, o també anomenat `package-private` (un modificador d'accés especial de Java).

Modificadors d'accés

Més endavant prestarem especial atenció als diferents modificadors d'accés que proporciona Java. Ara com ara, només considereu `public` i `private`.

Observem la declaració de l'atribut `age` de la classe `Person` en l'exemple anterior. Veieu alguna diferència amb la resta d'atributs? Efectivament, se li ha assignat un valor, concretament, zero. Aquesta inicialització es podria haver fet en un dels constructors de `Person` (ho veurem més endavant), però, en ser

tan simple, l'hem feta a la declaració mateix. Si la inicialització de l'atribut `age` hagués necessitat alguna lògica (per exemple, fer servir un bucle), no l'hauríem pogut fer a la declaració. La inicialització en la declaració només és possible si és simple.

De la mateixa manera, si un atribut no és inicialitzat enlloc de la classe (és a dir, declaració, constructor, bloc d'inicialització, etc.), el compilador li assignarà el valor per defecte del tipus que se li ha assignat, per exemple `0` per a atributs de tipus numèric, `false` per a `boolean` i `null` per a tipus referència.

Per acabar, cal indicar que els atributs d'una classe, a diferència de les variables, es creen dins de l'objecte. Per tant, pel que fa a la ubicació en memòria, els atributs estan en el *heap* (com els objectes), no en l'*stack*.

Mètodes

Tot el que hem comentat en l'apartat «Mètodes (funcions)» serveix ara que fem servir Java orientat a objectes. No obstant això, explicarem algunes qüestions addicionals que apareixen en utilitzar classes.

La primera d'elles és que el modificador `static` ja no el farem servir si no és estrictament necessari. Fins ara utilitzàvem aquest modificador perquè empràvem Java en un paradigma de programació estructurada i, sense `static`, no podíem cridar els mètodes des del `main`.

En segon lloc, veurem una situació que només pot donar-se quan utilitzem atributs dins dels mètodes. Mireu el mètode *setter* següent:

```
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 mètode `setAge` anterior comprova si el paràmetre `ageNew` és negatiu. Si ho és, escriu un missatge d'error per pantalla i assigna el valor `0` a l'atribut `age`. En cas contrari, assigna a l'atribut `age` el valor del paràmetre `ageNew`. No obstant això, l'ideal hauria estat anomenar el paràmetre igual que l'atribut, per saber que estan relacionats. És a dir, el codi «ideal» hauria estat:

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

Llavors, com sap el compilador (i nosaltres) en cada sentència escrita amb vermell si `age` és l'atribut o el paràmetre? El cas més evident de confusió és l'assignació de l'`else: age = age`. L'`age` del costat esquerre de l'assignació és l'atribut o el paràmetre? I l'`age` de la dreta? Així doncs, per evitar problemes, molts programadors posen al paràmetre del mètode el mateix nom de l'atribut de la classe més un afegitó. És per això que la declaració inicial del mètode `setAge` era:

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

És obvi que amb aquesta solució d'afegir un afegitó (en l'exemple, `"New"`) no hi ha cap dubte per al compilador ni per a nosaltres de què és què. Malgrat que podem fer servir una solució com l'anterior, el més habitual i millor (com hem comentat) és utilitzar com a noms dels paràmetres dels mètodes els mateixos noms que els dels atributs de la classe, en aquest cas, `age`. Per resoldre el conflicte de noms, Java ens proporciona una paraula reservada anomenada `this`. Així doncs, considerant una versió simplificada del mètode `setAge` com la següent:

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

El seu equivalent amb `this` seria:

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

El `this` fa d'afegitó i indica que allò que porta `this` pertany a l'objecte/classe i no és ni un paràmetre ni una variable declarats dins del mètode. Concretament `this` fa referència a l'objecte (o instància) que és del tipus de la classe. Per tant, gràcies a `this`, diem que a l'atribut `age` de l'objecte (o si us és més fàcil d'entendre, de la classe) cal assignar-li el valor del paràmetre `age` (anomenat `ageNew` en la versió anterior del codi).

3.1.3. Constructor

En Java podem definir tants constructors com vulguem. Fins i tot podem no declarar cap constructor. En aquest últim cas, el compilador de Java ho detectarà i en crearà un per defecte de manera transparent a nosaltres que no farà res, només permetre que es puguin instanciar objectes d'aquesta classe.

Així doncs, amb el codi anterior de la classe `Person` ja tenim un constructor per defecte. No serà creat per nosaltres, sinó que ho farà el compilador. És a dir, el codi anterior és equivalent a haver escrit el següent:

```
public class Person{
        private String name, surname;
        private int age;
        private int height;
        private float weight;

        //Constructor per defecte
        public Person(){
        }
}
```

Com us en deveu haver adonat, el constructor, a diferència dels mètodes, no defineix un tipus de retorn, ni tan sols `void`. Gràcies a aquesta diferència, el compilador pot saber que es tracta d'un constructor i no d'un mètode. A més, obligatòriament, el nom del constructor ha de coincidir amb el nom de la classe.

Constructor per defecte

Vegem com podríem emplenar un constructor per defecte.

```
public class Person{
        private String name, surname;
        private int age;
        private int height;
        private float weight;

        //Constructor per defecte
        public Person(){
                name = "David";
                surname = "García";
                age = 36;
                height = 172;
                weight = 66;
        }
}
```

El constructor per defecte anterior és correcte. Fins i tot, no faria falta inicialitzar tots els atributs (fixeu-vos que hem eliminat la inicialització en la declaració d'`age`... no té sentit duplicar). Ara bé, és interessant i una bona pràctica utilitzar els mètodes *setter* i *getter* dins de la classe mateix per tal d'aconseguir la màxima consistència, estabilitat i robustesa dels atributs.[11] Així doncs, una millor codificació del constructor anterior seria:

[11] Recordeu el que hem comentat en l'apartat «Protecció de les dades» del mòdul «Abstracció i encapsulació».

```
public Person(){
    setName("David");
    setSurname("García");
    setAge(36);
    setHeight(172);
    setWeight(66);
}
```

El fet de fer servir els *setters* ens ajuda a crear un punt comú de comprovacions i assignacions. Com recordareu, dins del mètode `setAge` es comprova que l'edat passada com a argument no sigui inferior a `0`. Si ho és, imprimim un missatge d'error i assignem el valor `0` a l'atribut `age`. En cas contrari, assignem el valor que ens faciliten.

Constructor amb arguments

Com hem dit, les nostres classes poden tenir tants constructors com necessitem. Per a això, el que haurem de fer és sobrecarregar el constructor. Vegem dos exemples de constructor amb arguments (o constructor parametritzat).

```
public class Person{
        private String name, surname;
        private int age;
        private int height;
        private float weight;

        //Constructor per defecte
        public Person(){
                setName("David");
                setSurname("García");
                setAge(36);
                setHeight(172);
                setWeight(66);
        }

        //Constructor amb arguments 1
        public Person(String name, String surname, int age){
                setName(name);
                setSurname(surname);
                setAge(age);
                setHeight(172);
                setWeight(66);
        }

        //Constructor amb arguments 2
        public Person(String name, String surname, int age, int height, float weight){
                setName(name);
                setSurname(surname);
                setAge(age);
                setHeight(height);
                setWeight(weight);
        }

        //TODO: Aquí estaria el codi de setName, setSurname, setAge, setHeight i setWeight
}
```

Si ens hi fixem, el primer constructor amb arguments només té tres paràmetres, mentre que el segon en té cinc (tants com els atributs que té la classe). Us imagineu què hauria passat si no cridéssim els mètodes *setters* en els tres constructors? Doncs que en tots tres repetiríem el mateix codi de comprovació que ara tenim centralitzat en el mètode `setAge`. Una vegada més, es veu la importància de cridar els *getters* i *setters* també des de dins de la classe. Així i

tot, podeu veure que els tres constructors tenen un codi comú: cridar els *setters* dels atributs. Per què repetir línies de codi idèntiques? Java també es va adonar que això era molt comú, per la qual cosa va crear un parell de maneres de reduir la repetició de codi en els constructors.

Mètode especial `this`

Java posa a la disposició dels programadors un mètode especial anomenat `this`. Aquest mètode té la mateixa signatura que qualsevol dels constructors que hem declarat. S'utilitza en els constructors per cridar altres constructors. Té la restricció que, si es vol utilitzar, la crida al mètode `this` ha de ser la primera instrucció que escriguem en el constructor en què l'utilitzarem. Vegem el codi anterior simplificat gràcies a l'ús del mètode especial `this`.

```
public class Person{
	private String name, surname;
	private int age;
	private int height;
	private float weight;

	//Constructor per defecte
	public Person(){
		this("David","García",36,172,66);
		//Aquí podríem afegir les instruccions que volguéssim
	}

	//Constructor amb arguments 1
	public Person(String name, String surname, int age){
		this(name,surname,age,172,66);
		//Aquí podríem afegir les instruccions que volguéssim
	}

	//Constructor amb arguments 2
	public Person(String name, String surname, int age, int height, float weight){
		setName(name);
		setSurname(surname);
		setAge(age);
		setHeight(height);
		setWeight(weight);
	}

	//TODO: Aquí estaria el codi de setName, setSurname, setAge, setHeight i setWeight
}
```

Si ens hi fixem, el canvi s'ha produït en els dos primers constructors (observeu les dues instruccions en vermell). El que hem fet és cridar l'últim constructor (el més genèric) des dels altres dos constructors. Podeu veure que la signatura del mètode `this` coincideix amb la de l'últim constructor amb arguments.

Bloc d'inicialització d'instància

Per als casos en els quals la inicialització dels atributs per als objectes (instàncies) creats en una classe tenen valors coneguts *a priori*, Java proporciona el que es coneix com a *bloc d'inicialització* (*initializer block*).

Imagineu, per exemple, que l'edat inicial sempre sigui `36` per a tots els objectes creats per a la classe `Person`. Llavors podríem fer:

```
public class Person{
	private String name, surname;
	private int age;
	private int height;
	private float weight;

	//Això és un bloc d'inicialització. Tan simple com obrir i tancar claus {}
	{
		setAge(36); //Podríem haver fet age = 36; (millor utilitzar el setter)
	}

	//Constructor per defecte
	public Person(){
		this("David","García",172,66);
		//Aquí podríem afegir les instruccions que volguéssim
	}

	//Constructor amb arguments 1
	public Person(String name, String surname){
		this(name,surname,172,66);
		//Aquí podríem afegir les instruccions que volguéssim
	}

	//Constructor amb arguments 2
	public Person(String name, String surname, int height, float weight){
		setName(name);
		setSurname(surname);
		setHeight(height);
		setWeight(weight);
	}

	//TODO: Aquí estaria el codi de setName, setSurname, setAge, setHeight i setWeight
}
```

Quan es cridi qualsevol dels tres constructors anteriors, el primer que s'executarà serà el bloc d'inicialització. És a dir, primer es cridarà el mètode `setAge` amb l'argument `36` i, a continuació, la resta de codi del constructor cridat. Podeu observar que el codi ha canviat una mica: on s'han introduït canvis està ressaltat amb vermell.

> Així doncs, ha de quedar clar que el bloc d'inicialització d'instància d'una classe s'executa abans que els constructors d'aquesta classe.

Una classe pot tenir tants blocs d'inicialització d'instància com calgui. En aquest cas, l'ordre d'execució serà *top-down*, és a dir, el que aparegui primer en el codi serà el primer que s'executi, i així successivament.

En resum:

- Cada bloc d'inicialització d'instància s'executa cada vegada que s'instancia un objecte nou i s'executa per a aquest objecte en creació.

- Si hi ha més d'un bloc d'inicialització d'instància, l'ordre d'execució comença pel que apareix primer en el codi de dalt a baix.

- Si la classe en la qual hi ha un bloc d'inicialització hereta d'una altra classe, primer es crida els blocs d'inicialització d'instància de la classe pare, després el constructor de la classe pare, després els blocs d'inicialització de la classe per a la qual creem l'objecte i, finalment, el codi del constructor que hem fet servir per instanciar l'objecte.

- Si un atribut s'ha inicialitzat en la declaració del mateix atribut, aquesta inicialització es fa abans de la crida al bloc d'inicialització d'instància.

Herència i classe pare

El concepte d'*herència* i *classe pare* el veurem més endavant tant en els apunts teòrics com en aquesta guia.

3.2. Objectes

3.2.1. Instanciar (creant objectes)

Com ja sabeu, un objecte és un exemplar d'una classe. Així doncs, a l'hora de crear un objecte, hem de seguir els passos següents:

1) Declarar una variable o atribut de tipus de la classe (en termes generals és un tipus referència):

```
Person david;
```

En aquest moment, la variable/atribut `david` no apunta cap objecte; la seva referència/valor és `null`.

2) Instanciar l'objecte. És a dir, crear un objecte a partir d'una classe.

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

Com heu vist en l'apartat «Tipus» i en el codi just anterior, per instanciar/crear un objecte nou (és a dir, un tipus referència) hem d'escriure la paraula reservada `new` seguida d'una crida a un dels constructors definits en la classe. En aquest cas, hem utilitzat el constructor per defecte, però també podríem haver emprat qualsevol dels dos constructors parametritzats. Per exemple:

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

En aquest moment ja tenim un objecte de tipus `Person` situat en memòria (*heap*) i amb una variable/atribut de tipus referència que apunta aquest objecte.

Els passos 1 i 2 es poden agrupar en un únic pas:

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

Com que el segon pas s'anomena *instanciar*, ara entendreu per què els objectes també s'anomenen *instàncies*.

3.2.2. Utilitzar un objecte (missatges)

Un cop hem creat un objecte i està referenciat per una variable/atribut, el més lògic és fer-ne alguna cosa, normalment accedir a un atribut o a un mètode. Per a això farem servir missatges. La forma d'un missatge és:

```
referencia.membre
```

en què `referencia` pot ser una variable o un atribut, i `membre` pot ser tant un atribut com un mètode de la classe a la qual pertany l'objecte.

```
Person elena = new Person("Elena","Lázaro");
elena.setAge(41); //cridem el mètode setAge de l'objecte que apunta elena.
Person david = new Person();
david.setAge(elena.getAge()); //cridem el mètode setAge de l'objecte que
//apunta david i li passem com a argument el valor retornat pel mètode
//setAge de l'objecte que apunta elena.
System.out.println(david.getAge()); //Imprimeix 41
```

3.3. Modificadors d'accés

Deveu haver vist que en els exemples anteriors els atributs, mètodes i constructors tenien la paraula `public` o `private` a l'inici de la seva declaració. Com bé sabeu, tant `public` com `private` són dos modificadors d'accés, la funcionalitat del quals és determinar l'accessibilitat/visibilitat d'un element. Quan parlem de l'ús de modificadors d'accés, ens referim al fet que, si tenim un objecte d'una classe `A` dins d'una altra classe `B`, segons com estiguin declarats els membres de la classe `A`, s'hi podrà accedir directament mitjançant un missatge o no des de `B` fent servir l'objecte de la classe `A`.

```
public class B{
       A objectA;
       public B(){
              objectA = new A();
              objectA.member; //això serà correcte o no depenent del
//modificador d'accés amb el qual sigui declarat "member" a la classe A.
       }
}
```

> Tots els membres d'una classe, independentment del seu modificador d'accés, són accessibles directament des de dins de la mateixa classe.

En Java hi ha quatre modificadors d'accés: `public`, `protected`, `private` i `package-private`. Cadascun d'aquests quatre modificadors permet que un element sigui més o menys accessible des de fora de la mateixa classe, i `public` i `private` en són els extrems.

En funció de l'element, se li podran assignar uns modificadors d'accés o uns altres, principalment els exposats en la taula 13.

Taula 13. Modificadors d'accés aplicables als diferents elements d'un programa

Tipus de nivell	Tipus d'element	Modificadors d'accés permès
Alt nivell (o element contenidor)	Classes, interfícies i enumeracions (`enum`)	`public` `package-private`

Membres	Atributs, mètodes, constructors, classes imbricades i enumeracions (`enum`)	`public`
		`protected`
		`private`
		`package-private`

Modificador `package-private`

Per assignar aquest modificador, no cal escriure cap modificador en declarar l'element. És a dir, quan no s'escriu ni `public` ni `protected` ni `private`, Java entén que el modificador que es vol assignar a l'element és `package-private`. Així doncs, mai no s'ha d'escriure la paraula composta `package-private`. Per aquest motiu, aquest modificador també es coneix com a *no modifier* o *default*.

Com podem veure, en declarar, per exemple, una classe, només podem assignar-li o bé el modificador `public` o bé el modificador `package-private`. No obstant això, si declarem una classe dins d'una altra classe, és a dir, declarem una classe imbricada (*nested class*), aquesta classe es comporta com un membre de la classe contenidora i, per tant, podem assignar-li qualsevol dels quatre modificadors d'accés que proporciona Java.

Un cop arribats a aquest punt, cal saber les diferències entre els quatre modificadors. És a dir, quines implicacions té cadascun d'ells. Els comentarem suposant que dins d'una classe `B` tenim un objecte de la classe `A` amb un membre al qual volem accedir.

1) **`public`**: indica que es pot accedir a l'element des de qualsevol part del programa. Així doncs, si dins de la classe `B` volem accedir a un membre públic d'un objecte de la classe `A`, ho podrem fer des de `B` de la manera següent:

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

2) **`private`**: indica que només es pot accedir al membre des de dins de la mateixa classe que el declara. Així doncs, si intentem accedir a un membre privat de la classe `A` mitjançant un objecte de la classe `A` pertanyent a la classe `B`, obtindrem un error de compilació:

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

Acabem de veure els dos modificadors extrems. És a dir, `public` és el modificador més permissiu i `private`, el més restrictiu. Ens falten dos modificadors d'accés que són molt similars entre si i que només difereixen en el cas que hi hagi una relació d'herència entre les classes implicades (és a dir, una relació superclasse-subclasse).

3) **`package-private`**: indica que només es pot accedir al membre declarat com a `package-private` (o `default`) des d'elements (per exemple, una classe) que estiguin dins del mateix *package* que l'element en el qual hi ha el membre `package-private`. Dit d'una altra manera, allò que ens diu el modificador `package-private` és que els elements declarats com a tals (és a dir, sense modificador) són accessibles/visibles directament per a tots els elements declarats dins del mateix *package*. Així doncs, només si la classe `B` i la classe `A` pertanyen al mateix *package*, la sentència següent a la classe `B` serà correcta:

```
objectA.defaultMember;
```

4) **`protected`**: a més del que permet el modificador `package-private`, també permet que els membres declarats com a `protected` siguin visibles/accessibles des de qualsevol subclasse de la classe en la qual s'ha declarat el membre `protected`. Això sí, des d'una subclasse situada en un *package* diferent del de la seva superclasse, el membre declarat com a `package-private` en la superclasse només és accessible per mitjà d'una referència/objecte que sigui com a mínim del mateix tipus que la subclasse. Així doncs, `protected` és una mica menys restrictiu que `package-private`.

Taula 14. Resum

Modificador	Des del contenidor mateix (p. ex., classe, interfície o enum.)	Des del mateix *package*	Subclasse en un altre *package*	La resta d'elements
`public`	Sí	Sí	Sí	Sí
`protected`	Sí	Sí	Sí (només mitjançant un objecte de la mateixa subclasse o subclasse d'aquesta)	No
`package-private` (sense modificador)	Sí	Sí	No	No
`private`	Sí	No	No	No

Com ja hem comentat i com es pot observar en la columna «Des del contenidor mateix» de la taula 14, qualsevol membre (atribut o mètode) és accessible des de dins de l'element que el declara sense necessitat, òbviament, d'instanciar cap objecte.

Per exemple, un atribut declarat en la classe `A`, independentment del modificador d'accés assignat, sempre és accessible des de qualsevol punt de la classe `A` (per exemple, qualsevol mètode declarat en la classe `A`). Així doncs, qualsevol element contenidor sempre té accés a tots els seus membres. Això ha de quedar molt clar.

En canvi, tal com indica la taula 14, una classe `B` creada dins del *package* `P1` no podrà accedir mitjançant un objecte de la classe `A` a un membre `private` declarat en la classe `A` del mateix *package* `P1`. No obstant això, si el mateix element hagués estat declarat com a `public`, `protected` o `package-private`, la classe `B` sí que hi hauria pogut accedir i utilitzar-lo.

Package i subclasse

Ara com ara no hem de preocupar-nos pels conceptes *package* i subclasse (ni per superclasse i herència). El primer el veurem en aquesta guia i el segon l'explicarem tant en els apunts teòrics com en aquesta guia.

Si continuem analitzant la taula 14, veiem que es pot accedir als membres declarats `public` o `protected` des de qualsevol de les seves subclasses, tant si estan en el mateix *package* com si no. Ara, cal fer una apreciació important per al cas `protected`. Quan la superclasse (per exemple, `A`) i la subclasse (per exemple, `B`) pertanyen a *packages* diferents, no podem accedir a un membre `protected` de la superclasse dins de la subclasse amb un objecte de la superclasse: `objectA.protectedMember;`. En canvi, sí que hi podrem accedir mitjançant un objecte de la mateixa subclasse `B` o d'altres subclasses de `B`. Per aquest motiu, hem dit en la columna «Subclasse en un altre *package*» de la taula 14 que un membre `protected` és accessible des d'una subclasse de la classe que el declara per mitjà d'una referència que sigui, almenys, del mateix tipus que la subclasse. Això és així per assegurar que només s'accedeix als membres d'instància `protected` per la jerarquia de classes a la qual pertanyen (subclasse i les seves subclasses), excloent-hi classes germanes (altres classes que hereten de la mateixa superclasse que declara el membre `protected`).

Finalment, un element declarat com a `public` és visible per a qualsevol altre element del programa. És a dir, si tenim un mètode declarat com a `public` en una classe `A`, hi podrem accedir des d'un objecte `A` declarat a una altra classe `B`, independentment de si `A` i `B` estan en el mateix *package* o no, i de si entre `A` i `B` hi ha una relació d'herència (superclasse-subclasse).

Vegem-ne alguns exemples per reforçar aquests conceptes. Farem servir la classe `A` següent:

```
package mypackage; //La classe A pertany al package mypackage
public class A{
       public int age;
       private String name;
       protected double weight;
       double height;
}
```

Imaginem que tenim la classe `B` següent amb un objecte de la classe `A` anterior:

```
package mypackage; //La classe B pertany al package mypackage
public class B{
	A objectA;

	public B(){
		objectA = new A();
		objectA.age = 5; //correcte perquè "age" és public
		objectA.name = "David"; //incorrecte perquè "name" és private
		objectA.weight = 63.2; //correcte perquè "weight" és protected i
//les classes A i B estan en el mateix package
		objectA.height = 1.73; //correcte perquè "height" és package-
//private i les classes A i B estan en el mateix package
	}
}
```

Imaginem que ara tenim la classe `C` següent amb un objecte de la classe `A` anterior:

```
package anotherpackage; //La classe C pertany al package anotherpackage
import mypackage.A; //ara com ara no facis cas d'aquesta sentència

public class C{
	A objectA;

	public C(){
		objectA = new A();
		objectA.age = 5; //correcte perquè "age" és public
		objectA.name = "David"; //incorrecte perquè "name" és private
		objectA.weight = 63.2; //incorrecte perquè "weight" és protected
//i les classes A i C no estan en el mateix package
		objectA.height = 1.73; //incorrecte perquè "height" és package-
//private i les classes A i C no estan en el mateix package
	}
}
```

Ara imaginem que la classe `D` és una subclasse (concepte que veurem més endavant) de la classe `A` i té un objecte de la classe `A`. Ambdues classes pertanyen al mateix *package*.

```
package mypackage;
public class D extends A{ //ara com ara no facis cas a "extends A"
	A objectA;

	public D(){
		super(); //ara com ara no facis cas a aquest mètode especial.
		objectA = new A();
		objectA.age = 5; //correcte perquè "age" és public
		objectA.name = "David"; //incorrecte perquè "name" és private
		objectA.weight = 63.2; //correcte perquè "weight" és protected i
//les classes A i D estan en el mateix package
		objectA.height = 1.73; //correcte perquè "height" és package-
//private i les classes A i D estan en el mateix package
	}
}
```

Per acabar, vegem una classe `E` que és subclasse de `A`, però ambdues classes pertanyen a *packages* diferents.

```
package anotherpackage;
import mypackage.A; //ara com ara no facis cas a aqueesta sentència

public class E extends A{ //ara com ara no facis cas a "extends A"
       A objectA;

       public E(){
              super(); //ara com ara no facis cas a aquest mètode especial.
              objectA = new A();
              objectA.age = 5; //correcte perquè "age" és public
                 ectA.name = "David"; //incorrecte perquè "name" és private
              objectA.weight = 63.2; //incorrecte perquè "weight" és protected
// i A i E no estan en el mateix package; i utilitzem un objecte de la superclasse A
       objectA.height = 1.73; //incorrecte perquè "height" és package-
//private i les classes A i E no estan en el mateix package
       }
}
```

Vegem una variant de la classe `E` anterior:

```
package anotherpackage;
import mypackage.A; //ara com ara no facis cas a aquesta sentència

public class E extends A{ //ara com ara no facis cas a "extends A"
       A objectA;
       E object E;

       public E(){
              super(); //ara com ara no facis cas a aquest mètode especial.
              objectA = new A();
              objectA.age = 5; //correcte perquè "age" és public
              objectA.name = "David"; //incorrecte perquè "name" és private
              objectA.weight = 63.2; //incorrecte perquè "weight" és protected
// i A i E no estan en el mateix package; i utilitzem un objecte de la superclasse A
              objectA.height = 1.73;  //incorrecte perquè "height" és package-
//private i les classes A i E no estan en el mateix package

              objectE =  new E();
              objectE.age = 5; //correcte perquè "age" és public
              objectE.name = "David"; //incorrecte perquè "name" és private
              objectE.weight = 63.2; //correcte perquè "weight" és protected i
//accedim amb un objecte E en què E és subclasse de la classe A. Podríem haver
//accedit amb un objecte el tipus del qual fos una subclasse de E.
              objectE.height = 1.73; //incorrecte perquè "height" és package-
//private i les classes A i E no estan en el mateix package
       }
}
```

3.4. Destructor d'una classe

En Java no hi ha un destructor com a tal. No obstant això, proporciona un mètode anomenat `finalize` que exerceix el rol de destructor. Aquest mètode `finalize` el tenen totes les classes, ja que, com comentarem més endavant, en Java totes les classes hereten implícitament de la classe arrel `Object`. Així doncs, `finalize` és un mètode de la classe `Object` que qualsevol classe hereta, tant si vol com si no.

El mètode `finalize` que proporciona la classe `Object` no fa res, per la qual cosa pot ser sobreescrit (veurem què significa això). Bàsicament hem de quedar-nos amb la idea que si volem que el mètode `finalize` faci alguna cosa,

hem d'escriure-ho explícitament i afegir un codi en el seu cos. Si no hem de fer res especial, llavors no és obligatori sobreescriure-ho (és a dir, afegir codi). El més habitual és no escriure el mètode `finalize`.

La manera d'explicitar (sobreescriure) el mètode `finalize` seria afegir el codi següent a la nostra classe, com si fos un mètode més (normalment s'hi posa l'últim per trobar-lo fàcilment):

```
public void finalize(){
        //El teu codi aquí
}
```

Alguns motius pels quals podem utilitzar el mètode `finalize` són: alliberar recursos compartits, tancar connexions (per exemple, bases de dades o *sockets*), etc. No obstant això, tingueu en compte que aquest mètode és cridat automàticament dins del procés conegut com a *garbage collection* quan la JVM ho considera oportú. És a dir, saber quan el mètode `finalize` és realment cridat és quelcom incert. Així doncs, no hem de dependre d'aquest mètode per fer algunes gestions crítiques. El que sí que hem de tenir present és que el mètode `finalize` no es cridarà fins que un objecte no tingui cap variable/atribut de tipus referència que l'apunti.

Si un objecte no és apuntat per cap referència, se'n cridarà el mètode `finalize`; quan? És una incògnita.

La JVM té un *garbage collector* que periòdicament allibera la memòria utilitzada per objectes que mai més no estan referenciats. El *garbage collector* fa aquesta tasca de manera automàtica quan creu que és el moment adequat.

```
Person p1 = new Person(); //creem un objecte Person i l'assignem a la
//variable p1
Person p2 = p1; //p2 apunta el mateix objecte que p1
p1 = null; //p1 ja no apunta l'objecte Person creat en la primera instrucció.
//No obstant això, no es cridarà el mètode finalize de l'objecte, ja que encara està
//essent apuntat per la variable p2.
p2 = null;
```

Després de l'última instrucció del codi anterior (`p2 = null;`), tant `p1` com `p2` no apunten cap objecte. L'objecte `Person` creat en la primera instrucció no està referenciat/apuntat per ningú, així que es cridarà automàticament (com a programadors no hem de fer res) el seu mètode `finalize` i s'alliberarà l'espai que ocupa en memòria (*heap*). Quan es farà aquesta crida? Quan el *garbage collector* ho consideri oportú.

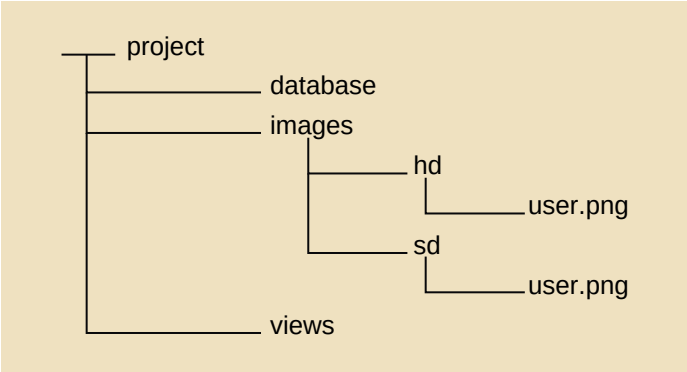
Com podeu veure, la gestió de la memòria en Java és molt més senzilla que en altres llenguatges com ara C++. Això també implica menys llibertat a l'hora de «jugar» amb la memòria de l'ordinador i poder optimitzar processos.

3.5. *Packages*

3.5.1. Què és un *package* i quina n'és la utilitat?

En Java hi ha un tipus de contenidor abstracte anomenat *package* que permet organitzar elements d'alt nivell (és a dir, classes, interfícies i enumeracions) que estan relacionats entre si. La manera més senzilla d'entendre què és un *package* seria fent una analogia amb un sistema de directoris. Per exemple, imaginem que tenim un projecte amb l'estructura de directoris que mostra la figura 21.

Figura 21



Si ens hi fixem, tant en les subcarpetes `hd` com `sd`, tenim dues imatges anomenades exactament igual: `user.png`. Com podem distingir-les unívocament? Per mitjà de la seva ruta (*path*): `project/images/hd/user.png` és diferent de `project/images/sd/user.png`.

> Doncs bé, els ***packages*** en Java fan exactament això: organitzar el nostre codi perquè sigui més fàcil trobar el que volem de manera unívoca i, a més, afegir informació lògica al nostre programari.

Així doncs, el primer que hem de saber és que en un programa podem tenir moltes classes escrites i algunes d'elles poden fins i tot anomenar-se igual.

Per exemple, l'API (*Application Programming Interface*) de Java (https://docs.oracle.com/en/java/javase/13/docs/api/allpackages-index.html), la qual no deixa de ser una llibreria amb centenars de classes ja creades que ens proporciona el llenguatge per programar el nostre propi programari de manera més senzilla, està organitzada en *packages*.

En l'API de Java hi ha elements anomenats exactament igual, però que pertanyen a *packages* diferents. Per exemple, `List` existeix tant en `java.util` com en `java.awt`. En aquest cas concret, a més, es dona la circumstància que `List` és una interfície en `java.util`, mentre que en `java.awt` és una classe: són

dues coses diferents que es diuen igual! A més, el *package* `java.awt` inclou un conjunt de classes per crear interfícies d'usuari, mentre que `java.util` conté un conjunt d'elements relacionats amb col·leccions (és a dir, estructures de dades): llistes, cues, piles, taules de *hash*, etc.

Per dir al compilador de Java quina `List` ha de fer servir al nostre programa, hem d'indicar-ho fent un `import`. Per exemple:

```
import java.awt.List; //Indiquem que volem la classe List de java.awt
import java.awt.Frame; //Indiquem que volem la classe Frame de java.awt
//(hi ha un altre package per a interfícies gràfiques anomenat swing que té 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);
       }
}
```

Així doncs, un *package* serveix per:

- Empaquetar/agrupar elements (classes, interfícies i enumeracions) que estan relacionats.

- Crear *namespaces* i evitar conflictes quan dos o més elements es diuen exactament igual. En estar els elements homònims en *packages* diferents, aquests són identificats unívocament. Per tant, el nom complet de l'element està compost pel nom *package* + el nom de l'element en si. Això és el que es coneix com a *fully qualified name*, el qual permet desambiguar l'element al qual ens referim. Així doncs, els *fully qualified names* dels `List` anteriors són: `java.awt.List` i `java.util.List`.

- Permetre accés especial entre els elements que pertanyen a un mateix *package*, gràcies al modificador d'accés `package-private`.

3.5.2. Crear un *package*

Crear un *package* en un IDE com Eclipse és tan senzill com clicar amb el botó dret amb el ratolí el directori `src` i escollir l'opció `New -> Package`. Tots els elements (classes, interfícies i enumeracions) que pertanyen a un *package* han de tenir com a **primera línia de codi** la sentència següent:

```
package packageName;
```

En què `packageName` és el nom del paquet al qual pertany l'element en el qual escrivim aquesta sentència.

Així doncs, imaginem que tenim dues classes que volem que pertanyin al *package* `mypackage`. El codi seria:

```
package mypackage;

public class A{
        //Codi
}
```

```
package mypackage;

public class B{
        //Codi
}
```

Amb el codi anterior diem que les classes `A` i `B` pertanyen al *package* anomenat `mypackage`. En un IDE com Eclipse si afegim la sentència `package` com a primera línia del codi d'un element, l'IDE ens donarà un error si l'element no està dins del *package* que acabem d'indicar. L'IDE ens suggerirà com a solució crear el *package* i incloure dins l'element. També podem crear manualment un *package* (`New --> Package`) i afegir (per exemple, arrossegant) els elements que pertanyen a aquest *package*. En aquest últim cas, l'IDE modificarà el nom del *package* dins del codi de cadascun dels elements perquè tot sigui correcte.

Tots els elements als quals no escrivim la sentència `package`, és a dir, no indiquem explícitament a quin *package* pertanyen, són tractats per Java com a elements pertanyents a un mateix *package* anomenat `default` o `unnamed`. L'ús d'un *default package* és molt comú quan es comença a programar en Java o es fan aplicacions temporals o molt petites. No obstant això, no es recomana no crear *packages* per organitzar un programa. De fet, un IDE com Eclipse ens crea un *package* `default` quan creem un projecte Java nou amb la finalitat de tenir organitzat el nostre codi des del primer moment.

Si mirem l'estructura de directoris del nostre programa, per exemple, en el *workspace* del nostre IDE, veurem que dins de la carpeta «src» s'ha creat una estructura de subcarpetes basada en els *packages* declarats. Així doncs, els *packages* pel que fa a organització no deixen de ser directoris amb els quals organitzar el nostre codi. De fet, aquesta és la manera de crear *packages* manualment: crear l'estructura de directoris, posar els elements en el directori que els correspongui i escriure en cada element la línia de codi `package packageName`. Com podem veure, un IDE ens estalvia molt de temps.

Un cop arribats a aquest punt, segurament us deveu preguntar: hi ha alguna convenció per anomenar *packages*? La resposta és sí. Cal respectar el màxim possible la convenció que explicarem a continuació per evitar crear dos *packages* que s'anomenin exactament igual perquè, en cas contrari, no podrem identificar unívocament dos elements amb el mateix nom pertanyents a *packages* diferents.

A l'hora d'anomenar un *package*, hem de tenir sempre present que:

1) El nom *package* s'escriu, tot ell, en minúscules.

2) Si el programa es desenvolupa dins d'una companyia o per a ella, fem servir el nom invertit del domini d'internet d'aquesta companyia per crear un prefix (és a dir, una arrel comuna) a partir del qual penjarà la resta de *packages*.

Per exemple, si desenvolupem un programa a/per a la UOC, el domini de la qual a internet és `uoc.edu`, tots els *packages* que anem creant aniran dins d'`edu.uoc`:

```
edu.uoc.mypackage
edu.uoc.graphics
edu.uoc.assessment
```

3) Si dins d'una companyia (o d'un mateix programa) hi ha un conflicte entre dos o més *packages*, s'ha de solucionar mitjançant convenis interns de la companyia.

Per exemple, imaginem que els programadors de les seus de Barcelona i Madrid de la UOC han creat el *package* `assessment`. Això significaria que tots dos equips tindrien `edu.uoc.assessment`, però amb contingut (és a dir, classes, interfícies, enumeracions, etc.) totalment diferents (podria haver-hi, per exemple, una classe que s'anomenés igual, però la funcionalitat de la qual fos diferent). Aquesta situació generaria conflictes si tots dos *packages* es volen utilitzar/integrar en un mateix programa. Per tant, cal buscar alguna manera de desambiguar. Com es desambigua ho decideix en aquests casos la mateixa companyia. Una possibilitat, en aquest cas, seria afegir la seu: `edu.uoc.barcelona.assessment` i `edu.uoc.madrid.assessment`. Amb aquest petit canvi ara sí que no hi pot haver cap ambigüitat.

4) Hi ha alguns noms de domini d'internet que no són adequats per crear *packages* o, si més no, creen noms conflictius que cal solucionar. Per exemple:

Domini d'internet	Nom de prefix de *package* incorrecte	Nom de prefix de *package* correcte
eimt.int	`int.eimt`	`int_.eimt`
uoc.8edu	`8edu.uoc`	`eightedu.uoc`
uoc-eimt.edu	`edu.uoc-eimt`	`edu.uoc_eimt`

5) Es recomana no fer servir ni les paraules `java` ni `javax` com a primera paraula del nom d'un *package*, ja que l'API de Java utilitza tots dos noms en la declaració dels seus *packages*.

3.5.3. Utilitzar els elements inclosos dins d'un *package*

Vegem com utilitzar un element (classe, interfície o enumeració) inclòs dins d'un *package*. Els elements inclosos en un *package* s'anomenen *membres del paquet*.[12] Hi ha tres formes d'indicar que volem utilitzar un element d'un paquet.

(12) En anglès, *package members*.

1) Importar només l'element que cal utilitzar. Aquesta manera ja l'hem vist a l'apartat «Modificadors d'accés». Recordem que consisteix a fer un `import` de l'element dins del codi de l'element en què el volem fer servir.

```
package edu.uoc.example; //si indiquem el package, ha de ser la primera línia
import java.awt.List; //Indiquem que volem la classe List de java.awt

public class GuiFamily{
       private List members;

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

2) Importar el *package* sencer. Una manera similar a l'anterior és importar tots els elements inclosos dins del *package* (és a dir, els *package members*), no només l'element que volem.

```
package edu.uoc.example; //si indiquem el package, ha de ser la primera línia
import java.awt.*; //Indiquem que volem tot el que estigui dins de
//java.awt. Això inclou List i també Frame, etc.

public class GuiFamily{
       private List members;

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

El codi anterior hauria estat «equivalent» a aquest altre:

```
package edu.uoc.example; //si indiquem el package, ha de ser la primera línia
import java.awt.Frame;
import java.awt.List;

public class GuiFamily{
       private List members;

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

Hem dit que el codi anterior és «equivalent» perquè lògicament no és exactament igual que posar simplement `import java.awt.*`.

Quan posem `import java.awt.Frame` i `import java.List` només importem els elements `Frame` i `List` del *package* `java.awt`, mentre que amb `java.awt.*` importem tots els elements (classes, interfícies i enumeracions) del *package* `java.awt`. Així doncs, amb `java.awt.*` també importaríem, per

exemple, la classe `Label` –que serveix per crear etiquetes (trossos de text) en una interfície gràfica–, a més d'altres classes i interfícies (per exemple, `Shape`, `ItemSelectable`, etc.) del *package* `java.awt`.

Quina de les dues formes és millor? Sens dubte, la primera: `import java.awt.Frame` i `import java.List`. És a dir, sempre és millor importar allò que realment fem servir. El primer motiu és que d'aquesta manera queda clar, per a nosaltres i altres programadors, què utilitzem i què no d'un *package*. A més, si ho importem tot pot donar-se alguna situació no volguda de manera accidental com, per exemple, quan dos *packages* tenen elements que s'anomenen igual. Imaginem que tenim dos *packages*: `package1` i `package2`, en què en el primer (`package1`) hi ha dues classes `A` i `B`, mentre que en el `package2` tenim `B`, `C` i `D`. Del `package1` fem servir la classe `A`, mentre que del `package2` utilitzem la classe `B`. Si fem:

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

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

A quina classe `B` fem referència, a la del `package1` o a la del `package2`? El compilador ens avisarà amb un missatge. Per contra, si escrivim el codi que es mostra a continuació, no hi ha dubte de quina classe `B` volem dir:

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

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

Utilitzar el *fully qualified name* de l'element

Una altra manera d'indicar quin element volem utilitzar és mitjançant el nom complet de l'element. És a dir, incloure el nom del paquet més el de l'element sense necessitat d'importar res.

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

Encara que el codi anterior és correcte, no sol ser habitual ni pràctic escriure tota l'estona el *fully qualified name* dels elements. És molt més còmode importar l'element, com en els apartats anteriors. No obstant això, aquesta forma d'identificar unívocament un element pot ser útil en casos com el següent:

```
import package1.B; //podríem posar import package1.*;
import package2.B; //podríem posar import package2.*;

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

De quina classe `B` és el `fieldB1`? Del `package1` o del `package2`? I el `fieldB2`? La manera de solucionar aquesta ambigüitat és mitjançant el *fully qualified name*.

```
import package1.B; //podríem posar import package1.*;
import package2.B; //podríem posar import package2.*;

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

Mitjançant el *fully qualified name* com en el codi anterior, l'ambigüitat desapareix per al compilador i no ens dirà res.

3.5.4. *Packages* importats automàticament per Java

El compilador de Java importa de manera automàtica i implícita dins de qualsevol element el mateix *package* al qual pertany i el *package* `java.lang`. És a dir, no cal fer un `import` ni del mateix *package* ni de `java.lang`.

1) Importació implícita del mateix *package*. Quan utilitzem una classe `A` del *package* `mypackage` dins d'una classe `B` del mateix *package*, no cal importar el *package*, ja que ambdues classes estan en el mateix *package* i «es veuen» l'una a l'altra.

```
package mypackage;
public class A{
        //Code
}
```

```
package mypackage;

public class B{
        public B(){
                //Podem utilitzar A perquè A i B
                //pertanyen a mypackage
                A objectA = new A();
        }
}
```

2) Importació implícita del *package* `java.lang`. El compilador de Java inclou automàticament aquest *package*. De fet, ja hem vist algunes classes que estan dins d'aquest *package* i fins ara no havíem fet cap `import`. Per exemple, la classe `String` és una de les classes que estan incloses en `java.lang`. De la mateixa manera, la classe `System` amb la qual mostrem text per la consola d'ordres també està dins d'aquest *package*.

3.6. Modificador `static`

Com ja heu llegit en els apunts teòrics, hi ha la paraula clau `static` que pot ser aplicada a un atribut, un mètode o una classe imbricada.

3.6.1. Atribut estàtic

Si apliquem el modificador `static` a un atribut, aquest és compartit per tots els objectes pertanyents a una mateixa classe. Així doncs, deixa de ser un atribut de la instància per ser un atribut de la classe. És per això que els atributs amb el modificador `static` són anomenats *atributs estàtics*[13] o *atributs de la classe*. Els atributs estàtics ocupen un sol espai de memòria per a tots els objectes, és a dir, no hi ha memòria reservada per a aquest atribut per cada objecte que s'instancia.

(13) En anglès, *static fields*.

Qualsevol objecte pot accedir i modificar el valor de l'atribut estàtic, però també es pot accedir a aquest atribut des d'objectes d'altres classes, si el seu modificador d'accés ho permet, sense necessitat de crear un objecte de la classe a la qual pertany l'atribut estàtic. Així doncs, si tenim:

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

Des d'una altra classe podem fer:

```
public class B{
        public B(){
                A.num = 8; //es pot accedir als atributs estàtics
//referenciant la classe a la qual pertanyen -> forma recomanada
                A objectA = new A();
                objectA.num = 9; //també es pot accedir als atributs estàtics mitjançant
//un objecte de la classe
                System.out.println(A.num); //9
        }
}
```

Es recomana accedir als atributs estàtics per mitjà de la classe, no d'un objecte, per facilitar la llegibilitat del codi.

3.6.2. Bloc d'inicialització estàtic

Per als casos en els quals volem inicialitzar atributs estàtics, aquests poden ser inicialitzats en la declaració del mateix atribut o en un bloc d'inicialització estàtic (similar al bloc d'inicialització).

Vegeu també

El bloc d'inicialització s'ha tractat en l'apartat «Constructor» d'aquesta mateixa guia.

Imagineu que tenim un atribut `id` que serveix per assignar un identificador a cada objecte `Person` creat i que aquest `id` ha de ser únic. Podríem tenir el codi següent:

```
public class Person{
	private static int id; //Podríem haver posat id = 0 i estalviar-nos el bloc
//d'inicialització estàtic
	private String name, surname;
	private int age;
	private int height;
	private float weight;

	//Això és un bloc d'inicialització estàtic. Només s'ha de posar static{}
	static{
		id = 0; //També podem cridar des d'aquí mètodes static
	}

	{ //Bloc d'inicialització d'instància
		setAge(36);
	}

	//Constructor per defecte
	public Person(){
		this("David","García",172,66);
	}

	//Constructor amb arguments 1
	public Person(String name, String surname){
		this(name,surname,172,66);
	}

	//Constructor amb arguments 2
	public Person(String name, String surname, int height, float weight){
		setName(name);
		setSurname(surname);
		setHeight(height);
		setWeight(weight);
		id++;
	}
}
```

És important entendre que el bloc d'inicialització estàtic, a diferència del d'instància, només s'executa una vegada per programa. Quan es fa aquesta única crida? Doncs, o bé quan es crea per primera vegada un objecte de la classe, o bé quan s'accedeix per primera vegada a un membre (és a dir, atribut o mètode) estàtic de la classe. En tots dos casos, el bloc d'inicialització estàtic es crida abans que el constructor o abans d'accedir al membre.

3.6.3. Mètode estàtic

Java permet utilitzar la paraula clau `static` amb mètodes. De la mateixa manera que els atributs, els mètodes estàtics pertanyen a la classe, no a la instància. Així doncs, aquests poden ser cridats mitjançant un objecte o el nom de la classe (aquesta forma és la recomanada).

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

Des d'una altra classe podem fer:

```
public class B{
        public B(){
                A.doSomething(); ////es pot accedir als mètodes estàtics
//referenciant la classe a la qual pertanyen -> forma recomanada
                A objectA = new A();
                objectA.doSomething(); //també es pot accedir als mètodes estàtics
//mitjançant un objecte de la classe
        }
}
```

Us heu fixat que el mètode especial `main` és estàtic? Aquest es crida fent `java NomClasse` sense instanciar cap objecte de la classe.

És important tenir present que hi ha una limitació en fer servir aquest tipus de mètodes.

> Des d'un mètode estàtic no podem accedir a atributs ni mètodes d'instància directament (necessitem instanciar un objecte).

```
public class A{
        private int num = 0;

        public static void doSomething(){
                num = 8; //ERROR: No podem accedir a un atribut d'instància
//des d'un mètode estàtic (és a dir, un mètode de la classe, no de la instància).
                //TODO
        }
}
```

Així mateix, un mètode estàtic no pot utilitzar la paraula clau `this` perquè no hi ha cap objecte al qual referir-se.

3.6.4. Classe estàtica

En Java no podem declarar una classe de nivell superior amb el modificador `static`. Només les classes imbricades poden ser `static`.

Vegeu també

Per saber-ne més sobre les classes imbricades (i les `static`), recomanem llegir l'apartat «Classes imbricades» d'aquesta guia.

3.7. Modificador `abstract`

En Java podem utilitzar la paraula clau `abstract` tant per a classes com per a mètodes, però no per a atributs.

3.7.1. Classe abstracta

En Java, quan una classe és declarada com a abstracta significa que aquesta classe no pot ser instanciada. És a dir, no podem crear objectes d'aquesta classe.

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

```
public class App{
	Vehicle car = new Vehicle(); //ERROR: Vehicle és una
//classe abstracta
}
```

3.7.2. Mètode abstracte

Quan un mètode és declarat abstracte diem que la seva codificació la deleguem en subclasses que heretin la classe en la qual es troba el mètode abstracte. Així mateix, **quan declarem un mètode abstracte, obligatòriament la seva classe també ha de ser declarada abstracta**.

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

En l'exemple anterior, el mètode `run` és abstracte. Per tant, no té codi (ni tan sols hem escrit les claus que limiten el cos del mètode). Serà en una subclasse (o subclasse de la subclasse) on li proporcionarem un codi.

Subclasse

Veurem el concepte *subclasse* tant en els apunts teòrics com més endavant en aquesta guia.

```
public class Car extends Vehicle{ //Ara com ara no feu cas d'"extends Vehicle"
	public void run(){
		System.out.println("I'm a car… brrr!!!");
	}
	//TODO
}
```

3.8. Modificador `final`

En Java hi ha la paraula clau `final` que pot ser aplicada tant a una classe com a un mètode o un atribut.

3.8.1. Classe `final`

Dels apunts teòrics sabem que una classe `final` és aquella de la qual no es pot heretar.

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

La paraula clau `final` pot escriure's abans o després del modificador d'accés. El codi següent és equivalent a l'anterior.

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

Així doncs, fer el següent provocaria un error en temps de compilació.

```
public class Car extends Vehicle{ //Error de compilació, Vehicle és final
       //TODO
}
```

3.8.2. Mètode `final`

Com ja sabem dels apunts teòrics, un mètode declarat com a `final` implica que no pot ser sobreescrit en una subclasse. Si ho intentem, obtindrem un error de compilació.

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

Considerant la declaració anterior, seria incorrecte fer el següent:

```
public class Car extends Vehicle{
       public void run(){
         //TODO: no podem sobreescriure run perquè és final en la classe Vehicle
       }
}
```

3.8.3. Atribut `final`

Pels apunts teòrics sabem que un atribut declarat com a `final` actua com a constant. Per tant, només li podem assignar un valor una vegada en tot el programa. Així doncs, una cop assignat un valor, ja no podem modificar-lo.

```
public class Vehicle{
       private final int MAX_SPEED = 250;//Utilitzem la convenció de les constants
}
```

Si intentem fer el següent, obtindrem un error de compilació.

```
public class Vehicle{
       private final int MAX_SPEED = 250; //Utilitzem la convenció de les constants

       public Vehicle(){
              MAX_SPEED = 200; //Error de compilació
       }
}
```

És molt habitual combinar els modificadors `static` i `final` per crear constants:

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

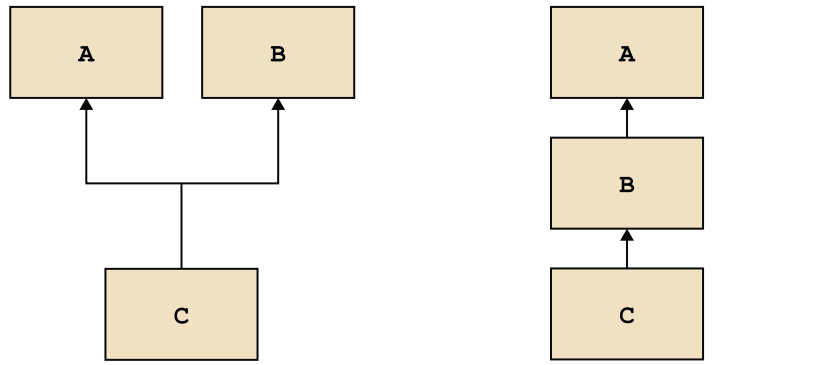
3.9. Herència

3.9.1. Concepte i sintaxi

L'herència és un mecanisme essencial en la programació orientada a objectes. Consisteix a permetre que una classe –anomenada subclasse– es defineixi a partir de la codificació d'una altra ja existent –anomenada superclasse–. Sembla una cosa senzilla, però és molt potent. Per això, cal dominar-ho, ja que, entre altres coses, facilita la reutilització de codi.

En Java només hi ha l'herència simple: una classe només pot heretar directament d'una altra classe. És a dir, la classe `C` no pot heretar directament d'`A` i `B`, alhora. No obstant això, existeix la transitivitat. És a dir, si una classe `B` hereta d'una classe `A`, i una classe `C` hereta de `B`, llavors la classe `C` hereta de `B` i hereta també tot allò que hagi heretat `B` de `A`.

Figura 22



Herència múltiple, no permesa Transitivitat en l'herència, permesa

La sintaxi per indicar que una classe hereta d'una altra és la següent:

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

Amb el codi anterior diríem que la classe `B` hereta de la classe `A`. O, dit d'una altra manera, la classe `B` és una subclasse de la classe `A` i, al seu torn, la classe `A` és una superclasse de la classe `B`.

3.9.2. La classe `Object`

En aquest punt, cal dir que en Java totes les classes i els *arrays* hereten implícitament d'una superclasse anomenada `Object`, definida pel llenguatge Java mateix. És a dir, tant si ho volem com si no, totes les classes i *arrays* hereten d'`Object`.

Enllaç recomanat

En l'enllaç següent podeu veure com es defineix la classe `Object`: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html.

Aquesta classe defineix un conjunt de mètodes, entre els quals destaquen: `clone`, `equals` i `toString`.

3.9.3. Què hereta una subclasse?

Constructors i herència

Els constructors no són considerats membres d'una classe, perquè, entre altres coses, no s'hereten.

És clau saber què hereta la classe `B` de la classe `A`. La resposta és: *tots els membres*. Tots els membres? Sí, una altra qüestió és si la classe `B` té accés a tots els membres heretats de `A` com si aquests s'haguessin declarat a la mateixa classe `B`. Sobre això últim, la resposta és que una subclasse només té accés directe als membres declarats com a `public` o `protected` en la superclasse. En el cas dels membres `package-private` només té accés directe si la superclasse i la subclasse pertanyen al mateix *package*.

Hi ha molta documentació (inclosa la documentació oficial) que barreja «el que hereta la subclasse» amb «el que es pot accedir des de la subclasse» i, per això, és molt habitual llegir que en Java no s'hereten els membres privats. Per aquest motiu, en aquesta guia pensem que és més senzill d'entendre com funciona l'herència en Java si diem que s'hereta tot de la superclasse, però que la subclasse no pot accedir directament als membres `private` de la classe `A` ni als membres `package-private` si superclasse i subclasse pertanyen a *packages* diferents.

Vegem i analitzem un exemple pràctic per entendre com afecta el mecanisme d'herència a una subclasse.

Animal.java

```
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;
    }

    private 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());
    }
}
```

Dog.java

```
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 mètode super ens permet cridar
                                            //un constructor de la superclasse
                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 //Sobreescrivim el mètode speak d'Animal
        public void speak(){
                super.speak(); //amb super cridem el mètode speak d'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 codi anterior requereix una anàlisi en profunditat:

1) La classe `Animal` és la que menys explicacions requereix. Hem assignat un modificador diferent als atributs per exemplificar-ne el comportament durant l'herència. Cal destacar l'ús del mètode especial `this` en el primer i el segon constructor per cridar el tercer constructor i, d'aquesta manera, minimitzar el codi que cal escriure.

2) La classe `Dog` hereta de la classe `Animal`. És a dir, `Dog` és una subclasse d'`Animal` i, per tant, `Animal` és una superclasse de la classe `Dog`.

3) Les subclasses no hereten els constructors definits en la seva superclasse, ja que, com ja sabem, els constructors (i els destructors) no són considerats membres d'una classe.

4) En el constructor de la classe `Dog`, cal destacar l'ús del mètode `super`, mètode especial que s'utilitza en la subclasse per cridar un dels constructors de la superclasse. La signatura del mètode `super` ha de coincidir amb alguna de les definides en la superclasse. Si no s'explicita una crida al mètode `super`, el

compilador crida automàticament `super()`, és a dir, crida el constructor per defecte de la superclasse. La crida al constructor de la superclasse ha de ser la primera instrucció del constructor d'una subclasse.

5) Des de dins de la subclasse `Dog`, a més de tenir accés directe als mètodes `setOwner`, `getOwner`, `setBreed`, `getBreed`, `speak` i `toString`, també es té accés directe als mètodes de la superclasse `Animal` declarats com a `public` o `protected`. És a dir: `getName`, `setName`, `setAge`, `getAge` i `speak`. Fixeu-vos que no té accés directe als mètodes `getNumberOfLegs` i `setNumberOfLegs`, ja que han estat declarats `private` en la classe `Animal`. Així doncs, en el codi de la classe `Dog` no podem cridar directament els mètodes `getNumberOfLegs` i `setNumberOfLegs`. En canvi, dins de la classe `Dog`, podrem accedir directament a l'atribut `vegetarian` si `Animal` i `Dog` estan en el mateix *package*.

6) En les subclasses podem canviar el comportament dels mètodes definits en la superclasse. Aquest procés de redefinició s'anomena **sobreescriptura** (*overriding*). Això és just el que fa la classe `Dog` amb el mètode `speak`. Si ens hi fixem, hem tornat a codificar el mètode `speak` en la classe `Dog` amb exactament la mateixa signatura (si la canviem, sobrecarreguem). D'aquesta manera, li assignem un comportament diferent que el de la superclasse `Animal`. Si no l'haguéssim sobreescrit, estaria present en la classe `Dog`, però amb el comportament definit en la superclasse `Animal`.

7) Per avisar el compilador (i nosaltres mateixos) que hem sobreescrit un mètode de la superclasse, és una bona pràctica fer servir l'anotació `@Override` just abans de la signatura del mètode. No n'és obligatori l'ús, però sí molt recomanable, ja que el compilador farà comprovacions extres per avisar-nos de si hem comès algun error a l'hora de fer la sobreescriptura.

8) En la versió sobreescrita del mètode `speak`, és a dir, el mètode `speak` de la classe `Dog`, veiem que cridem el mètode `speak` de la superclasse. Per fer això, que no és obligatori, hem de fer servir l'objecte especial `super`. D'aquesta manera, el compilador sap que cridem el mètode `speak` de la superclasse i no el de la mateixa classe. Així doncs, el mètode `speak` d'un objecte de tipus `Dog`, a més d'escriure `"My name is "` i el nom del gos, també escriurà `"Woof!!"`.

9) Fixem-nos que el mètode `speak` en la subclasse `Dog` té un modificador d'accés menys restrictiu que en la superclasse `Animal`. Ha passat d'ésser `protected` a `public`. Això és correcte, ja que en la subclasse podem assignar a un mètode un modificador menys restrictiu que el que té en la superclasse. A l'inrevés, no és correcte. És a dir, a un mètode sobreescrit no li podem assignar un modificador d'accés més restrictiu en la subclasse que en la superclasse. Així doncs, si en `Dog` haguéssim declarat `speak` com a `private`, el compilador ens llançaria un error.

10) L'últim mètode que trobem és `toString`. Com veiem, aquest mètode està sobreescrit, però de quina superclasse l'ha heretat `Dog` si en `Animal` no hi és? Doncs bé, com hem dit prèviament, totes les classes en Java hereten implícitament de la superclasse `Object` definida pel llenguatge Java mateix. Aquesta classe `Object` té definits, entre altres mètodes, el mètode `toString` (que ja havíem vist anteriorment en aquesta guia). El mètode `toString` s'utilitza per donar informació sobre l'objecte. El seu codi per defecte, és a dir, allò que codifica la classe `Object`, és retornar un `String` amb una representació interna de l'objecte i informació de la referència de l'objecte (és a dir, la posició de memòria que ocupa en el *heap*).

11) Si ens fixem en el nou codi que hem assignat al mètode `toString` en la classe `Dog`, veiem que no cridem el mètode `getNumberOfLegs`, ja que aquest és `private` en la superclasse `Animal` i, per tant, des de la subclasse `Dog` no hi tenim accés. No obstant això, com que l'atribut `numberOfLegs` està definit com a `protected` en la superclasse `Animal`, sí que hi podem accedir directament dins de la subclasse `Dog`.

12) Un codi alternatiu per al mètode `toString` amb el nombre mínim de crides a mètodes seria:

```
return "I am the dog "+getName()+" ("+breed+"). I am "+age+" and I have "+numberOfLegs+" legs.";
```

Veiem que no podem accedir a l'atribut `name` directament des de `Dog` perquè aquest atribut ha estat declarat `private` en la superclasse. Així doncs, l'única manera d'accedir-hi és fent servir indirectament un mètode declarat en la superclasse com a `public` o `protected` (o `package-private` si `Animal` i `Dog` estan en el mateix *package*) que ens doni accés a aquest atribut. En aquest cas, accedim a `name` mitjançant `getName`, que és `public`.

13) Un cop arribats a aquest punt hem de preguntar-nos què succeeix en la subclasse `Dog` amb l'atribut `vegetarian` de la seva superclasse `Animal`. Aquest ha estat declarat sense modificador d'accés, com a `package-private`. Doncs bé, si la classe `Animal` i la seva subclasse `Dog` pertanyen al mateix *package*, des de dins de la classe `Dog` es pot accedir a l'atribut `vegetarian` com si hagués estat declarat dins de `Dog`. En cas contrari, es comporta com si fos `private`, és a dir, no es podria accedir a `vegetarian` directament des de dins del codi de `Dog`.

Ara vegem un altre codi Java que utilitza les classes `Animal` i `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 pertany al mateix package que Animal,
//aleshores aquesta crida és correcta. Si estan en packages diferents, no
//pot ser cridat des de la classe Example perquè, entre altres coses, no és subclasse
//d'Animal

              dog1.speak(); //Imprimeix "My Name is Lassie" i en la línia següent
                            //imprimeix "Woof!!".

              dog1.age = 10; //és correcte, perquè s'hereta d'Animal com a public

              animal1.age = 5; //és correcte, perquè és public

              animal1.numberOfLegs = 5; //incorrecte si Example i Animal pertanyen
//a packages diferents. Crida correcta si pertanyen al mateix package.

              animal1.setNumberOfLegs(5); //incorrecte, perquè és private

              dog1.numberOfLegs = 5; //incorrecte si Example i Animal (no Dog) pertanyen
//a packages diferents. Crida correcta si pertanyen al mateix package.

              dog1.setNumberOfLegs(5); //incorrecte, perquè Dog no té accés a
//aquest mètode en haver estat declarat private en la superclasse Animal

              dog1.setName("Pelusa"); //és correcte, perquè s'hereta d'Animal
                                      //com a public

              dog1.setBreed("Chihuahua"); //és correcte, perquè és public en Dog

              animal1.setBreed("Chihuahua"); //incorrecte, perquè setBreed és de la
//subclasse Dog, no pertany a la superclasse Animal. Animal no hereta res de Dog.

              animal1.vegetarian = true; //el resultat dependrà: si la classe
//Example pertany al mateix package que Animal, aquesta sentència seria
//correcta. En cas que pertanyi a packages diferents, seria incorrecta.
       }
}
```

3.9.4. El modificador `abstract` i l'herència

Com ja hem vist amb anterioritat, en Java hi ha el modificador `abstract`, el qual pot ser aplicat tant a classes com a mètodes. L'única cosa que afecta en el cas de l'herència és que en la subclasse hem de codificar tots els mètodes declarats com a `abstract` en la superclasse. Si no es codifiquen, han de ser `abstract` en la subclasse perquè una subclasse de la subclasse els codifiqui. A més, la subclasse també haurà de ser una classe declarada com a `abstract`.

Un dubte que sorgeix sovint amb les classes abstractes és si té sentit declarar-los un constructor si una classe abstracta, per definició, no pot ser instanciada. Doncs bé, una cop vist com es relaciona una subclasse amb la seva superclasse i que en el constructor de la subclasse s'ha de cridar algun constructor de la superclasse, la resposta és clara: sí, té sentit declarar un constructor en una classe abstracta si aquesta serà superclasse d'alguna classe. El constructor de la superclasse `abstract` no ha de ser forçosament `public`, però pot ésser `protected` perquè així hi tinguin accés directe els constructors de la subclasse mitjançant el mètode especial `super`.

3.9.5. El modificador `static` i l'herència

Amb aquest modificador, cal diferenciar tres situacions:

1) Mètode no estàtic en la superclasse. Si en la subclasse sobreescrivim un mètode no `static` de la superclasse i, a més, el definim `static` en la subclasse, el compilador ens donarà un error.

2) Mètode estàtic en la superclasse. Si en la subclasse el sobreescrivim, però sense declarar-lo `static`, el compilador ens donarà un error.

3) Mètode estàtic en la superclasse. Si en la subclasse el sobreescrivim mantenint-ne la qualitat de `static`, llavors el mètode de la superclasse queda ocult. De fet, allò correcte és dir que un mètode estàtic en la superclasse no pot ser sobreescrit en una subclasse, sinó que pot ser ocultat. Així doncs, en la subclasse, en veritat, hi ha els dos mètodes `static` i es cridarà l'un o l'altre en funció de si aquest mètode és invocat mitjançant la superclasse o la subclasse.

El que hem explicat fins ara per als mètodes també succeeix amb els atributs estàtics. És a dir, en la subclasse podem declarar un atribut anomenat exactament igual que un de la superclasse. En aquest cas, el que fem és ocultar aquest atribut i podrem cridar l'un o l'altre en funció de si el cridem mitjançant la superclasse o la subclasse.

3.9.6. El modificador `final` i l'herència

Com ja hem vist, aquest modificador té una afectació directa amb l'herència. Recordem que si una classe és `final`, no pot ser heretada; per tant, no podrem definir subclasses a partir de la classe `final`.

Quant als mètodes declarats com a `final`, no podran ser sobreescrits en les subclasses. És a dir, el comportament definit en la superclasse no podrà ser modificat en cap de les seves subclasses.

3.9.7. Ocultació d'atributs

En una subclasse podem declarar un atribut amb el mateix nom que un atribut declarat en la superclasse, encara que els seus tipus siguin diferents. Si des de la subclasse es pot accedir directament a l'atribut de la superclasse, en declarar un atribut en la subclasse amb el mateix nom el que faríem és ocultar l'atribut heretat de la superclasse. Així doncs, si volem accedir a l'atribut de la superclasse haurem de fer servir l'objecte `super`.

```
public class A{
      protected int value;
}
```

```
public class B extends A{
       protected int value;

       public B(){
              super();
              value = 10; //classe B
              super.value = 15; //classe A
       }
}
```

No es recomana ocultar atributs d'una superclasse en les seves subclasses.

3.10. Interfícies

3.10.1. Concepte i sintaxi

Com ja sabem, les interfícies són contractes que indiquen quins mètodes han de ser codificats en les classes que les implementen. Les interfícies no proporcionen el codi dels mètodes, només les seves signatures.

Per definir una interfície en Java, hem d'utilitzar la sintaxi següent:

```
public interface Movable{
       void goAhead();
       void reverse();
       void left();
       void right();
       double getSpeed();
       void setSpeed(double speed);
}
```

Com podem veure, hem assignat el modificador d'accés `public` a la interfície `Movable`. Això vol dir que qualsevol classe o interfície de qualsevol *package* del nostre programa pot fer servir la interfície `Movable`. En cas de no posar el modificador `public`, la visibilitat de la interfície és `package-private` i només les classes i interfícies que pertanyen al mateix *package* que la interfície `Movable` poden utilitzar-la. Altres modificadors no són possibles a alt nivell. No obstant això, si la interfície està imbricada dins d'una altra classe, llavors sí que pot ser, per exemple, `private`.

En una interfície no cal indicar el modificador d'accés dels mètodes. Si no s'escriu explícitament, s'entén que és `public`. De fet, els mètodes en una interfície Java són sempre `public` i, per defecte, `abstract` (veurem que els mètodes també poden ser `static` o `default`). Però, no cal escriure cap dels dos modificadors perquè se sobreentén que si no s'indica res, es declaren com a `public` i `abstract`.

3.10.2. Ús d'una interfície

Qualsevol classe pot implementar una interfície mitjançant la paraula reservada (*keyword*) `implements`. Una classe pot implementar múltiples interfícies alhora. Per a cadascuna d'elles, haurà de proporcionar el codi dels mètodes definits en cadascuna de les interfícies que implementa. En cas contrari, el compilador donarà un error.

```
public class Animal implements Movable, Comparable{
    //TODO: codificar els mètodes definits a les interfícies Movable i Comparable
}
```

Una interfície no pot implementar una altra interfície, ja que no li'n podria proporcionar el codi, però sí que pot heretar una o més interfícies.

```
public interface Movable extends Comparable{
    //TODO
}
```

3.10.3. Quins elements pot contenir una interfície?

Aquesta és una pregunta molt habitual. També cal dir que la resposta depèn de la versió de Java de la qual parlem, ja que el concepte d'interfície en Java ha anat evolucionant amb el pas dels anys. Això sí, allò que s'ha introduït en una versió s'ha mantingut en les següents.

Des de Java ES 7, les interfícies només permetien constants (atributs que implícitament són declarats `public static final`) i mètodes abstractes.

```
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, les interfícies van ser capaces d'incloure també mètodes estàtics i mètodes per defecte.[14] La inclusió d'aquests dos tipus de mètodes ens permet escriure codi dins d'un mètode.

(14) En anglès, *default methods*.

```
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 codi anterior, hem afegit un mètode `static` anomenat `isFast` i hem convertit el mètode `up` en un mètode `default` al qual hem proporcionat una codificació (el fet de convertir-lo en `default` ens obliga a codificar-lo). Així doncs, qualsevol classe que implementi la interfície `Movable` ja tindrà una codificació per defecte (de base) per al mètode `up` i no serà necessari, si no vol, que sobreescrigui aquest mètode. El mètode `isFast`, que és `static`, no pot ser sobreescrit en les classes que implementin la interfície `Movable`.

En Java 9 es va introduir la possibilitat de definir mètodes privats i mètodes `static` privats. Amb ells es facilita la reutilització de codi dins de les interfícies i la seva encapsulació. Així doncs, els mètodes `private` són útils si, per exemple, dos mètodes `default` comparteixen un tros de codi. Gràcies als mètodes privats podríem compartir codi entre mètodes sense exposar-lo fora de la interfície. Els mètodes privats han de complir les regles següents:

- Els mètodes privats d'una interfície no poden ser abstractes. O, dit d'una altra manera, els mètodes privats d'una interfície han de tenir codi en el seu cos.

- Els mètodes privats d'una interfície només poden utilitzar-se dins de la interfície en la qual han estat definits.

- Els mètodes privats d'una interfície poden cridar mètodes `default` i mètodes `static` (tant si són `public` com `private`).

- Els mètodes privats `static` d'una interfície poden utilitzar-se tant en mètodes estàtics com no estàtics de la interfície.

- Els mètodes privats d'una interfície que no siguin `static` no poden utilitzar-se dins de mètodes privats `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){
              return speed>120;
        }
        private void log(String msg){
               System.out.println(msg);
        }
}
```

En el codi anterior, fem servir un mètode privat anomenat `log` per fer un seguiment de les crides que es fan als mètodes de la interfície. Si volguéssim cridar el mètode `log` des d'`isFast`, `log` hauria d'haver-se declarat `static` (tant si és `public` com `private`).

En resum:

Element	Des de quan es pot utilitzar
Atributs `public static final`	Java 7
Mètodes `public abstract`	Java 7
Mètodes `public default`	Java 8
Mètodes `public static`	Java 8
Mètodes `private`	Java 9
Mètodes `private static`	Java 9

3.11. Polimorfisme

Com hem comentat en el mòdul teòric «Herència (relació entre classes)», el polimorfisme és un dels pilars del paradigma de la programació orientada a objectes. Per aquest motiu, és molt important dominar-lo.

Ja sabem que el **polimorfisme** és la capacitat que un objecte declarat com a tipus superclasse pugui comportar-se com un objecte de qualsevol de les seves subclasses.

A continuació, vegem-ho amb codi (suposeu que totes les classes pertanyen al mateix `package`):

```
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;
    }
}
```

```
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!!");
    }
}
```

Si ara tinguéssim el codi següent en una altra classe, com es comportaria el programa?

```
public class Check{
       public static void main(String[] args){
              Animal animalDog = new Dog("Eddie"); //línia 1
              animalDog.greeting();//línia 2
              Dog dog1 = (Dog) animalDog; //línia 3
              dog1.greeting();//línia 4
              Dog dog2 = (Dog) animalDog; //línia 5
              Cat cat1 = (Cat) animalDog; //línia 6
              Cat cat2 = animalDog; //línia 7
              animalDog = new Cat(); //línia 8
              animalDog.greeting(); //línia 9
       }
}
```

- La línia 1 seria correcta. Així, el tipus estàtic de la variable `animalDog` és `Animal`, que és superclasse del tipus dinàmic que se li assigna: objecte de la subclasse `Dog`.

- La línia 2 imprimirà per pantalla el text propi de la classe `Dog`, perquè `animalDog` es comporta com un `Dog`, en ser el seu tipus dinàmic `Dog`. Així doncs, imprimirà `"Woof!!"`.

- En la línia 3 es fa un càsting d'`animalDog` a `Dog`. Ara, la variable `dog1` té l'objecte `Dog` que hi ha en `animalDog`. Tant `animalDog` com `dog1` apunten la mateixa referència, el mateix objecte.

- La línia 4 imprimeix `"Woof!!"`, ja que tant el tipus estàtic com dinàmic de la variable `dog1` és `Dog`.

- Després d'executar la línia 5, tant `dog1` com `dog2` i `animalDog` apunten el mateix objecte, són la mateixa referència.

- La línia 6 és errònia. No obstant això, l'error apareix en temps d'execució, no de compilació. L'error es deu al fet que no podem fer un càsting d'un objecte `Dog` a un objecte `Cat`, o el que és el mateix, no podem assignar un objecte `Dog` a una referència de tipus `Cat`.

- La línia 7 és errònia, però, a diferència de la línia anterior, l'error es produeix en temps de compilació. Així doncs, hem d'eliminar-la per poder executar el programa. Per què és errònia la línia 7? Perquè el tipus estàtic d'`animalDog` és `Animal` i no pot ser assignat a una variable `Cat`.

- La línia 8 és el cas contrari de la línia anterior. Per tant, és correcta, ja que `Animal` és superclasse de `Cat`. Justament això és l'exemple més senzill de polimorfisme.

- L'última línia, la 9, imprimiria per pantalla `"Meow!!"`, ja que el tipus dinàmic de la variable `animalDog` és `Cat`.

Una de les utilitats del polimorfisme és simplificar el codi. Això es veu clar en l'exemple següent:

```
Animal [] animals = new Animal[2];
animals[0] = new Dog("Bobby");
animals[1] = new Cat();

for(var animal : animals){
    animal.greeting();
}
```

El codi anterior executaria en la primera iteració de l'*enhanced for* el mètode `greeting` de `Dog`, ja que la primera casella de l'*array* `animals` és un objecte `Dog`. És a dir, la casella `0` d'`animals` té com a tipus estàtic `Animal`, però el seu tipus dinàmic és `Dog`. En la segona iteració, per la mateixa raó, s'executaria el mètode `greeting` de `Cat`. Així doncs, en comptes de tenir dos *arrays* (o dues variables), un de tipus `Dog` i un altre `Cat`, amb una única variable (`animals`) podem fer totes les operacions segons si el seu tipus és dinàmic en cada cas. És a dir, `animals` adopta moltes formes (polimorfisme) dins del mateix programa.

El polimorfisme també és interessant quan fem servir interfícies; això es veu d'una manera molt clara en Java quan utilitzem, per exemple, interfícies que modelen col·leccions. Aquest és el cas de la interfície `List` de l'API de Java.

```
List<Person> lista = new ArrayList<Person>(); //new LinkedList();
public List<Person> getPeople(){
        return lista;
}
```

Fixem-nos en el *getter* `getPeople`. Independentment de com inicialitzem l'atribut `llista`, com a objecte de la classe `ArrayList` o de `LinkedList`, el mètode `getPeople` serà el mateix en tots dos casos, ja que retornem `List<Person>`, generalitzant/abstraient el mètode. Evidentment, els dos codis següents serien correctes:

```
ArrayList<Person> lista = new ArrayList<Person>();

public ArrayList<Person> getPeople(){
        return lista;
}
```

```
LinkedList<Person> lista = new LinkedList<Person>();

public LinkedList<Person> getPeople(){
        return lista;
}
```

No obstant això, els dos codis anteriors tindrien un manteniment pitjor a la llarga, ja que en canviar la inicialització/instanciació, hauríem de canviar la signatura del mètode `getPeople` per adaptar-lo a la nova classe i, de la mateixa manera, tot el codi en què no generalitzem amb l'ús de `List<Person>`.

Finalment, el polimorfisme també ens permet relacionar elements que no estan relacionats per una jerarquia d'herència, sinó per una jerarquia d'interfícies (és a dir, de capacitats). Dit d'una altra manera, gràcies al polimorfisme podrem relacionar elements que *a priori* no estan relacionats. Per exemple, si les classes `Car` i `Fish` implementen la interfície `Movable`, podríem fer:

```
Movable [] elements = new Movable[2];
elements[0] = new Car();
elements[1] = new Fish();

for(var element : elements){
    element.up(); //el mètode up està declarat a la interfície Movable i, per
//això, tant la classe Car com la classe Fish li han hagut de donar un codi.
}
```

3.12. Excepcions

3.12.1. Concepte

Per entendre què és una excepció, imaginem que un programador vol llegir dades d'un fitxer de text en el seu programa. Allò habitual en llenguatges que no inclouen el maneig d'excepcions és deixar segons el parer del programador el control de les excepcions (casos anòmals o no desitjats). Vegem una possible solució adoptada per un programador (sense entrar en els detalls de codi, l'important és l'estructura):

```
if(fitxer existeix){
        //obrim fitxer
        if(fitxer obert correctament){
                while(existeixen línies de text){
                        //llegim una línia de text
                        if(no hi ha hagut problema amb la lectura){
                                //Fem alguna cosa amb la informació llegida
                        }else{
                                //ERROR: Missatge que hi ha hagut un problema en llegir
                        }
                }
                //tanquem el fitxer
                if(fitxer mal tancat){
                        //ERROR: Missatge que el fitxer no s'ha tancat correctament.
                }
        }else{
                //ERROR: Missatge que el fitxer no s'ha obert correctament.
        }
}else{
        //ERROR: Missatge que el fitxer no existeix.
}
```

En el codi anterior, si estigués dins d'un mètode, en comptes d'escriure missatges d'error, podríem haver optat per retornar un codi d'error i, des d'on es va invocar, escriure un missatge o un altre depenent d'aquest codi.

Com podem veure en el codi anterior, la lògica del programa en si i la gestió de les excepcions està barrejada. Això fa que el codi sigui més difícil de llegir i entendre. És per això que Java inclou la gestió d'excepcions i treu responsabilitat als programadors. Vegem el codi anterior en la seva versió Java:

```
try{
        //obrim fitxer
        while(existeixen línies de text){
           //llegim una línia de text
           //Fem alguna cosa amb la informació llegida
        }
        //tanquem el fitxer
}catch(excepció que el fitxer no existeix){
        //Fer alguna cosa, p. ex., tractar l'excepció.
}catch(excepció en obrir fitxer){
        //Fer alguna cosa, p. ex., tractar l'excepció.
}catch(excepció en llegir una línia del fitxer){
        //Fer alguna cosa, p. ex., tractar l'excepció.
}catch(excepció en tancar el fitxer){
        //Fer alguna cosa, p. ex., tractar l'excepció.
}
```

Com podem veure, gràcies al bloc `try-catch`, Java ens permet separar la lògica del programa de la gestió de les excepcions. El flux normal del programa està dins del bloc `try`, mentre que cada excepció es tracta en el bloc `catch` corresponent. Aquesta separació realment facilita molt la lectura i l'enteniment del codi.

3.12.2. Declarar i llançar una excepció

Continuem veient un altre codi:

```
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;
        }
    }
}
```

```
package edu.uoc.exceptions;

public class Main{
    public Main(){
        Person david = new Person();

        if(!david.setAge(-1)){
            System.out.print("Incorrect age");
        }
    }
}
```

En el codi, si passem com a argument al mètode `setAge` de la classe `Person` un valor negatiu, aquest mètode no assigna el valor i retorna `false` per avisar que hi ha hagut un error (és a dir, l'intent d'assignar una edat negativa) i no s'ha pogut realitzar l'acció esperada correctament. En la classe `Main` es comprova el valor retornat per `setAge` de l'objecte `david` per saber si aquest mètode s'ha executat correctament o no. Si el mètode `setAge` retorna el valor `false`, vol dir que aquest mètode no s'ha executat bé. Així doncs, des d'on s'ha fet la crida a `setAge`, gestionem aquest cas no desitjat advertint l'usuari final amb un missatge per pantalla.

L'ús d'un `boolean` (o de retornar un valor numèric conegut que faci de codi d'errors, com es fa en el protocol HTTP amb els codis 404, 500, etc.) és una manera molt habitual de controlar i gestionar els errors o les situacions anòmales en llenguatges basats en el paradigma de la programació estructurada. No obstant això, no és la millor manera per tractar els errors, les anomalies o els casos excepcionals. És per això que llenguatges com Java van incloure el concepte d'excepció o esdeveniment excepcional.

> Una **excepció** és un esdeveniment no desitjat que succeeix durant l'execució d'un programa i que interromp, d'alguna manera, el flux normal (o volgut/esperat) d'execució del programa.

Quan es produeix una excepció, el programa crea un objecte anomenat *exception object* que inclou informació valuosa sobre l'error, com, per exemple, el tipus d'excepció, on ha succeït aquest error, etc. L'*exception object* és llançat

(15) En anglès, *throwing an exception*.

a la JVM, que el gestiona. Aquest procés de llançar *exception objects* és el que es coneix com a llançar una excepció.[15] Vegem com seria el codi anterior mitjançant excepcions.

```
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;
  }
}
```

Veiem que el mètode `setAge` de la classe `Person` ha experimentat alguns canvis. Per començar ja no retorna res (és `void`). En segon lloc, hem afegit `throws Exception` a la seva signatura. Amb això diem que aquest mètode llança una excepció. Quan llança un mètode una excepció? Doncs, hi pot haver dos moments:

1) Quan dins del mètode es crida un altre mètode que llança una excepció (és a dir, quan la signatura d'aquest mètode que es crida té `throws Exception`) i dins d'ell es llança l'excepció.

2) Quan el programador ho indica explícitament amb la instrucció `throw new Exception("Missatge d'error per mostrar")`. En el cas de `setAge` succeeix el segon cas. Com podeu veure, en el `throw` es crea un objecte de la classe `Exception` (pertany al *package* `java.lang.Exception`).

Si el programa executa la línia de codi `throw new Exception("...")` en l'exemple anterior, el mètode s'atura i surt. És a dir, no executaria les línies restants del mètode. En el cas de `setAge`, si aquest llancés l'excepció, no s'executaria mai la línia de codi `this.age = age`. El mateix passaria si on ara tenim `throw new Exception("...")` tinguéssim en el seu lloc una crida a un altre mètode que llança una excepció i es llancés aquesta excepció.

3.12.3. Flux que segueix una excepció

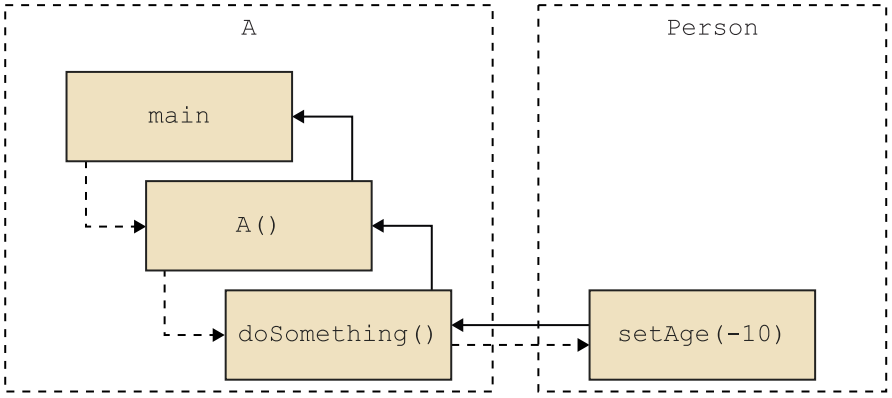
Quan se surt d'un mètode per culpa d'una excepció, el programa no s'atura de cop, sinó que llança una excepció cap amunt esperant que en algun moment es capturi aquesta excepció i es tracti. Quan diem que l'excepció es llança cap amunt, volem dir que va enrere en l'ordre d'execució del programa. Aquest ordre d'execució està format per una pila –anomenada *call stack*– amb els mètodes que han estat invocats fins que ha ocorregut l'excepció.

Així doncs, imaginem que tenim la classe `Person` definida abans, de la qual el mètode `setAge` llança una excepció quan l'edat passada com a paràmetres és negativa. I que, a més, tenim el codi següent:

```
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();
    }
}
```

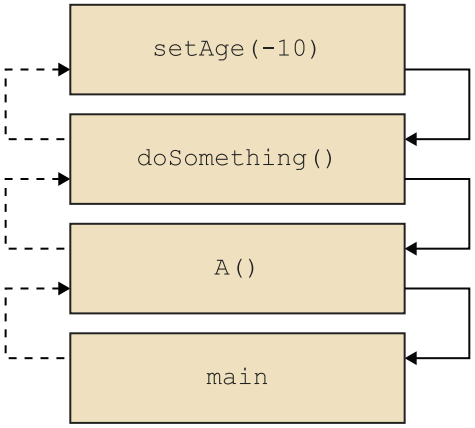
Ara executem a la consola d'ordres la instrucció `java A`. Aleshores, l'execució del programa en un format visual seria:

Figura 23



Veiem que no hi ha la crida al constructor de `Person` perquè s'executa sense problemes i diguem que desapareix. Potser si ho veiem en format *call stack* és més fàcil d'entendre:

Figura 24



En ambdues figures les fletxes discontínues indiquen crida a un mètode i les fletxes contínues són llançaments de l'excepció (és a dir, el camí que segueix una excepció fins que és tractada). Sabent això, analitzarem el programa. En primer lloc, en fer `java A`, s'executa el `main` d'aquesta classe i en ell es crida el constructor de la classe `A` quan instanciem un objecte per a la referència `objectA`. En el constructor de la classe `A` es crida el mètode `doSomething` de la classe `A` que té dues instruccions. Primer, crea un objecte de la classe `Person` (crida que, en acabar correctament, desapareix de la *call stack*) i, a continuació, crida el mètode `setAge` amb un argument negatiu (`-10`). En executar el codi del mètode `setAge` es llança una excepció perquè el valor passat com a argument no és correcte i, per tant, el flux d'execució entra en l'`if` que crea (`new`) un objecte `Exception` i el llança (`throw`). Aquest *exception object* és enviat cap amunt (fletxa contínua), ja que així ho indica l'afegitó `throws Exception` de la signatura del mètode `setAge`. Aleshores, l'objecte `Exception` creat en `setAge` és rebut pel mètode `doSomething`, que decideix si tracta l'excepció o no. En aquest cas, hem decidit que no la tracti (o com es diu informalment, que no la *capturi*). Podeu veure que hem decidit que el mètode `doSomething` no la capturi perquè la signatura del mètode `doSomething` té afegit `throws Exception`. Per tant, el mètode `doSomething` delega/passa el tractament de l'excepció del mètode `setAge` cap amunt (cap a qui l'ha cridat), en aquest cas, cap al constructor de la classe `A`. Com podem veure, el constructor de la classe `A` també delega cap amunt el tractament de l'error. Cal dir que, en qualsevol programa en Java, el mètode especial `main` és l'últim lloc en el qual podem capturar i tractar les excepcions llançades. Si en el `main` no capturem i tractem l'error, el programa acaba de sobte, que és justament el que succeirà en aquest programa.

3.12.4. Tractar/capturar una excepció

Ara aprofundirem en el bloc `try-catch` que ja hem vist. Com hem explicat, quan se surt d'un mètode per culpa d'una excepció, el programa no s'atura de cop, sinó que va retrocedint en el *call stack* per veure si algú tracta/captura l'excepció. Com indicar en un punt del codi que volem capturar l'excepció? Amb un bloc anomenat `try-catch` que té la sintaxi següent:

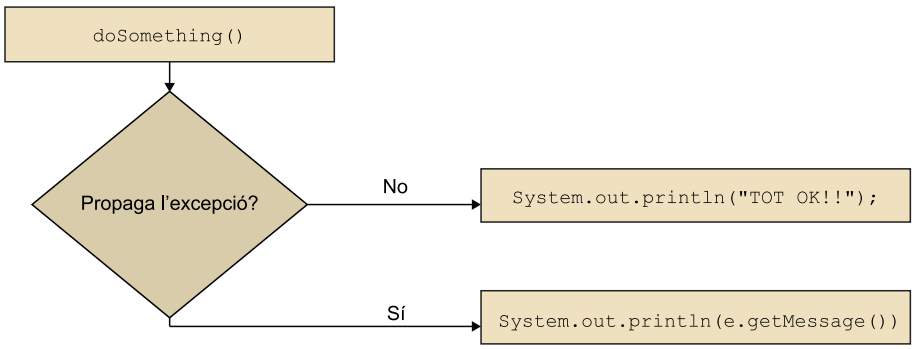
```
try{
        //Codi que pot generar una excepció
}catch(ExceptionType variable){
        //Codi que cal executar en cas de llançar-se una excepció des del codi del try
}
```

Seguirem amb l'exemple de l'apartat anterior, però lleugerament modificat per introduir un `try-catch`:

```
package edu.uoc.exceptions;

public class A{

        public A(){ //Hem tret "throws Exception" i ja no propaga l'excepció
                try{
                    doSomething();
                    System.out.println("TOT OK!!"); //s'executa si doSomething no llança
//una excepció
                }catch(Exception e){
                    System.out.println(e.getMessage()); //Imprimirà  el text "Incorrect age!"
//que indica el mètode setAge de la classe Person si doSomething llança una excepció
                }
                //Si aquí hi hagués codi sempre s'executaria…
        }

        public void doSomething() throws Exception{
                Person p = new Person();
                p.setAge(-10);
        }

        public static void main(String[] args){
                A objectA = new A();
        }
}
```

Si ens hi fixem, ara el constructor de la classe ja no propaga/delega l'excepció del mètode `doSomething` cap a qui l'ha cridat (en aquest cas, el mètode `main`), ja que no hi ha l'afegitó `throws Exception` en la signatura del constructor de la classe `A`. Ara, en la classe `A` capturem o tractem l'excepció generada per `doSomething` (si es genera). Si el mètode `doSomething` no propaga cap excepció, és a dir, en cridar `setAge` no es llança una excepció, llavors s'executaria el `println` amb el missatge `"TOT OK!!"` i el constructor de la classe `A` acabaria. En cas contrari, com passa en l'exemple, atès que `p.setAge(-10)` fa que `setAge` llanci una excepció, `doSomething` propagarà aquesta excepció i serà en el constructor de la classe `A` on es tractarà. On es tractarà? En el tros de codi del bloc `catch`. És a dir, en produir-se una excepció, totes les sentències que hi hagi dins del bloc `try` i després del mètode que ha provocat l'excepció no s'executaran. En l'exemple, no s'executarà la sentència `println` amb el text `"TOT OK!!"` i el flux del programa se n'anirà de `doSomething` al codi que hi ha en el bloc `catch`. És a dir, després de `doSomething` s'executarà el `println` que imprimeix la informació que hi ha en l'objecte `e`, en aquest cas, el missatge que té l'objecte `e` de tipus `Exception` assignat en el mètode `setAge`: `"Incorrect age!"`. Tota aquesta explicació, gràficament, seria:

Figura 25



Quan el diagrama anterior diu «Propaga l'excepció?» vol dir si el mètode `doSomething` rep una excepció llançada per `setAge` i aquesta s'ha propagat fins al constructor de la classe `A`. Si hi ha una excepció, s'atura el flux d'execució del bloc `try` i el programa segueix en el bloc `catch`, per la qual cosa executa el codi que hi ha en aquest bloc, en el cas de l'exemple: `System.out.println(e.getMessage());`. Si no hi ha excepció, l'execució continua de manera normal amb la resta del codi situat dins del `try`.

Quan fem servir el bloc `try-catch`, indiquem que en aquest punt volem capturar una excepció que es produeixi dins del `try` i tractar-la dins del `catch`.

3.12.5. *Checked* i *unchecked exceptions*

En Java, hi ha dues categories d'excepcions: les *checked* i les *unchecked*. Així doncs, definirem tots dos tipus:

Les ***checked exceptions*** (o comprovades o verificades) són aquelles per a les quals el compilador comprova que es capturin o es propaguin. En cas que una excepció sigui *checked* i ni es capturi ni es propagui, obtindrem un error en temps de compilació.

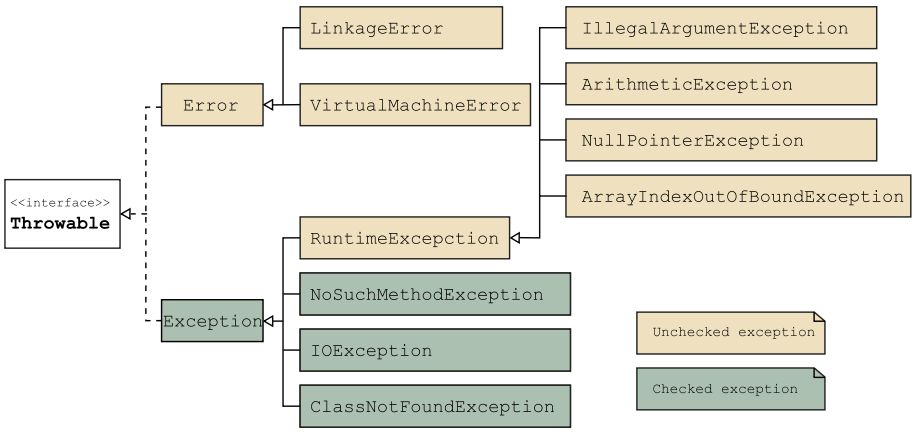
Les ***unchecked exceptions*** són les excepcions per a les quals el compilador no comprova que siguin capturades o propagades. Per tant, com a programadors no tenim obligació de capturar-les ni propagar-les.

Per què unes excepcions són *checked* i altres, *unchecked*? Java considera que les *unchecked* són totes les excepcions en les quals el programador poca cosa pot fer per aconseguir que el flux d'execució del programa no sigui alterat. En altres paraules, Java considera que de les excepcions *unchecked* és difícil recuperar-se, és a dir, trobar una solució que permeti continuar l'execució del programa com si no s'hagués donat aquesta excepció.

Per exemple, quan dividim un nombre entre zero es produeix una excepció anomenada `ArithmeticException` que és de tipus *unchecked*. Poca cosa hi podem fer, *a priori*, si hi ha una divisió entre zero, segurament es deu a una mala lògica del programa, ja que abans de dividir dos nombres hauríem de comprovar que el denominador sigui diferent de zero. El mateix passa quan intentem accedir a una casella que no existeix d'un *array*. Per exemple, si volem accedir a la casella/posició `24` d'un *array* de `5` caselles, la JVM, durant l'execució del programa (no en temps de compilació), llançarà una excepció de tipus `ArrayIndexOutOfBoundException`. En aquest cas, poca cosa hi podem fer. De nou, es deu a un error en el nostre codi: o bé hem fet servir un índex el valor del qual està mal assignat (un error clàssic quan es comença a aprendre a utilitzar els blocs d'iteració), o bé no hem comprovat que l'índex que es fa servir per accedir a un element de l'*array* estigui dins del rang correcte de l'*array*.

Les excepcions en Java estan organitzades entorn d'una jerarquia de classes i, per tant, basada totalment en el mecanisme d'herència. Molt en resum i sense entrar en gaires detalls, vegem una petita mostra d'aquestes classes relacionades amb les excepcions:

Figura 26



Totes les classes en un to més fosc són *checked*, la qual cosa vol dir que s'han de capturar o propagar obligatòriament en el nostre codi. Per què? Perquè per a aquest tipus d'errors podem gestionar què fer en el programa per solucionar l'aparició d'aquesta excepció. Per exemple, realitzar una acció alternativa, fer un *rollback* (desfer operacions que s'han fet), sortir del programa avisant amablement l'usuari final (no sobtadament) i tancant fitxers, *sockets*, connexions a bases de dades per deixar-los en un estat consistent, etc.

En canvi, totes les classes en un to clar són *unchecked* i no cal capturar-les ni propagar-les. Com podeu veure, són errors (de fet hereten/pengen de la classe `Error`, tret de `RuntimeException` i les seves subclasses) que succeeixen poc sovint i per a les quals poca cosa es pot fer com a programador per solucionar el problema. No obstant això, com a programadors, podem capturar-les i intentar tractar-les, encara que, com ja hem comentat, Java no ens hi obliga.

3.12.6. Declarar, llançar i capturar més d'una excepció

Un cop arribats a aquest punt, cal destacar que, com hem vist abans, podem tenir tants `catch` com excepcions vulguem tractar de manera diferent, ja que dins d'un bloc `try` es poden donar diferents tipus d'excepcions. És més, un únic mètode pot llançar més d'una excepció. Per exemple:

```
public void read(String fileName) throws FileNotFoundException, IOException{
        File file = new File(fileName);
        FileReader fr = new FileReader(file); //Llança FileNotFoundException
        int data = fr.read(); //Llança IOException
        while(data!=-1){
                data = fr.read(); //Llança IOException
        }
        fr.close(); //Llança IOException
}
```

Llavors, on es crida el mètode `read` podem fer:

```
try{
        read("c:/file.txt");
}catch(FileNotFoundException e){ //Si read llança FileNotFoundException, aquest catch
//el capturarà
        System.out.println("File not found");
}catch(IOException e){ //Si read llança IOException, aquest catch el capturarà
        System.out.println("Error reading file");
}
```

Imaginem que canviéssim el codi anterior per fer això:

```
try{
        read("c:/file.txt");
}catch(FileNotFoundException e){
        e.printStackTrace();
}catch(IOException e){
        e.printStackTrace();
}
```

Veiem que tant si es captura una excepció de tipus `FileNotFoundException` com `IOException`, fem el mateix: cridar el mètode `printStackTrace` de l'*exception object*. Per evitar repetir codi, des de JDK 7 és possible fer el següent:

```
try{
        read("c:/file.txt");
}catch(FileNotFoundException | IOException e){
        e.printStackTrace();
}
```

Com podem veure, el codi s'ha reduït facilitant la llegibilitat del codi.

Observeu el codi següent, sembla similar al que crida `printStackTrace` per separat, però el seu comportament no és exactament idèntic:

```
try{
        read("c:/file.txt");
}catch(IOException e){
        e.printStackTrace();
}catch(FileNotFoundException e){
        e.printStackTrace();
}
```

Fixeu-vos que el que hem canviat és l'ordre dels `catch`, en anteposar el `catch` d'`IOException` al de `FileNotFoundException`. Aquest canvi subtil pot semblar menor i que, per tant, no tingui conseqüències en el nostre codi,

però les té. Per què? Perquè mai no s'executarà el `catch` de `FileNotFoundException`. Per què? Perquè, per començar, els `catch` s'executen per ordre d'aparició. A més, `FileNotFoundException` és una subclasse de la classe `IOException` i, per tant, `IOException` és capaç de capturar tant les excepcions `IOException` com les de totes les seves subclasses. Considerant aquestes dues condicions, és lògic que mai no s'executi el `catch` de `FileNotFoundException`. Perquè ho vegeu més clar:

```
try{
        read("c:/file.txt");
}catch(Exception e){
        e.printStackTrace();
}catch(IOException e){ //Mai no s'executarà
        e.printStackTrace();
}catch(FileNotFoundException e){ //Mai no s'executarà
        e.printStackTrace();
}
```

El codi anterior utilitza en primera instància un `catch` d'`Exception`. Com ja hem vist amb anterioritat, `IOException` és una subclasse d'`Exception` i, com que `FileNotFoundException` és subclasse d'`IOException`, també ho és d'`Exception`. Així doncs, el `catch` d'`Exception` pot capturar qualsevol *exception object* que sigui d'una subclasse d'`Exception`. Sabent això, és important que en cas de necessitar un cas `default` de captura d'excepcions, és a dir, un `catch` d'`Exception`, aquest sigui l'últim que escriguem en el nostre codi.

Un cop arribats a aquest punt, potser us pregunteu: i per què no fem un `catch` d'`Exception` i així simplifiquem el codi al màxim? Doncs, perquè ens pot interessar fer accions diferents en funció del tipus d'excepció.

3.12.7. Bloc `finally`

El bloc `try-catch` pot completar-se, opcionalment, amb el bloc `finally`. És a dir:

```
try{
        //mètode que llança una excepció
        doSomething();
}catch(Exception1 e){
        //Fer quelcom, p. ex., tractar l'excepció de tipus Exception1.
}catch(Exception2 e){
        //Fer quelcom, p. ex., tractar l'excepció de tipus Exception2.
}finally{
        //Aquest bloc de codi sempre s'executa, tant si hi ha try com si hi ha catch
}
```

El bloc `finally` té la particularitat que sempre s'executa independentment del que passi. És a dir, tant si s'executa de manera correcta el bloc `try` com si es llança una excepció que capturi un `catch`, el codi situat dins de `finally` s'executarà a continuació del codi `try` o `catch`. També s'executarà si llancem una excepció i la propaguem. Abans de fer la propagació de l'excepció (i sortir

del mètode), el bloc `finally` s'executarà. L'únic cas en què no s'executa el bloc `finally` és si el codi que hi ha dins del `catch` que ha capturat l'excepció finalitza de manera inesperada.

3.12.8. Mètodes de l'*exception object*

Qualsevol objecte que sigui d'una classe que hereta de `Throwable`, és a dir, totes les que fem servir per propagar excepcions, té mètodes que ens proporcionen informació addicional sobre l'excepció. Vegem-ne els tres més interessants:

1) **`getMessage()`**. Retorna un `String` amb el text que s'hagi indicat en el constructor de l'*exception object*, per exemple, si s'ha utilitzat `Exception(String msg)` o `Throwable(String msg)`, etc.

2) **`toString()`**. Retorna un String amb una descripció breu de l'objecte. Bàsicament, el nom de la classe de tipus `Throwable` seguit del símbol `:` i el text de `getMessage()`.

3) **`printStackTrace()`**. Imprimeix per mitjà de `System.err` el contingut del mètode `toString` seguit per la traça del *call stack*.

3.12.9. Sentència `try-with-resources`

En JDK 7 es va afegir aquesta sentència per tancar recursos de manera automàtica i, d'aquesta manera, simplificar el codi. Les referències les classes de les quals implementen la interfície `AutoCloseable` poden declarar-se en el bloc `try-with-resources` i, així, els seus mètodes `close()` seran cridats automàticament després del bloc `finally`.

Vegem un exemple de codi abans de JDK 7:

```
public int read(String fileName) throws IOException{
        File file = new File(fileName);
        FileReader fr = new FileReader(file); //Llança FileNotFoundException

        try{
                return fr.read(); //Llança IOException
        }finally{
                if(fr != null) fr.close(); //Llança IOException
        }
}
```

El codi anterior, amb la nova funcionalitat `try-with-resources` de JDK 7:

```
public int read(String fileName) throws IOException{
        File file = new File(fileName);
        try(FileReader fr = new FileReader(file)){ //Llança FileNotFoundException
                return fr.read(); //Llança IOException
        }
}
```

El codi anterior realitza implícitament el mateix codi que fèiem en el bloc `finally`, però el nostre codi s'ha simplificat.

La majoria de classes i interfícies relacionades amb entrada i sortida implementen la interfície `AutoCloseable`: manipulació de fitxers (per exemple, `FileReader`), llegir i escriure dades (`InputStream`), gestió de fluxos de xarxa (`DatagramSocket`), connexió a bases de dades (`Connection`), etc.

3.13. Enumeracions

3.13.1. Concepte i sintaxi

Si heu codificat en C/C++ o altres llenguatges, deveu haver utilitzat enumeracions[16] o simplement `enum`. Com ja deveu saber, els `enum` són un tipus definit pel programador. Normalment, s'utilitzen quan volem restringir els possibles valors que pot prendre una variable/atribut o aquests valors conformen un domini concret i conegut. Per exemple, dies de la setmana, mesos de l'any, planetes del sistema solar, etc.

(16) En anglès, *enumeration*.

En Java, malgrat això, el **tipus `enum`** (introduït en JDK 1.5) és més potent que en altres llenguatges, com, per exemple, C. Per defecte, Java defineix cada `enum` com una classe i, de fet, en la seva declaració es poden incloure atributs i mètodes i, alhora, aquesta classe inclou per defecte alguns mètodes, com, per exemple, `values()`, que retorna tots els valors de l'`enum`. A més, la manera ideal de declarar un `enum` és posar-ne el codi dins d'un fitxer `.java` que tingui el mateix nom que l'`enum` (tal com es fa amb una classe). No obstant això, un `enum` es pot declarar dins d'una classe ja existent.

La sintaxi bàsica d'un `enum` en Java és:

```
modificadorAcces enum nomEnum{
    ITEM1, ITEM2, …
}
```

Per exemple:

```
public enum HandSign{
    ROCK, PAPER, SCISSORS
}

private enum Day{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Com ja hem dit, Java tracta els *enumeration* com a classes. Això sí, són classes especials que proporcionen una implementació *type-safe* (seguretat de tipus) de dades constants. És a dir, en declarar una variable del tipus de l'*enumeration*, només se li poden assignar els valors de l'*enumeration*, cap altre.

```
Day d; //Declarem una variable del tipus enum Day
d = Day.MONDAY; //Assigna un valor correcte del tipus Day (enum)
d = 0; //Error en temps de compilació
```

3.13.2. Diferència entre utilitzar constants i un *enumeration*

Comentarem la diferència que hi ha entre fer l'`enum` de `HandSign` o utilitzar constants senceres que simulen un *enumeration* de la manera següent:

```
public class Game{
        public static final int STONE = 0; //Constant
        public static final int PAPER = 1;
        public static final int SCISSORS = 2;
        …
        int playerHand = SCISSORS;
        …
}
```

Utilitzar constants té els problemes següents:

1) Com que la variable no és de tipus `enum`, podem assignar-li, per error, qualsevol valor. En l'exemple, a `playerHand` es podria assignar el valor `33` i no donaria error en temps de compilació. A més, com que els valors són constants i no `enum`, es podrien fer operacions aritmètiques sense sentit, per exemple, `PAPER + ROCK`. També podríem barrejar «enumerations» diferents que no es poden comparar: `STONE == SUNDAY`, en què `STONE` i `SUNDAY` són per al compilador només el valor `0` (no tenen una semàntica associada).

2) No crea un `namespace`. Mentre que els `enum` sempre creen el `namespace` amb el nom de l'*enumeration*. Així, per utilitzar el valor `SCISSORS`, hem de fer servir `HandSign.SCISSORS`. Què passaria si hi hagués una altra variable anomenada `SCISSORS` en una altra part del codi? Doncs, que tindríem una col·lisió i hauríem de reanomenar una de les dues variables, per exemple, `HAND_SCISSORS` en comptes de simplement `SCISSORS`.

3) Fragilitat.[17] Degut a què les constants d'un domini (que simularien un *enumeration*) es creen dins d'una classe (o fins i tot estan repetides en diverses classes), el fet de canviar el valor d'una constant o d'afegir-ne alguna de nova provoca que haguem de recompilar el/s fitxer/s que les contenen. En canvi, si fem servir un `enum` de Java, només hem de modificar el fitxer en el qual estigui declarat l'`enum` i els canvis seran propagats a tota l'aplicació.

(17) En anglès, *brittleness*.

4) `Print's` sense informació valuosa. Perquè les constants només són enters, en fer un `println`, aquest imprimeix els valors enters: `0`, `1` i `2`. Això no ens dona informació sobre el que representen. En canvi, l'`enum` imprimeix `STONE`, `PAPER` i `SCISSORS`.

Com veieu, els *enumeration* introdueixen molts avantatges respecte a simular-los amb enters o un altre tipus de dades primitives (per exemple, un `char`). Doncs bé, si els *enumeration* ja comporten avantatges per si mateix, en Java encara més, perquè, com ja hem comentat, Java tracta els *enumeration* com una classe i els afegeix característiques extra respecte als *enumeration* d'altres llenguatges de programació.

3.13.3. Compatibilitat amb `switch`

Els `enum` poden utilitzar-se amb `switch` com si es tractés d'un enter o d'un altre tipus bàsic.

```
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;
        }
    }
…
}
```

3.13.4. Implementació interna de les constants (dels valors)

A més, el compilador crea una instància de la classe per a cada constant definida dins de l'`enum`. Aquestes instàncies són `public static final`. És a dir, no es poden modificar (`final`) i s'hi accedeix anteposant el nom de l'`enum`. Per exemple: `Day.MONDAY`. Així mateix, no es pot instanciar cap objecte de tipus de l'`enum`.

3.13.5. Herència de la `classe Enum`

Quan un `enum` és definit, Java crea una classe que hereta de la classe `Enum`. Per aquest motiu, cap `enum` no pot heretar d'una altra classe o `enum`. No obstant això, podem fer que un `enum` implementi una o més interfícies.

La classe `Enum` aporta els mètodes següents:

```
public final String name() //Retorna el nom de la constant tal com es
//declara. No es pot sobreescriure.

public String toString() //Retorna el nom de la constant, tal com està
//en la declaració. Es pot sobreescriure. És més utilitzat i recomanable
//que name().

public final int ordinal() //Retorna el número ordinal de la constant.
//Comença per 0 (zero).

public static T valueOf(String name) //Retorna la constant de l'enum a partir
//d'un String de la constant.
```

Vegem-ne l'ús:

```
Day day = Day.valueOf("MONDAY"); //day apunta Day.MONDAY
System.out.println(day.name()); //Imprimirà MONDAY
System.out.println(day); //Imprimirà "MONDAY" (crida toString());
System.out.println(day.ordinal()); //Imprimirà 1
```

A més, els `enum` tenen un mètode molt interessant anomenat `values()` que retorna un *array* amb els valors de l'`enum`.

```
Day[] week = Day.values(); //retorna un array en el qual cada casella és un
//valor de l'enum Day.
```

3.13.6. Constructor i membres

Tota variable de tipus `enum` és una referència a una zona de memòria, de la mateixa manera que una classe, una interfície o un *array*. Aquesta variable és implícitament `final`, ja que els seus valors no poden modificar-se. No obstant això, l'`enum`, en ser com «una classe especial», pot definir un constructor i membres de l'`enum` (és a dir, atributs i mètodes). Això és precisament el que fa que l'`enum` de Java sigui més poderós en comparació dels `enum` d'altres llenguatges de programació. Vegem-ne un exemple:

```
public enum AlertLevel{
        HIGH (3), //crida el constructor amb valor 3
        MEDIUM (2), //crida el constructor amb valor 2
        LOW (1) //crida el constructor amb valor 1
        ; //quan s'afegeixen atributs o mètodes, cal posar;

        private final int levelIntensity;

        //Constructor
        private AlertLevel(int levelIntensity){
                this.levelIntensity = levelIntensity;
        }

        public int getLevelIntensity(){
                return levelIntensity;
        }

        @Override
        public Strign toString(){
                return getLevelIntensity()+" | "+this.name();
        }
}
```

És important destacar que el constructor d'un `enum` només pot ser `private` o `package-private` (per defecte).

3.13.7. Mètodes abstractes

En un `enum`, podem declarar un mètode abstracte i fer que cada valor, que com hem dit és una instància, en codifiqui la implementació/comportament.

```
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 AlertLevel(int levelIntensity){
                this.levelIntensity = levelIntensity;
        }

        public abstract AlertLevel next(); //Mètode abstracte
}
```

El seu ús podria ser el següent:

```
for(AlertLevel level : AlertLevel.values()){
    System.out.println(level+", the next one is "+level.next());
}
```

3.14. *Generics*

3.14.1. Concepte

Els *generics* van ser introduïts en JDK 5. Permeten la definició i l'ús de tipus (és a dir, classes i interfícies) i mètodes genèrics. Els tipus i mètodes genèrics difereixen dels tipus i mètodes «normals», ja que els primers tenen paràmetres de tipus.

Així doncs, per exemple, la classe `ArrayList<E>` és un tipus genèric (més comunament anomenat *tipus parametritzat*) que té un paràmetre de tipus anomenat `E`. Quan utilitzem `ArrayList<Person>` o `ArrayList<Animal>`, aquests s'anomenen tipus parametritzats, en què `Person` i `Animal` són els arguments de tipus, respectivament. Veurem això amb més calma a continuació.

3.14.2. Classe genèrica

Amb Java podem crear les nostres pròpies classes genèriques. Aquestes classes tenen un o més paràmetres de tipus, que són tipus genèrics. La seva sintaxi més simple és la següent:

```
public class GenericClass <T>{
       private T element;
       //TODO
}
```

Com podem veure en el codi anterior, en definir la classe `GenericClass` hem indicat que rep un **paràmetre de tipus** anomenat `T`. Això ho hem indicat mitjançant l'operador diamant (*diamond operator*), <>. D'aquesta manera, podem reutilitzar la mateixa classe per a diferents classes d'entrada. Així doncs, gràcies a les classes i interfícies genèriques podem abstreure'ns dels tipus.

A l'hora d'instanciar la classe `GenericClass`, podrem fer, per exemple:

```
GenericClass<Animal> gc1 = new GenericClass<Animal>();
GenericClass<Person> gc2 = new GenericClass<Person>();
```

En el primer cas, l'atribut privat `element` de la classe `GenericClass` serà del tipus `Animal` (és a dir, `T` és `Animal` en la primera instanciació) i en el segon cas serà del tipus `Person`.

Des de la versió 7 de Java no cal indicar la classe en cridar el constructor. És a dir, el codi anterior es pot reescriure així:

```
GenericClass<Animal> gc1 = new GenericClass<>();
GenericClass<Person> gc2 = new GenericClass<>();
```

Gràcies a JDK 7 es redueix la verbositat, ja que <> (anomenat *diamond operator*) infereix el paràmetre de tipus a partir de l'especificat com a tipus per a l'atribut o la variable que s'instancia.

3.14.3. Avantatges de *generics*

Un cop arribats a aquest punt, encara que només hem parlat de classes genèriques, veurem quins avantatges té l'ús de *generics*:

- **Comprova els tipus en temps de compilació**. El compilador comprova que el tipus amb el qual s'instància la classe genèrica no viola la seguretat de tipus. Si això succeeix, el compilador llançarà un error en temps de compilació, la qual cosa és molt millor que en temps d'execució.

```
GenericClass<Integer> gc1 = new GenericClass<Double>(36.5); //error de compilació
```

- **Elimina la necessitat de fer càsting**. Com que durant la instanciació indiquem el tipus de la classe que fa de paràmetre en la classe genèrica, no cal fer càsting quan recuperem l'objecte. Això, a més, millora la llegibilitat del nostre codi.

```
ArrayList<String> lista = new ArrayList<String>(); //ArrayList és genèrica
lista.add("Hola!");
String s = lista.get(0); //no cal fer cast: (String) lista.get(0);
```

- **Facilita als programadors crear classes** (i sobretot, algorismes) que siguin genèriques. Això ajuda a reutilitzar una gran quantitat de codi en diferents programes. Aquest avantatge queda manifest de manera clara en l'ús de classes proporcionades per Java que formen el conjunt de col·leccions.

Vegeu també

Vegeu l'apartat «Col·leccions» d'aquesta guia.

Vegem ara l'avantatge d'utilitzar una classe genèrica en comptes de fer ús de la classe `Object` per a la mateixa finalitat.

```
public class A{
    private Object obj;

    public void set(Object obj){
        this.obj = obj;
    }

    public Object get(){
        return obj;
    }
}
```

```
public class A <T>{
    private T obj;

    public void set(T obj){
        this.obj = obj;
    }

    public T get(){
        return obj;
    }
}
```

Els dos codis anteriors són dues maneres de codificar la classe `A`. En el primer, utilitzem la classe arrel `Object` de la qual hereten totes les classes. En canvi, en el segon codi hem convertit la classe `A` en una classe genèrica.

A priori pot semblar que són el «mateix» codi, però hi ha diferències significatives en l'ús. En el codi anterior, de fet, no hi ha gaires avantatges d'utilitzar un codi o l'altre. L'avantatge més gran és que la classe del primer codi pot rebre qualsevol classe i retornar qualsevol classe perquè treballem sempre amb `Object`. En canvi, el segon codi permet definir amb quina classe treballa la classe genèrica i, llavors, el compilador pot detectar errors i que la classe `A` no sigui utilitzada amb classes que no hem definit en instanciar-la.

```
A objA = new A();
objA.set("hola!") ;
Integer a = objA.get(); //error en execució
objA.set(50); //OK
```

```
A objA = new A<String>();
objA.set("hola!");
Integer a = objA.get(); //error compilació
objA.set(3); //error compilació
```

D'altra banda, què passaria si tinguéssim un mètode en la classe `A` que fes ús de l'atribut `obj` per cridar un dels seus mètodes?

```
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();
    }
}
```

```
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();
    }
}
```

Com podem veure, en el primer codi, aplicant polimorfisme i un *downcasting*, l'atribut `obj` pot ser qualsevol classe. No obstant això, no podem garantir la seguretat de tipus perquè necessitem convertir explícitament d'`Object` a una altra classe, i ja sabem que en fer un *downcasting* és freqüent crear accidentalment errors en temps d'execució. Així mateix, què succeeix si la classe `A` també ha de servir per a la classe `Person` que també té el mètode *greeting* però no està relacionat amb la classe `Dog`? Doncs que el fet d'haver de fer un càsting explícit no ens permet generalitzar l'ús de la classe `A`. Si féssim servir la classe `A` exclusivament per a `Dog` i `Cat`, n'hi hauria prou de fer un *downcasting* a `Animal`, i el primer codi funcionaria quan `obj` fos `Dog` o `Cat`.

3.14.4. Interfície genèrica

De manera similar a una classe, també podem definir les nostres pròpies interfícies genèriques.

```
public interface GenericInterface <T>{
    void doSomething(T element);
}
```

Cal tenir present que una interfície de Java que és genèrica és `Comparable`.

3.14.5. Mètode genèric

Els mètodes genèrics són similars conceptualment a la classe genèrica, però el paràmetre de tipus només afecta el mètode, no tota la classe. Un mètode genèric pot estar tant en una classe o interfície genèrica com normal. Hem vist el primer cas quan hem definit el mètode `doSomething` en la interfície `GenericInterface` en el subapartat anterior. Ara vegem la definició d'un mètode genèric en una classe normal.

```
public class A{
	public <T> boolean findItem(T[] list, T item){
		for(T element : list){
			if(element==item) return true;
		}
		return false;
	}
}
```

Com podem veure, abans de la declaració del tipus de retorn, hem de posar els paràmetres de tipus dins de <>.

Per cridar aquest mètode que troba un element determinat en un *array*, podem fer:

```
A objA = new A();
String names = {"David", "Elena", "Marina", "Pau"};
System.out.println(objA.<String>findItem(names,"Elena")); //true
System.out.println(objA.findItem(names,"Elena")); //true
```

Com veiem, el mètode *generic* es pot anomenar de dues formes diferents: la primera és indicant l'argument de tipus i l'altra és sense indicar-lo (deixant que el compilador l'infereixi).

Vegem-ne dos exemples més:

```
public class B{
	public static <T> int boolean countNumItems(T[] list){
		return list.length;
	}
	public <T> T getLastItem(T[] list){
		return list[list.length-1];
	}
}
```

3.14.6. Paràmetres de tipus

En primer lloc, és important destacar que podem tenir un o més paràmetres de tipus. Per exemple:

```
public class GenericClass <T,S>{
	private T field1;
	private S field2;
	//TODO
}
```

Així mateix, cal indicar que, per conveni, el **nom** del paràmetre de tipus sol ser un únic caràcter escrit en majúscules. Els noms més habituals són:

- `E`: utilitzat per a elements d'una col·lecció.
- `T`: tipus (és a dir, una classe, una interfície o un tipus parametritzat).

Vegeu també

Vegeu l'apartat «Col·leccions» d'aquesta guia.

- `S`, `U`, `V`, etc.: per a segon, tercer, quart tipus, etc. En una classe genèrica podem tenir més d'un paràmetre de tipus.

- `K`: clau (per a classes que modelen una clau-valor).

- `V`: valor.

D'altra banda, els tipus que poden utilitzar-se com a paràmetres de tipus només poden ser classes o interfícies. Així doncs, si volem fer servir tipus primitius, no podem. La solució serà utilitzar les corresponents **classes *wrapper*** (embolcall). Aquestes són:

- Per al tipus primitiu `byte`, la classe *wrapper* és `Byte`.
- Per al tipus primitiu `short`, la classe *wrapper* és `Short`.
- Per al tipus primitiu `int`, la classe *wrapper* és `Integer`.
- Per al tipus primitiu `long`, la classe *wrapper* és `Long`.
- Per al tipus primitiu `float`, la classe *wrapper* és `Float`.
- Per al tipus primitiu `double`, la classe *wrapper* és `Double`.
- Per al tipus primitiu `char`, la classe *wrapper* és `Character`.
- Per al tipus primitiu `boolean`, la classe *wrapper* és `Boolean`.

3.14.7. Subtipus

Podem tenir un subtipus d'una classe o interfície genèrica en heretar d'ella o en implementar-la, respectivament:

```
public interface MyList<E,T> extends List<E>{
        //TODO
}
```

Amb el codi anterior creem una interfície anomenada `MyList` que hereta de la interfície de Java anomenada `List`.[18] Si `List` fos `List<Integer>`, la nostra interfície `MyList` podria ser `MyList<Integer,Object>`, `MyList<Integer,Integer>`, `MyList<Integer,Double>`, `MyList<Integer,String>`, etc.

[18] En parlarem a l'apartat següent.

3.14.8. Utilitzar tipus parametritzats com a paràmetre de tipus

Un paràmetre de tipus pot ser un tipus parametritzat. Per exemple:

```
public class A<T,S>{
        public void doSomething(T obj1, S obj2){
                //TODO
        }
}

A<String, List<String>> lista = new A<>(); //List és una interfície genèrica
```

3.14.9. Ús del *wildcard* ?

Amb els *generics* podem utilitzar el comodí[19] `?`. Aquest comodí representa un tipus desconegut o, dit d'una altra manera, qualsevol tipus. Aquest comodí pot ser utilitzat com a paràmetre de tipus, tipus d'un atribut o, fins i tot, com a tipus del retorn d'un mètode (encara que això últim no és aconsellable). No obstant això, `?` no pot ser utilitzat per cridar mètodes genèrics ni per instanciar objectes de classes genèriques.

(19) En anglès, *wildcard*.

```
public class A{
        public void areTheSame(? obj1, ? obj2){
                return obj1.equals(obj2);
        }
}
```

Un altre exemple podria ser:

```
public class A{
        public void getSize(?[] list){
                return list.length;
        }
}
```

Així doncs, l'ús del comodí `?` seria recomanable quan:

- poguéssim utilitzar en lloc d'ell `Object`.
- els mètodes en els quals l'utilitzem no depenen del paràmetre de tipus, per exemple, `list.length`, etc.

3.14.10. Limitació de tipus

Fins ara hem vist que els paràmetres de tipus podien ser substituïts per qualsevol classe o interfície. Amb tot, això de vegades no és el que volem i preferim limitar els tipus que es poden passar a un paràmetre de tipus.

Un exemple clar és si volem que els tipus que es rebin siguin numèrics. És a dir, `Integer`, `Float` o `Double`, però no `String` ni un altre tipus. Si tinguéssim el codi següent, aquest no funcionaria:

```
public class Maths <T>{
        public T add(T t1, T t2){
                return t1 + t2;
        }
        //TODO
}
```

Amb el codi anterior, el compilador ens donaria error perquè no sap que la nostra intenció és utilitzar la classe `Maths` només amb tipus numèrics i que, per tant, sumar dos elements és correcte. Així doncs, hem d'indicar-li que només farem servir classes que siguin numèriques. És a dir, limitarem els tipus que pot acceptar el paràmetre de tipus `T`. Això ho farem amb els *bounded types* (és a dir, tipus limitats). El codi anterior mitjançant *bounded types* seria:

```
public class Maths <T extends Number>{ //es podria haver substituït T per ?
        public T add(T t1, T t2){
                return t1 + t2;
        }
        //TODO
}
```

Com podem veure, hem utilitzat la paraula reservada `extends` per dir que tots els tipus que rebi `T` han d'heretar de la classe `Number`. En aquest cas, tant `Integer` com `Float` i `Double` hereten d'aquesta classe. Ara podrem fer:

```
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 compilació
Maths m4 = new Maths<Person>(); //error de compilació
```

Un altre exemple en el qual indiquem que els tipus passats al paràmetre de tipus compleixen una condició és el següent:

```
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];
                        }
                }
        }
}
```

Gràcies a l'ús d'`extends Comparable` li diem que tots els tipus que rep el paràmetre de tipus `T` han d'implementar `Comparable`.

```
String names = {"David", "Elena", "Marina", "Pau"};
System.out.println(C.findSmallestItem(names)); //David; La classe String implementa Comparable

Person[] people = {new Person("David"), new Person("Elena")};
System.out.println(C.findSmallestItem(names)); //error; La classe Person no implementa
//Comparable, si l'implementés, funcionaria correctament.
```

Podem indicar més d'una restricció per a un mateix paràmetre de tipus mitjançant `&`:

```
public class Maths <T extends Number & Comparable>{
        //TODO
}
```

L'ús d'`extends` ens permet indicar un límit superior. Per aquest motiu, la combinació d'`extends` seguit d'una classe s'anomena *upper bound*. Si en comptes d'utilitzar un nom, per exemple, `T`, es fa servir el comodí `?`, llavors el conjunt s'anomena *upper bound wildcard*. En qualsevol cas, el que ens permet l'ús d'`extends` és dir que allò que hi ha a l'esquerra de l'`extends` ha de ser un subtipus de la classe/interfície que escriguem a la dreta o la mateixa classe/interfície. Així doncs:

```
public class Maths <T extends Number>{
        //TODO
}
```

El codi anterior ens diu que el que posem en `T` ha de ser o `Number` o una subclasse de `Number` (és a dir, una classe que hereti directament o indirectament de `Number`).

Vegem-ne un altre exemple mitjançant `List`. Entre el codi següent:

```
public void doSomething(List<? extends Number> elements){
        //TODO
}
```

I aquest:

```
public void doSomething(List<Number> elements){
        //TODO
}
```

Hi veieu la diferència? Amb el primer, `elements` podria ser qualsevol `List`, és a dir: `List<Integer>`, `List<Double>`, etc., mentre que amb el segon codi només podríem passar-li com a arguments un `List<Number>`. Encara que `Integer` és un subtipus de `Number`, `List<Integer>` i `List<Number>` no estan relacionats, ja que `List<Integer>` no és un subtipus de `List<Number>`. L'element comú/pare de `List<Integer>` i `List<Number>` és `List<?>`.

Hi ha un altre limitador anomenat `super`. Amb ell diem que el tipus que s'utilitza (costat esquerre) ha de ser una superclasse o superinterfície de l'element indicat a la dreta o l'element mateix. És a dir:

```
<? super Animal>
```

Ens diu que el que utilitzem (?) ha de ser o `Animal` o una superclasse directa o indirecta d'`Animal`.

Ha arribat el moment d'enfrontar-nos a la gran pregunta: quan utilitzar `extends`, `super` o cap d'ells?

- Utilitzarem `extends` quan només vulguem consultar els valors (és a dir, accés de lectura).
- Utilitzarem `super` quan només vulguem escriure/afegir valors (és a dir, accés d'escriptura).
- No utilitzarem ni `extends` ni `super` si necessitem tant llegir com escriure.

3.14.11. Restriccions en l'ús de *generics*

Els *generics* tenen algunes restriccions:

1) No es poden instanciar tipus genèrics amb tipus primitius.

```
ArrayList<int> list = new ArrayList<int>(); //KO: ArrayList és una classe generic
ArrayList<Integer> list = new ArrayList<Integer>(); //OK
```

2) No es poden crear instàncies dels paràmetres de tipus.

```
public class A<T>{
        public A(){
            new T(); //KO
            T lista = new T[10]; //KO
        }
}
```

3) No es poden declarar atributs `static` els tipus dels quals són paràmetres de tipus.

```
public class A<T>{
        private static T field1; //KO
}
```

4) No es poden fer càstings amb tipus parametritzats:

```
List<Integer> lista = new ArrayList<>();
List<Number> lista2 = (List<Number>) lista; //KO
```

5) No es pot utilitzar `instanceof` amb tipus parametritzats.

```
public <E> void doSomething(List<E> list){
        if(list instanceof ArrayList<Integer>){ //KO

        }
}
```

6) No es poden crear *arrays* de tipus parametritzats.

```
ArrayList<Integer> [] elements = new ArrayList<Integer>[10]; //KO
```

7) No es poden crear, capturar o llançar tipus parametritzats que estenguin de `Throwable`.

```
public class MyOwnException<T> extends Exception{} //KO: Exception
//hereta de Throwable
```

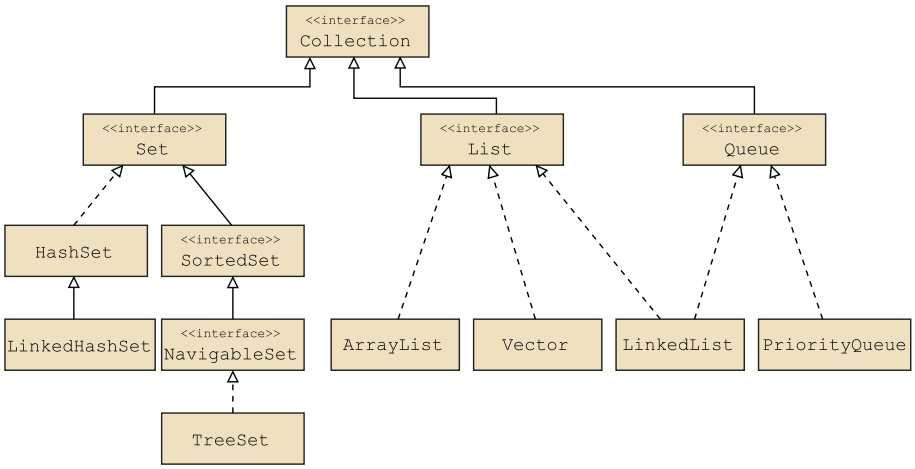
8) No es pot sobrecarregar un mètode que tingui la mateixa signatura que un altre després del *type erasure* (és el procés que segueix Java a l'hora de convertir el codi a *bytecode*).

```
public class A{
        public void do(List<T> tList){}
        public void do(List<Integer> intList){}
        public void do(List<V> vList){} //KO: els 3 mètodes són el mateix

        }
}
```

3.15. Col·leccions

En Java, hi ha un conjunt de **classes genèriques/parametritzades** que es comporten com una estructura de dades que guarden objectes de qualsevol classe. En aquest sentit, un subconjunt dins del conjunt de classes que modelen una estructura de dades són les classes que implementen la interfície `Collection`. Podeu veure una part del diagrama de classes a continuació.

Figura 27



Totes les classes representades eliminen dues restriccions de l'*array*:

1) Indicar obligatòriament quan es crea/s'inicialitza el nombre d'ítems (és a dir, caselles) que tindrà.

2) No poder ampliar o reduir el nombre de caselles (és a dir, elements que cal desar) una vegada creat l'*array*.

Com es pot veure en el diagrama de la figura 27, hi ha tres interfícies (`Set`, `List` i `Queue`) que hereten de la interfície `Collection`. Les classes que implementen aquestes tres interfícies permeten afegir un nombre «infinit» d'elements (és a dir, l'asterisc dels diagrames de classes UML).

3.15.1. Interfície `Set`

La primera interfície que tractarem serà `Set`.

> Cadascuna de les classes que implementen `Set` es comporta com una estructura de dades que guarda un conjunt d'objectes únics. És a dir, les classes que implementen `Set` no permeten elements duplicats.

No obstant això, les diferents classes que implementen `Set` afegeixen funcionalitats i comportaments. Per exemple:

1) Ordenació. `HashSet` desa els elements sense seguir un ordre lògic, mentre que `LinkedHashSet` desa els elements pel seu ordre d'inserció i `TreeSet` els desa seguint o bé les indicacions que codifiquem en el mètode `compareTo()` (per utilitzar aquest mètode, la classe de l'objecte que desem ha d'implementar la interfície `Comparable` i sobreescriure aquest mètode), o bé mitjançant un objecte d'una classe que implementi la interfície `Comparator` i, en conseqüència, sobreescrigui el mètode `compare` (aquest objecte es pot passar com a argument en un dels constructors de `TreeSet`).

2) Element `null`. `TreeSet` no permet inserir l'element `null`, mentre que `HashSet` i `LinkedHashSet` sí que permeten l'element `null`, però només una vegada, ja que la interfície `Set` no permet elements duplicats.

En línies generals, de les tres classes se sol utilitzar `HashSet`, ja que és millor en rendiment, tret que realment es necessiti desar elements únics (és a dir, sense duplicats) en un ordre específic. En aquest cas, s'utilitza `LinkedHashSet` o `TreeSet`.

3.15.2. Interfície `List`

Com podeu veure en el diagrama UML anterior (figura 27), una altra branca és la que comença amb la interfície `List`. Com es pot veure, hi ha tres classes (en realitat n'hi ha més) que implementen `List`: `ArrayList`, `LinkedList` i `Vector`. Aquestes classes, les més utilitzades de `List`, tenen en comú que es comporten com una llista (per aquest motiu implementen la interfície `List`).

> Les tres classes permeten inserir objectes duplicats i, a més, els objectes inserits estan ordenats per índex.

Tant `ArrayList` com `LinkedList` no són sincronitzats. Això vol dir que no poden ser utilitzats directament en programes amb més d'un fil d'execució (és a dir, més d'un *thread*; *multi-threading*). Per utilitzar-los amb més d'un *thread* s'han de sincronitzar externament. El concepte de *thread* s'estudia, per exemple, en les assignatures «Sistemes operatius» i «Sistemes distribuïts», així que ara com ara no us preocupeu d'això. Per la seva banda, `Vector` és similar a `ArrayList`, excepte que és sincronitzat. Així doncs, es prefereix utilitzar `ArrayList` quan no es programa amb fils múltiples.

Tal com estan implementats `ArrayList` i `LinkedList`, els requisits de memòria són menors per a `ArrayList`. De fet, `ArrayList` està basat en el concepte d'*array* dinàmic (*dynamic array*), mentre que `LinkedList` ho està en el de *doubly linked list*.

Si es fan moltes insercions (és a dir, `add`), especialment al mig de la llista, és preferible utilitzar `LinkedList` que `ArrayList`, ja que el cost és menor. Per contra, si les insercions es fan al final, `ArrayList` té un rendiment una mica millor, tret que es passi de la capacitat inicial molt sovint i s'hagi d'augmentar moltes vegades l'*array* que l'implementa.

D'altra banda, si es fan més consultes d'accés aleatori (és a dir, `get`) que insercions, és preferible utilitzar `ArrayList`, ja que `LinkedList` recorre tota la llista fins a arribar a la posició/índex indicat. L'esborrament, sobretot si es fa al començament o al mig de la llista, té un cost millor en `LinkedList`, ja que si s'elimina un objecte del mig, no s'han de moure tots els objectes posteriors, com succeiria amb l'`ArrayList`.

Pel fet que `LinkedList` també implementa la interfície `Queue`, inclou mètodes propis de les cues, com, per exemple, `peek` (obtenir el primer element de la llista) o `poll` (obtenir el primer element de la llista i esborrar-lo de la llista).

Finalment, arribem a la branca de la interfície `Queue` (en català, cua), les classes de la qual segueixen la filosofia FIFO.[20] Com ja hem esmentat, `LinkedList` implementa `Queue`, a més de la interfície `List`, i té un comportament híbrid. L'altra classe que implementa `Queue` és `PriorityQueue`, que es tracta d'una cua amb prioritat.

[20] Acrònim de *First-In, First-Out*.

3.15.3. Interfície `Map`

El que hem vist fins ara són estructures que, més o menys, coneixeu, almenys, de manera conceptual: llista, cua i conjunt (*set*). Ens ha faltat veure la pila, una estructura que també coneixeu i que segueix la filosofia LIFO[21] i que en Java està codificada amb la classe `Stack` que hereta de la classe `Vector`.

[21] Acrònim de *Last-In, First-Out*.

Ara volem animar-vos a conèixer unes noves estructures de dades, molt freqüents en el món de la programació, que es basen en el concepte d'*array associatiu*. Un *array* (o matriu) associatiu (també conegut com a *taula hash* o *taula de dispersió*) és una estructura de dades molt útil que consisteix en un conjunt de parells (clau, valor). Molts llenguatges de programació, com PHP i Javascript, implementen la funcionalitat per treballar amb aquestes estructures. Vegem l'exemple fàcil de Javascript:

```
let items = []; //declarem un array, no entrarem en detalls
items["9781412902243"] = "Terracota Warriors";
items["9781412902244"] = "Venus de Milo";
items["9781412902245"] = "David by Donatello";
items["9781412902246"] = "The thinker by Rodin";
```

Si analitzem el codi Javascript anterior, el que fem és crear un *array* (sense longitud, això ho permet fer Javascript) i a cada casella, en comptes d'utilitzar els índexs `0`, `1`, `2` i `3`, fem servir una clau, en aquest cas de tipus textual (`String`). Ara ja no utilitzem `0`, `1`, `2` i `3` per accedir a les caselles de l'*array*, sinó claus. Així doncs, en la casella amb clau igual a `"9781412902243"` (que no vol dir que sigui ni la casella `0` ni la casella situada en la posició `9781412902243` de l'*array*) hi ha el valor `"Terracota Warriors"`. Per tant, fem servir una estructura basada en *un parell clau-valor*.[22] Segurament us deveu preguntar: «per a què pot ser útil un *array* associatiu?» Doncs, per exemple, imaginem que la clau és el nom curt utilitzat per a cadascun dels equips (`Team`) que tenim en un joc de futbol i dins de la casella corresponent desem un objecte de la classe `Team`. Gràcies a l'*array* associatiu serem capaços d'accedir de manera directa a tot l'objecte `Team` si en sabem el nom curt únic. Si utilitzem un altre tipus d'estructura, per exemple, una llista, per poder trobar l'ítem amb identificador `"Barça"`, hauríem de fer una cerca per tota l'estructura fins a trobar-lo. Això sí, potser us ho plantegeu, però cada objecte desat en *l'array* associatiu ha de tenir una **clau única**; si no, només podrem desar un dels dos objectes com a valor d'aquella clau. Per exemple:

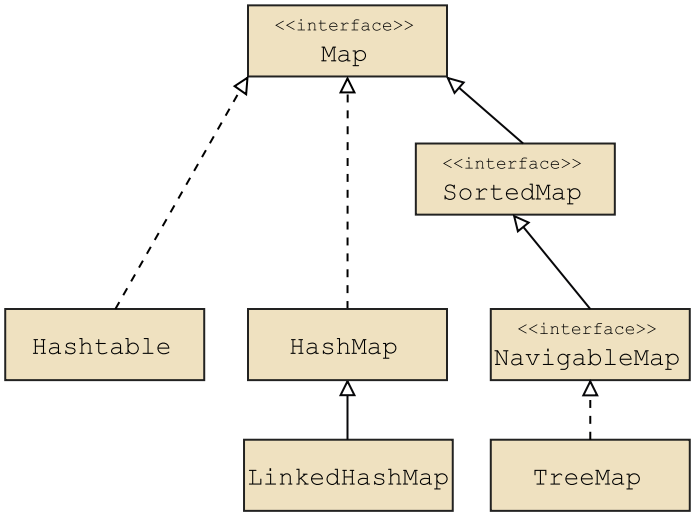
[22] En anglès, *key-value pair*.

```
items["9781412902243"]= "Terracota Warriors";
items["9781412902243"]= "Venus de Milo";
```

En aquest cas, el parell clau-valor desat serà `"9781412902243"-"Venus de Milo"`, és a dir, en la casella `"9781412902243"` no hi haurà `"Terracota Warriors"`.

En Java, les classes associatives es modelen, principalment, amb les classes que implementen la interfície `Map`.

Figura 28



Bàsicament s'utilitzen quatre classes: `HashMap`, `Hashtable`, `LinkedHashMap` i `TreeMap`, que tenen un comportament similar al que hem comentat, amb les seves particularitats. En qualsevol cas, la clau ha de ser un objecte (d'una classe *wrapper* si volem utilitzar els tipus bàsics –per exemple, `Integer` en comptes d'int `int`, `String`, `Float` en comptes de `float`, etc.– o d'una classe creada per nosaltres).

Ni `HashMap` ni `Hashtable` desen els elements ordenats, ni per clau ni per valor. Per contra, els elements dins de `TreeMap` estan ordenats per clau mitjançant el mètode `compare` de la interfície `Comparator` i en `LinkedHashMap` s'ordenen per ordre d'inserció.

`Hashtable` és com `HashMap`, però sincronitzat. Això vol dir que es pot fer servir en programes amb més d'un fil d'execució. Òbviament, aquesta característica comporta un sobrecost al programa.

Una altra diferència és que `Hashtable` no permet valors ni claus `null`, mentre que `HashMap` i `LinkedHashMap` sí que permeten valors `null`, i només una clau `null`. Per la seva banda, `TreeMap` només permet valors `null`, no claus.

3.15.4. Resum

Amb l'explicació de les classes que implementen la interfície `Map`, sembla que tot són avantatges i el millor és utilitzar una estructura de dades de tipus `Map`, en comptes de tipus `List` o `Set`. Bé, doncs, evidentment, cada classe té el seu ús. Per exemple:

- Les classes procedents de `List`, com, per exemple, `ArrayList`, són útils si necessitem desar l'ordre dels objectes que emmagatzemem.

- A més, les classes de tipus `List` consumeixen menys memòria, ja que només desen l'objecte que inserim, mentre que les que són de tipus `Map` desen dos objectes: la clau i el valor.

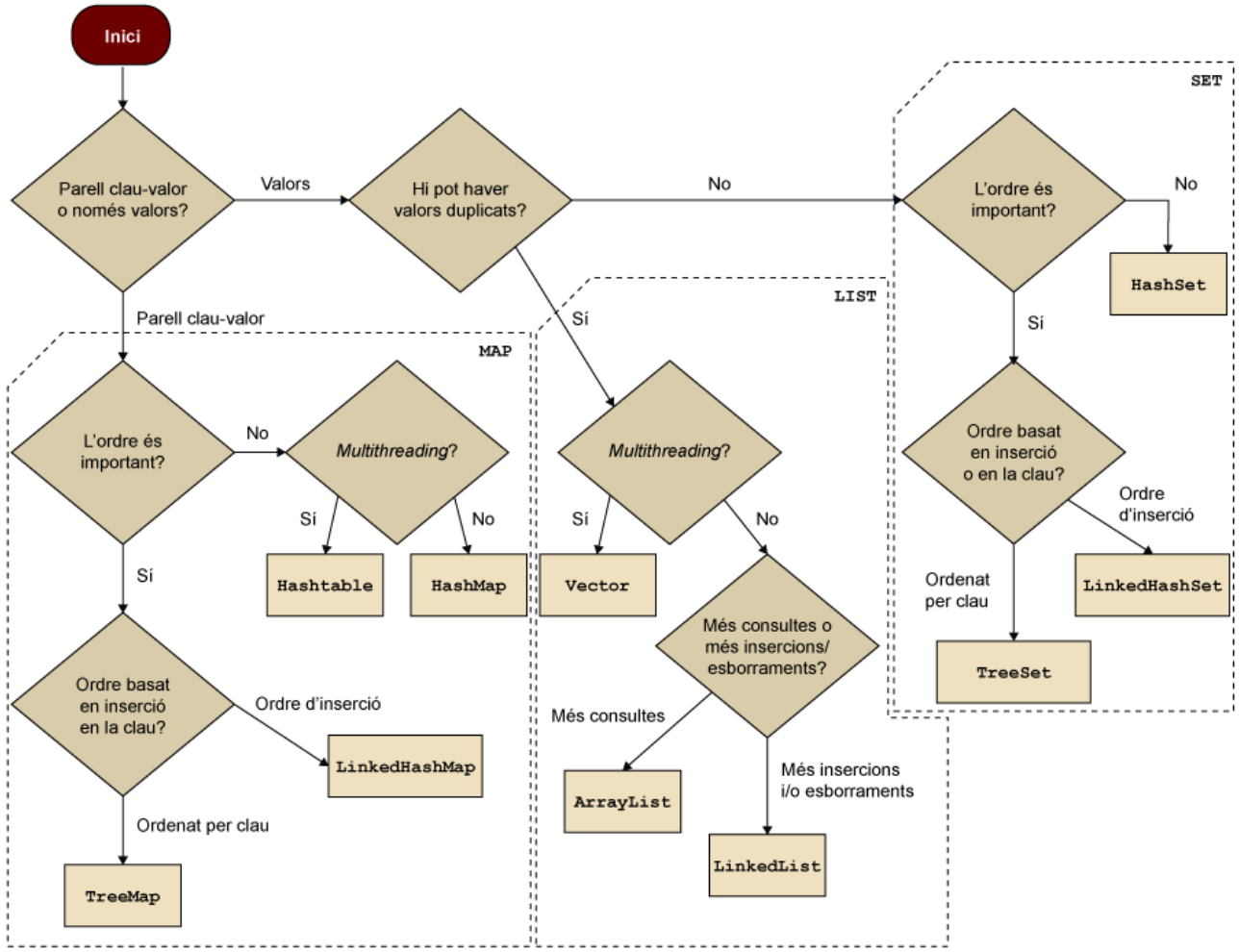
- Les classes de tipus `List` poden desar duplicats de valor, mentre que les de tipus `Map` només poden duplicar valors, no claus. Les classes de tipus `Set` no permeten duplicats.

- Les classes de tipus `List` i `Map` permeten el valor `null`. Això sí, les de tipus `Map` només permeten `null` una vegada com a clau.

- A diferència de les classes que implementen `List` o `Map`, les classes que implementen `Set` no tenen un mètode per accedir directament a un element desat (conegut com a *accés aleatori*), sinó que s'ha d'iterar sobre tota la col·lecció fins a arribar a l'element volgut.

En la figura 29, facilitem un diagrama de decisió que us pot ajudar a l'hora d'escollir una estructura o una altra. És un diagrama genèric i, per això, serà el problema que s'hagi de tractar i el vostre coneixement de les estructures allò que finalment acabarà determinant la vostra decisió.

Figura 29


Inici
Parell clau-valor o només valors?
Valors
Hi pot haver valors duplicats?
No
L'ordre és important?
No
HashSet
SET
Sí
Ordre basat en inserció o en la clau?
Ordre d'inserció
LinkedHashSet
Ordenat per clau
TreeSet
Parell clau-valor
MAP
L'ordre és important?
No
Multithreading?
Sí
No
Hashtable
HashMap
Sí
Ordre basat en inserció en la clau?
Ordre d'inserció
LinkedHashMap
Ordenat per clau
TreeMap
Sí
LIST
Multithreading?
Sí
No
Vector
Més consultes o més insercions/ esborraments?
Més consultes
ArrayList
Més insercions i/o esborraments
LinkedList

4. Avançat

4.1. La classe `String`

A diferència d'altres llenguatges de programació com C o C++, en què un *string* és un *array* de `char`, l'`String`, en Java, com ja sabem, és una classe que hereta, com qualsevol altra classe en Java, de la classe `Object`. No obstant això, la classe `String` és especial respecte a la resta de classes perquè:

1) Podem assignar un text amb cometes dobles (*String literal*) a una variable de tipus `String` sense necessitat de cridar el constructor per crear una instància de la classe `String`.

```
String text = "Hello!";
```

Així doncs, un objecte `String` pot crear-se com si fos un tipus primitiu (per exemple, com fa un enter: `int a = 5;`) o com un objecte:

```
String s = new String("Hi!");
```

Val a dir que el més habitual és construir-lo com si es tractés d'un tipus primitiu, la qual cosa s'anomena *String literal* per diferenciar-ho de la segona forma anomenada *String object*.

2) L'operador + està sobrecarregat per poder concatenar dos `String`. Aquest operador no funciona amb objectes d'altres classes. Així doncs:

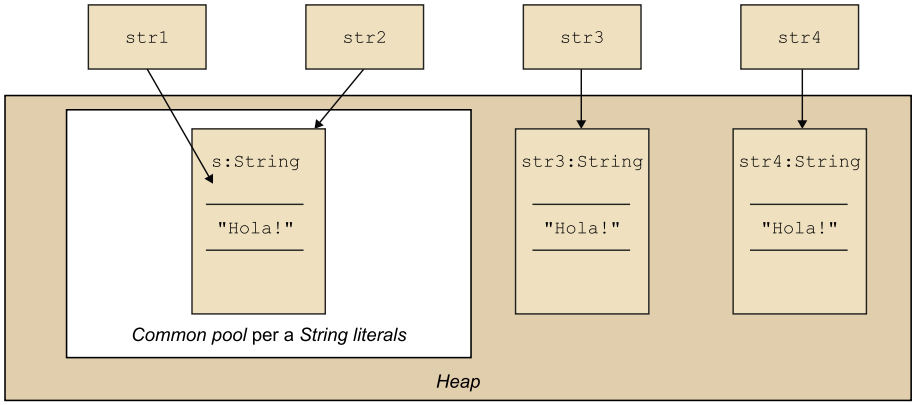
```
String str1 = "Hola" + " David!";
```

3) Els *String literals* són emmagatzemats com a objectes en un lloc comú anomenat *String common pool* dins del *heap*. Això facilita la compartició i l'eficiència de l'emmagatzematge. Per la seva banda, quan l'`String` és creat amb la crida explícita al constructor, es crea un objecte (que anomenarem *String object* per distingir-lo de l'*String literal*) que s'emmagatzema directament en el *heap*. Mireu aquest codi:

```
String str1 = "Hola!";
String str2 = "Hola!";
String str3 = new String("Hola!");
String str4 = new String("Hola!");
```

El codi anterior, gràficament, seria:

Figura 30



Així doncs, si dos *String literals* tenen el mateix contingut, apunten i comparteixen el mateix espai de memòria en l'*String common pool*. Per la seva banda, cadascun dels `String` creat com a objecte (és a dir, amb `new`) es comporta com qualsevol objecte d'una altra classe que ocupa el seu propi espai en el *heap*, encara que el contingut de dos *String objects* sigui el mateix.

4) Pel fet que els *String literals* amb un contingut idèntic comparteixen la zona de memòria en l'*String common pool*, Java defineix `String` com a **immutable**. És a dir, el contingut no pot ser modificat un cop ha estat creat. Per aquest motiu, quan cridem un mètode com `toLowerCase()`, aquest mètode no modifica l'`String` que li passem, sinó que en crea un de nou amb totes les lletres en minúscules. Això es fa així perquè `String` es comporti com a immutable, però aquest comportament pot crear-nos problemes:

```
String str1 = "Hola";
str1.concat(" David!");
System.out.println(str1); //Imprimeix "Hola"
```

En el codi anterior, tindríem en l'*String common pool* dos `String`, un amb el text `"Hola"` i un altre amb el text `"Hola David!"`. El problema és que mentre que `str1` apunta `"Hola"`, ningú no apunta `"Hola David!"`, la qual cosa faria que malgastéssim memòria. És veritat que Java té un mecanisme especial anomenat *garbage collector* que s'encarrega d'anar esborrant periòdicament i automàticament els objectes sense referència, però, que es facin excessives crides al *garbage collector* afectarà greument el rendiment del programa.

Per veure més clara la diferència entre *String literal* i *String object*, vegem l'exemple següent:

```
String str1 = "Hola";
String str2 = "Hola";
String str3 = new String("Hola");

if(str1 == str2){
    System.out.println("str1 == str2");
}

if(str1 != str3){
    System.out.println("str1 != str3");
}
```

En l'exemple anterior, s'imprimiran per pantalla tots dos missatges, ja que `str1` i `str2` són el mateix objecte en l'*String common pool*, mentre que `str3` és un objecte situat en el *heap*.

Si el contingut d'un `String` ha de ser modificat sovint, val més utilitzar les classes `StringBuilder` o `StringBuffer`, que són mutables (l'objecte es crea en el *heap* general, i qualsevol canvi sobre l'objecte modificarà l'objecte, no en crearà un de nou), encara que no són tan potents en quant a mètodes com `String`.

```
String str = "Hola ", text = "Hola ";
for(int i = 0; i<100; i++) str += i;
```

Encara que, com hem comentat, el codi anterior no seria la millor opció a l'hora de treballar amb un text que és modificat tan sovint, el compilador de Java és prou intel·ligent per adonar-se'n en la majoria de casos i substituir internament l'`String` del bucle per un `StringBuilder` i, posteriorment, assignar el valor resultant en format de *String literal* a la variable original `str`.

4.1.1. `StringBuilder` enfront de `StringBuffer`

La principal diferència entre aquestes dues classes és que `StringBuilder` no és sincronitzada, mentre que `StringBuffer` sí que ho és. Això significa que podem utilitzar `StringBuffer` en programes amb més d'un fil d'execució,[23] és a dir, quan fem programació paral·lela. Per aquest motiu, el més habitual és utilitzar `StringBuilder` perquè, a més de la seva eficiència, en no ser sincronitzat, és millor que `StringBuffer`.

[23] En anglès, *thread*.

Com que ambdues classes són classes normals, només es poden crear a partir d'un constructor, no podem assignar-los directament un *String literal*. Tampoc no es pot utilitzar l'operador `+`, sinó els mètodes `append()` o `insert()`.

4.1.2. Resum

Si heu de modificar sovint un text, utilitzeu `StringBuilder` (en programes *single thread*) o `StringBuffer` (en programes *multi-thread*), encara que, com hem comentat, el compilador de Java és bastant intel·ligent per utilitzar internament `StringBuilder` quan fem servir `String` en situacions amb massa

modificacions. No obstant això, sempre que pugueu i prevegeu poques modificacions, utilitzeu `String` perquè és més eficient, ja que doneu al compilador la possibilitat d'optimitzar l'ús de memòria per part del programa mitjançant l'ús de l'*String common pool*.

4.2. Programació funcional (expressions lambda i Stream API)

La versió 8 de Java va introduir la programació funcional. D'aquesta manera, Java va posar de manifest que a partir d'aleshores no calia pensar en aquest llenguatge com a exclusivament orientat a objectes. La programació funcional guanya popularitat any rere any. Fins i tot hi ha tendències que mostren que comença a ser més popular que l'orientada a objectes.

4.2.1. Expressions lambda

La versió 8 de Java va venir acompanyada de les expressions lambda. Bàsicament, una expressió lambda és una funció anònima. Això vol dir que no requereix ser un mètode d'una classe i, per tant, no necessita un nom per ser invocada. La seva sintaxi és la següent:

```
(paràmetres) -> {cos-expressió-lambda}
```

Com es pot veure, hi ha tres parts:

1) Operador lambda, que està format pels símbols matemàtics menys i més gran que, és a dir, `->`.

2) La part esquerra, que fa referència als **paràmetres** de la funció. Si només hi ha un paràmetre, no cal posar els parèntesis. En el cas que no hi hagi paràmetres o n'hi hagi més d'un, llavors sí que són obligatoris. A més, Java pot deduir els tipus dels paràmetres que se'ls passa i, per tant, no cal indicar-los.

3) La part dreta, que fa referència al **cos**, és a dir, el codi en si de la funció. Si el cos només té una línia, no són necessàries ni les claus ni la sentència `return` en cas que retorni un valor.

Alguns exemples d'expressió lambda són:

```
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;
}
```

Les expressions lambda es basen en la programació funcional, que és un paradigma dins de la **programació declarativa** basada en l'ús de funcions matemàtiques. Això permet una programació més expressiva i elegant. La programació declarativa ens permet dir a un programa què és el que volem/necessitem, però sense dir-li com ho ha de dur a terme. Aquesta és la principal diferència amb la programació imperativa, en la qual s'indica al programa tots els passos que cal seguir per aconseguir el que volem. Per exemple, SQL (utilitzat per manipular bases de dades) és un llenguatge declaratiu:

```
SELECT * from users where surname="Anderson";
```

La sentència anterior, que en SQL s'anomena *query*, demana a la base de dades que retorni tots els usuaris de la taula `users` el cognom dels quals sigui `Anderson`. En aquest cas, no sabem si la base de dades internament farà servir un `for` o un `while` o un `if` o un `switch` o un *array* o una altra estructura. Per dir-ho així, nosaltres demanem el que volem i el programa es «busca la vida» per satisfer-nos.

4.2.2. Interfície funcional

Les expressions lambda estan íntimament lligades amb el concepte d'**interfície funcional**, concepte que també va ser afegit en Java 8. Una interfície funcional és igual que una interfície Java normal, però afegeix dues regles:

1) Té un sol mètode abstracte.
2) La resta de mètodes han de ser mètodes estàtics o `default`.

Encara que és opcional, en la declaració de la interfície funcional es pot afegir l'anotació `@FunctionalInterface` per fer que el codi sigui més fàcil d'entendre. Per exemple:

```
@FunctionalInterface
public interface Operacion{
       /*mètode abstracte que suma 2 números i el cos del qual s'implementarà
mitjançant una expressió lambda*/
       public void suma(int a, int b);
}
```

A continuació, utilitzem la interfície funcional que hem declarat:

```
public class Test{
       public static void main(String[] args){
       //Escrivim el codi del mètode abstracte de la interfície amb
//una expressió lambda
           Operacion op = (a,b) -> {System.out.println(a+b);};
       //Utilitzem el mètode amb la implementació
           op.suma(10,5);
        }
}
```

Així doncs, les interfícies funcionals es poden utilitzar per crear un tipus concret a partir de la definició d'una expressió lambda.

Un cop arribats a aquest punt pot semblar que moltes vegades haguem de crear una interfície funcional si volem fer alguna cosa complexa. Com hem vist, les expressions lambda poden treballar amb interfícies funcionals per aconseguir la reutilització del seu codi al llarg del nostre programa, però moltes vegades utilitzarem una expressió lambda en un context concret una sola vegada. De fet, Java té molts mètodes en les seves diferents classes que accepten com a paràmetre una expressió lambda. Per exemple, ara les classes que implementen la interfície `Iterable` tenen el mètode `forEach`. Algunes d'aquestes classes són `ArrayList`, `LinkedList`, `HashSet`, etc. El que fa aquest mètode és recórrer (o iterar) una estructura de dades i, per a cada element de l'estructura, executar l'operació (que serà una expressió lambda) passada per paràmetres. Així doncs:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(1);
lista.add(2);
lista.add(3);
lista.forEach(elemento -> System.out.println(elemento));
```

El codi anterior imprimeix els nombres que hi ha en `lista`.

Finalment, cal dir que les expressions lambda s'utilitzen majoritàriament amb la nova API `Stream` que va introduir JDK 8. De manera resumida, podem dir que, per mitjà de l'API `Stream`, podem treballar sobre col·leccions com si realitzéssim sentències SQL (gràcies a les expressions lambda). Això permet que treballem d'una manera neta i clara, evitant bucles i algorismes que alenteixen els programes i, fins i tot, fan que el codi es torni immanejable.

Enllaç recomanat

En l'enllaç https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html trobareu exemples de com utilitzar l'API Stream.

Hi ha tres parts que componen un `Stream`, que, de manera general, serien:

1) Un `Stream` funciona a partir d'una **llista o col·lecció**, que també es coneix com la font de la qual s'obtenen les dades.

2) Operacions intermèdies com, per exemple, el mètode `filter`, que permet fer una selecció de les dades a partir d'un predicat.

3) Operacions terminals com, per exemple, els mètodes `min`, `sum`, `forEach`, `max`, `findFirst`, etc.

Així doncs, una possible sentència amb `Stream` seria:

```
ArrayList<Empleado> empleados = new ArrayList<Empleado>();
empleados.stream().filter(empleado -> empleado.getSalary()>=1000).sum();
```

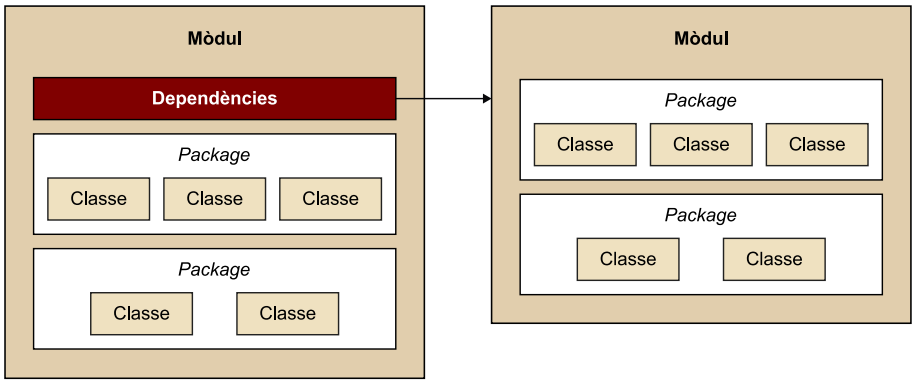
L'expressió anterior filtra els empleats que cobren `1000` € o més i en suma les quantitats. Així sabem quin és el cost salarial dels empleats que cobren `1000` € o més.

4.3. Mòduls

Si heu creat projectes amb Eclipse mitjançant una versió igual o superior a JDK 9, us deveu haver adonat que us pregunta si voleu crear un fitxer anomenat `module-info.java`. El fet és que JDK 9 va introduir el concepte de mòdul per millorar encara més l'encapsulació.

En Java, un **mòdul** és un conjunt de classes que poden contenir un o diversos `packages` i que defineix les dependències amb la resta de mòduls, i també la visibilitat de les classes que conté.

Figura 31



Per definir tot això, els mòduls utilitzen **descriptors** que s'escriuen en un fitxer anomenat `module-info.java` que està situat a l'arrel del codi font del mòdul. Fixeu-vos que el nom que té aquest fitxer trenca la regla de nomenclatura de Java, la qual no permet posar un guió com a nom de fitxer. D'aquesta manera, les eines (`java`, `javac`, IDE, etc.) no el confonen amb una classe Java normal.

Cada mòdul definit en el fitxer `module-info.java` té aquesta forma:

```
module nomModul{}
```

Es recomana que el nom del mòdul no coincideixi amb el d'una classe, interfície o paquet per tal d'evitar confusions.

Dins de les claus, podem utilitzar diferents sentències. Les dues més utilitzades són:

1) `exports` indica que les classes públiques del paquet exportat són visibles per a la resta del món.

```
module a{
    exports edu.uoc.logging;
}
```

Per a aquest exemple, només les classes públiques que estan dins del paquet `edu.uoc.logging` seran visibles fora del mòdul `a`. Així doncs, les classes públiques d'un altre paquet del mòdul `a`, per exemple, `edu.uoc.classroom`, no seran visibles fora del mòdul.

Per tant, veiem que, gràcies als mòduls, tot i que una classe sigui pública en un paquet, no ho serà per als elements de fora del mòdul si no s'exporta explícitament. Així doncs, els mòduls poden ocultar visibilitat pública, la qual cosa millora l'encapsulació en Java.

2) `requires` indica una dependència a un mòdul. És a dir, un mòdul necessita o vol utilitzar un altre mòdul.

```
module b{
    requires a;
}
```

4.4. Classes imbricades

En primer lloc, definirem què és una classe imbricada.[24]

[24] En anglès, *nested class*.

> Una **classe imbricada** és la que està dins d'una altra.

La possibilitat de crear classes imbricades es va incorporar en Java 1.1. Encara que són un tema avançat, són interessants ja que fan referència a les classes que s'utilitzen en un punt concret del nostre programa. A més, milloren l'encapsulació i la llegibilitat del codi.

Hi ha dos tipus de classes imbricades: les *static nested classes* i les *inner classes*. Vegem cadascuna d'elles.

4.4.1. *Static nested class*

Vegem un exemple per entendre el funcionament d'aquest tipus de classe:

```
public class OuterClass{
       private static String name = "David";

       public static class StaticNestedClass{
              public void doSomething(){
                     System.out.println("Hello! "+name); //Des d'una classe
//imbricada estàtica es pot accedir a qualsevol atribut estàtic de la classe
//externa.
              }
       }
}
```

La classe imbricada estàtica té accés directe a tots els membres estàtics de la classe externa. Si vol accedir a membres no estàtics, s'haurà d'utilitzar una referència a un objecte de la classe externa. Per aquest motiu, les classes imbricades estàtiques s'utilitzen poc.

Des de la classe externa no es pot accedir directament als membres de la classe imbricada estàtica. La classe imbricada estàtica és un membre de la classe externa i, per tant, podem assignar-li el modificador d'accés que ens sembli oportú.

Si volguéssim instanciar un objecte de la classe imbricada estàtica, hauríem de fer:

```
OuterClass.StaticNestedClass objeto = new OuterClass.StaticNestedClass();
```

4.4.2. *Inner class*

Una classe interna (*inner class*) és una classe imbricada no estàtica. El seu ús més habitual és proporcionar una classe que només s'utilitza dins de la seva classe externa. Es declara de la manera següent:

```
public class OuterClass{
       private static String name = "David";

       public class InnerClass{
       }
}
```

Per crear una instància de la classe interna, primer s'ha de crear l'objecte de la classe externa:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Des de la classe interna es pot accedir directament a qualsevol membre de la classe 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 classe interna és un membre de la classe externa i, per tant, podem assignar-li el modificador d'accés que ens sembli oportú.

Dins de la categoria *inner classes*, hi ha dues variants: les classes internes locals i les classes internes anònimes. Deixem a les vostres mans l'aprofundiment en aquestes dues variants.

4.5. Anotacions

Una **anotació** és un element que proporciona informació extra (metadades) sobre un element del nostre codi. La seva presència no té un efecte directe en la lògica del nostre codi.

Els usos més habituals de les anotacions són:

- Proporcionar informació al compilador: gràcies a l'ús d'una anotació, el compilador pot ajudar-nos a detectar errors o a suprimir avisos (*warnings*) que ja coneixem.

- Algunes eines són capaces de processar la informació proporcionada per les anotacions per generar codi, fitxers, etc.

- Algunes anotacions estan disponibles en temps d'execució.

El format bàsic d'una anotació és el símbol `@` seguit del nom de l'anotació, per exemple, `@Override`.

L'anotació pot ser simplement el seu nom (sense elements addicionals):

```
@Override
public void doSomething(){
        //TODO
}
```

O incloure elements:

```
@SuppressWarnings(
        value = "unchecked"
)
public void doSomething(){
        //TODO
}
```

Podem assignar a un mateix element més d'una anotació:

```
@SuppressWarnings(
        value = "unchecked"
)
@Override
public void doSomething(){
        //TODO
}
```

Java inclou un conjunt predefinit d'anotacions que estan situades en els *packages* `java.lang` i `java.lang.annotation`. Per exemple:

Anotació	Descripció
`@Deprecated`	Indica que l'element (és a dir, el mètode, la classe o l'atribut) marcat no ha de ser utilitzat mai més. El compilador genera un avís.
`@Override`	Avisa el compilador que el mètode amb aquesta anotació és una sobreescriptura d'un mètode d'una superclasse. No és obligatori fer servir aquesta anotació, però sí que ens ajuda a l'hora de detectar errors, ja que el compilador verificarà que es faci la sobreescriptura correctament.
`@SuppressWarning`	Demana al compilador que no avisi de segons quins *warnings*.
`@SafeVarargs`	Només es pot utilitzar amb mètodes amb *varargs*. Serveix perquè no es facin operacions insegures sobre el paràmetre *varargs*.
`@FunctionalInterface`	Indica que la interfície sigui considerada funcional.

4.5.1. Creació d'una anotació personalitzada

Per crear la nostra pròpia anotació, hem de crear un fitxer `.java` amb el nom de l'anotació i utilitzar-hi la sintaxi següent:

```
@interface nombreAnotacion{
        //Elements
}
```

Un exemple:

```
@interface Article{
        String[] authors();
        String title();
        int currentVersion() default 1;
        String lastModified();
        String[] reviewers();
}
```

Com podem veure:

- Utilitzem l'anotació `@interface` per crear una anotació.
- Cada element és un mètode sense paràmetres ni *throws*.
- Els tipus de retorn dels mètodes són tipus primitius, `String`, `Class`, `enum`, anotacions i *arrays* dels tipus anteriors.
- Els mètodes poden tenir un valor per defecte. L'indiquem amb la paraula clau `default`.

En la creació de la nostra anotació, podem afegir algunes anotacions que venen en el *package* `java.lang.annotation`.

Anotació	Descripció
`@Retention`	Indica quan es pot accedir a l'anotació.
`@Documented`	Indica que la informació de l'anotació ha de ser afegida en generar documentació amb `javadoc`.
`@Target`	Especifica el tipus d'element al qual s'associarà l'anotació.
`@Inherited`	Indica que l'anotació serà heretada automàticament.
`@Repeatable`	Indica que l'anotació pot utilitzar-se més d'una vegada per a un mateix element.

Vegem-ne un exemple:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface InfoMethod{
        String[] authors();
        int currentVersion() default 1;
        String lastModified();
        String[] reviewers();
}
```

L'anotació `@InfoMethod` que acabem de definir només es pot aplicar a mètodes i s'hi pot accedir en temps d'execució.

4.5.2. Ús d'una anotació personalitzada

Ara veurem com utilitzar l'anotació `@InfoMethod` creada en l'apartat anterior:

```
@InfoMethod(
       authors = {"David García", "Carlos Caballero"},
       lastModified = "03/03/2020",
       reviewers = {"Jordina", "Marta", "Sergio", "Romualdo", "Fran"}
)
public void doSomething(){
       //TODO
}
```

Hem de fixar-nos que quan utilitzem una anotació, els seus elements se situen dins de parèntesis.

4.6. Mètode `requireNonNull`

Amb JDK 7 es va afegir a la classe `Objects` un mètode estàtic anomenat `requireNonNull` que internament fa el següent:

```
public static <T> T requireNonNull(T obj){
    if(obj == null)
        throw new NullPointerException();

    return obj;
}
```

És a dir, comprova que l'objecte passat per paràmetres no sigui `null`. Si ho és, llança una excepció de tipus `NullPointerException`. En cas que no sigui `null`, retorna l'objecte en si.

Hi ha una sobrecàrrega del mètode que permet indicar-li el missatge d'error que cal mostrar si es llança l'excepció de tipus `NullPointerException`.

Així doncs, el codi següent:

```
public void doSomething(String name){
    if(name == null)
        throw new NullPointerException("Name cannot be null");
    this.name = name;
}
```

Pot ser simplificat mitjançant `requireNonNull`:

```
public void doSomething(String name){
    this.name = Objects.requireNonNull(name,"Name cannot be null");
}
```

5. Extres

5.1. Javadoc

En els arxius Java es poden posar comentaris en un format especial que permet, mitjançant l'ordre `javadoc`, generar de manera automàtica documentació dels paquets, de les classes, de les interfícies, dels atributs i dels mètodes del nostre projecte. La sintaxi de Javadoc és:

```
/**
* Text de la descripció (s'hi poden afegir etiquetes HTML)
*
* A partir d'aquí, etiquetes Javadoc que comencen per @
*
*/
```

En el cas dels comentaris normals (és a dir, `//` i `/* */`), Eclipse els acoloreix amb verd. No obstant això, els comentaris en format Javadoc els acoloreix amb blau. Així podem distingir visualment entre comentaris normals i de Javadoc.

Per a les classes se'n sol fer una breu descripció i utilitzar les etiquetes `@author` i `@version`. La primera indica el nom de l'autor que ha fet la classe (si n'hi ha més d'un, es poden fer servir tantes etiquetes `@author` com autors). Per la seva banda, la segona etiqueta informa de la versió de la classe (una classe pot ser modificada en el temps i, en un moment concret, es pot considerar que és una nova versió).

En el cas dels mètodes (inclosos els constructors), la primera part del comentari Javadoc és una descripció del que fa el mètode (de manera breu), i les línies següents consisteixen en un conjunt d'etiquetes Javadoc. Les etiquetes més utilitzades en els mètodes són:

Etiqueta	Descripció
`@param`	Descripció d'un paràmetre. Cada paràmetre té la seva etiqueta `@param`.
`@return`	Descripció del que retorna el mètode si el retorn no és `void`.
`@throws`	Descripció de l'excepció que pot propagar. Hi haurà una etiqueta `throws` per cada tipus d'excepció.
`@see`	Enllaç a la documentació d'una altra classe, que pot ser de Java, del mateix projecte o d'un altre.
`@since`	Indica des de quan existeix aquest mètode. Pot ser qualsevol text, però normalment és el nombre de versió de la classe.

Etiqueta	Descripció
`@deprecated`	Marca el mètode com a obsolet. Només es manté per compatibilitat i, per tant, encara es pot utilitzar. És possible que en futures versions de la classe el mètode desaparegui definitivament i no es pugui fer servir.

En els atributs d'una classe, simplement se'n posa la descripció amb `/** */` just abans de la seva declaració. Vegem-ne un exemple:

```
/**
    * This class represents an Example Class
    * @author David Garcia Solorzano
    * @version 1.0
    */
public class MyClass{

    /**
    * MyClass' name, e.g. Jerez de la Frontera.
    */
    private String name;

    /**
    * It is name's setter.
    * @param name MyClass's name.
    * @throws Exception When param "name" is longer than 40 characters.
    */
    public void setName(String name) throws Exception {
        if(name.length()>40) {
            throw new Exception("ERROR");
        }
        this.name = name;
    }
}
```

Un cop escrits els comentaris, hi ha diverses maneres de crear la documentació. Vegem-ne dues.

5.1.1. Generar `javadoc` per línia d'ordres

Podem crear la documentació per línia d'ordres, és a dir, sense fer servir cap IDE com Eclipse. Per a això, hem d'obrir una consola d'ordres (`cmd`) i situar-nos on tinguem els fitxers `.java`. A partir d'aquí podem:

1) Generar la documentació d'una classe concreta (per exemple, `Program.java`):

```
>> javadoc Program.java
```

2) O, el que és més habitual, generar la documentació de tots els `.java` que hi hagi en el directori:

```
>> javadoc *.java
```

És a dir, d'aquesta segona manera generarem la documentació de tota l'aplicació (o programa).

En qualsevol de les dues versions podem indicar que ens generi tota la documentació en un directori específic (si no existeix, el crea) mitjançant l'ordre `-d nomDirectori`:

```
>> javadoc *.java -d doc
```

Un cop executada correctament l'ordre `javadoc`, només hem d'obrir l'arxiu `index.html` amb un navegador web per veure la documentació.

Per acabar, cal indicar que quan es crida l'ordre `javadoc` per línia d'ordres, només genera la documentació per als atributs i mètodes que són públics i protegits. Si volem que inclogui també els privats, l'hi hem d'indicar de la manera següent:

```
>> javadoc *.java -d doc -private
```

Donar informació sobre els atributs i mètodes privats és útil si compartim informació amb algú que ha de programar la classe per dins. En cas contrari, no s'han de donar detalls dels atributs i mètodes privats! Aquesta és la gràcia de l'encapsulació!

5.1.2. Generar `javadoc` amb Eclipse

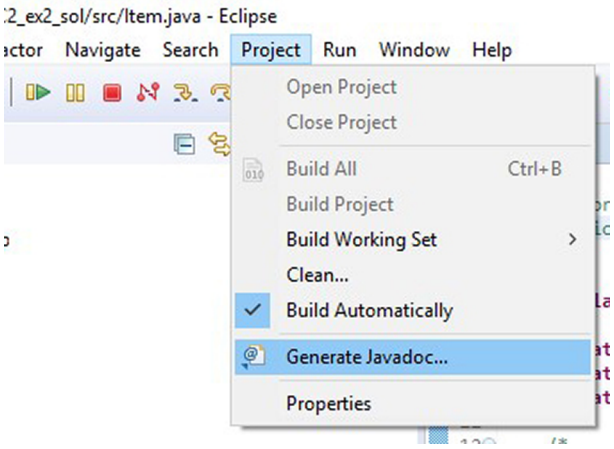
Per generar la documentació amb `javadoc` a Eclipse, hem de seguir els passos següents:

Nota

És important seleccionar el projecte, ja que si tenim seleccionada una classe, només ens farà el Javadoc d'aquesta classe.

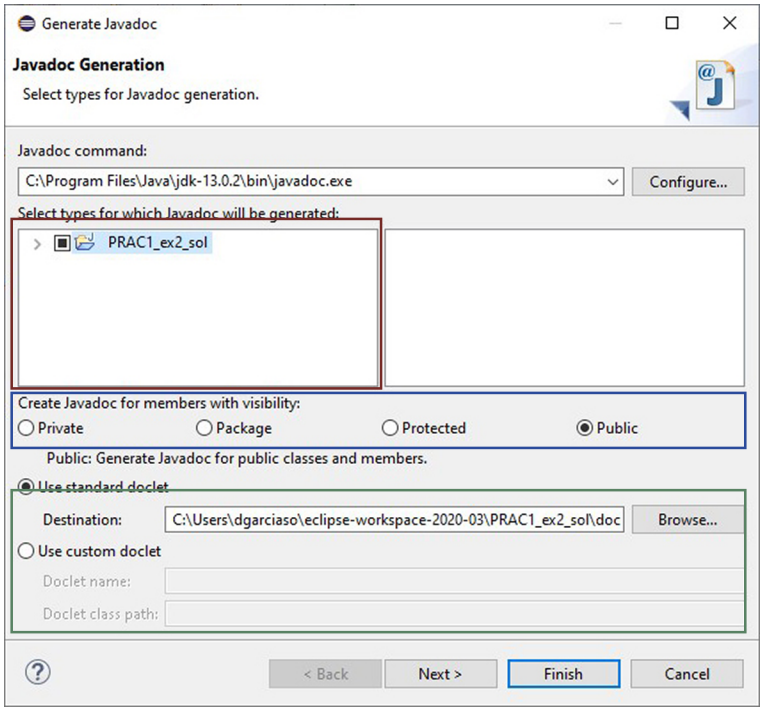
1) Anar al menú superior d'Eclipse i seleccionar «Project → Generate Javadoc. . . ».

Figura 32



2) Ens apareixerà una finestra com la que es mostra en la figura 33.

Figura 33



3) Si el camp «Javadoc command» està buit, hem d'indicar, en prémer el botó «Configure...», la ruta del nostre `javadoc.exe`, que es troba a la carpeta del nostre JDK (per exemple, `C:\Program Files\Java\jdk-13.0.2\bin\javadoc.exe`).

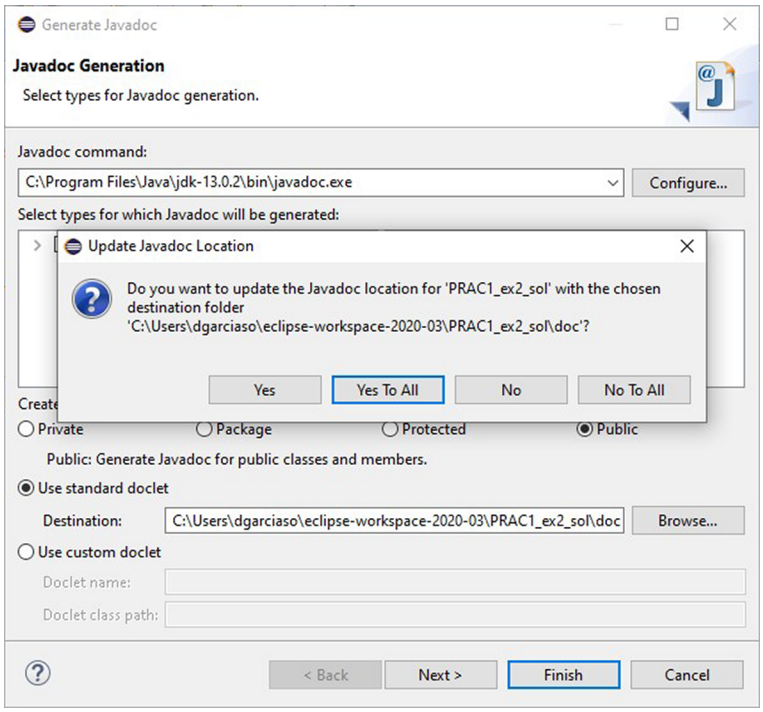
4) Indiquem per a quins projectes volem generar la documentació (vegeu requadre granat en la figura 33) i en quina carpeta volem desar la documentació generada (vegeu requadre verd en la figura 33).

5) Eclipse permet indicar per a quins atributs i mètodes es genera la documentació (vegeu requadre blau de la figura 33).

6) Una vegada ho tenim tot configurat, premem el botó «Finish». Òbviament, podem clicar el botó «Next» per personalitzar el comportament de `javadoc`.

7) Si deixem l'opció estàndard, és possible que la primera vegada ens aparegui la finestra de diàleg que es mostra en la figura següent. Cliquem «Yes To All».

Figura 34



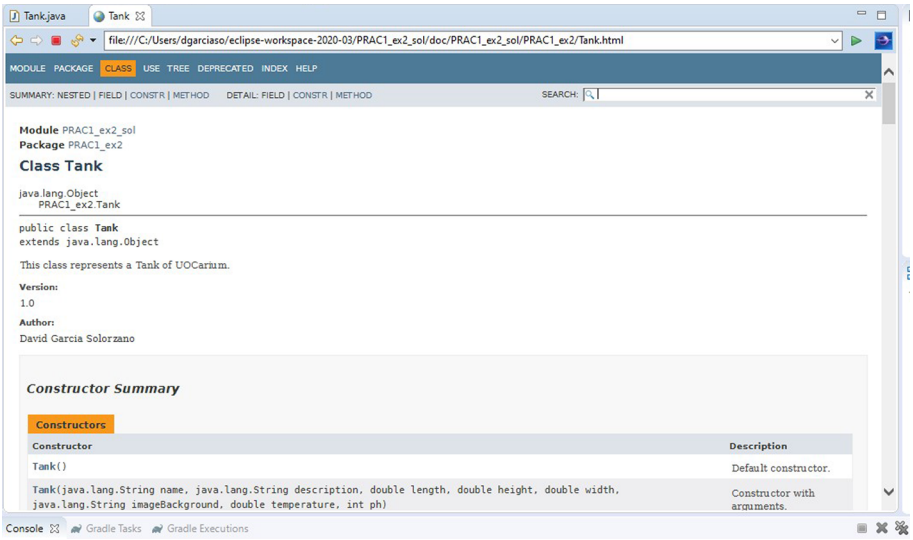
8) Si fem «F5» (o botó dret «Refresh») sobre el projecte (no sempre fa falta), veurem que ha aparegut una nova carpeta anomenada `doc`. Aquesta carpeta conté diversos fitxers `.html`, que és el format en què Javadoc genera la documentació.

9) Per veure correctament la documentació generada, hem d'anar a la carpeta `doc` a Eclipse, clicar amb el botó dret el fitxer `index.html` i escollir l'opció «Open → With Web browser». A la web que apareixerà, triem l'opció «Package» del menú superior i després escollim qualsevol classe per veure'n els detalls. També podem obrir aquest fitxer amb el navegador web des del *workspace*, és a dir, sense fer servir Eclipse.

Com podem imaginar, Eclipse fa per nosaltres les crides a l'ordre `javadoc` explicades a l'apartat «Generar javadoc per línia d'ordres» segons la configuració que li hem indicat.

A continuació, podem veure com es mostra la documentació generada amb Javadoc a Eclipse per a una classe anomenada `Tank`.

Figura 35



5.2. JavaFX

5.2.1. Introducció

Les **interfícies gràfiques** (GUI)[25] són un element bàsic avui dia. Java té tres biblioteques per fer interfícies gràfiques: **AWT**, **Swing** i **JavaFX**. Resumint molt, es podria dir que Swing és més avançat que AWT i que JavaFX ha arribat per ser el substitut de Swing. Fins fa relativament poc, Swing era la llibreria predominant, però JavaFX comença a guanyar-li terreny. No obstant això, en un programa podeu combinar components (és a dir, botó, camp de text, desplegable, etc.) d'ambdues llibreries, encara que no és gaire recomanable. Oracle no ha donat per obsolet Swing, però sembla que no li incorporarà millores en el futur. És per això que en aquesta guia ens centrarem en JavaFX. Així mateix, amb JDK 11, Oracle ha decidit treure del nucli del JDK la llibreria JavaFX perquè aquesta sigui independent i tingui el seu propi ritme de desenvolupament, cosa que n'allibera l'evolució. Això, evidentment, comporta alguns canvis, especialment a l'hora de configurar l'entorn de treball (per exemple, Eclipse).

[25] Acrònim de *graphical user interface*, en anglès.

Swing i JavaFX comparteixen moltes coses, com ara l'ús de components (també coneguts com a *widgets*), però tenen coses totalment diferents com, per exemple, la manera de treballar, a més de les seves capacitats.

5.2.2. Configuració de JavaFX a Eclipse IDE

En aquest apartat explicarem, pas a pas, com configurar JavaFX a Eclipse.

1) Cal assegurar-se de tenir instal·lat JDK 11 o superior i, a més, que el tinguem establert com a compilador per fer servir a Eclipse (o en el projecte en el qual vulguem utilitzar JavaFX).

2) Com que ara JavaFX està fora del nucli de JDK, cal anar a la pàgina web següent: https://gluonhq.com/products/javafx/.

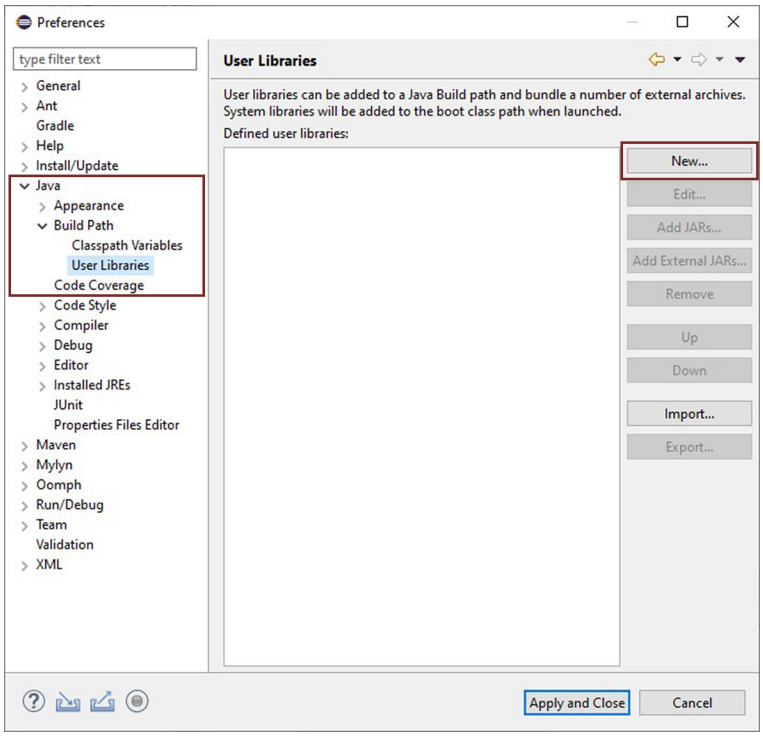
3) En aquesta pàgina web, instal·larem **JavaFX 11** (Long Term Support) o versió posterior. Aquí presentem els passos per a Windows. Així doncs, descarreguem l'opció «JavaFX Windows SDK».

4) Descomprimim el fitxer on creguem convenient, per exemple, dins d'«Arxius de programa/Java».

5) Dins d'Eclipse, anem a l'opció «Window» del menú superior i escollim l'opció «Preferences».

6) S'obrirà una finestra i escollim en el menú de l'esquerra l'opció «Java → Build Path → User Libraries». Fem clic en el botó «New...».

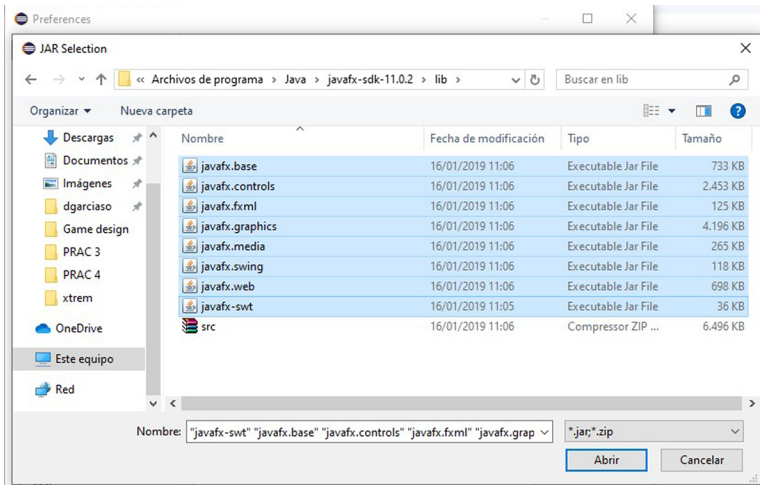
Figura 36



6) En la nova pantalla ens demana un nom per a la llibreria. Escrivim: «JavaFX11» i fem clic en el botó «OK».

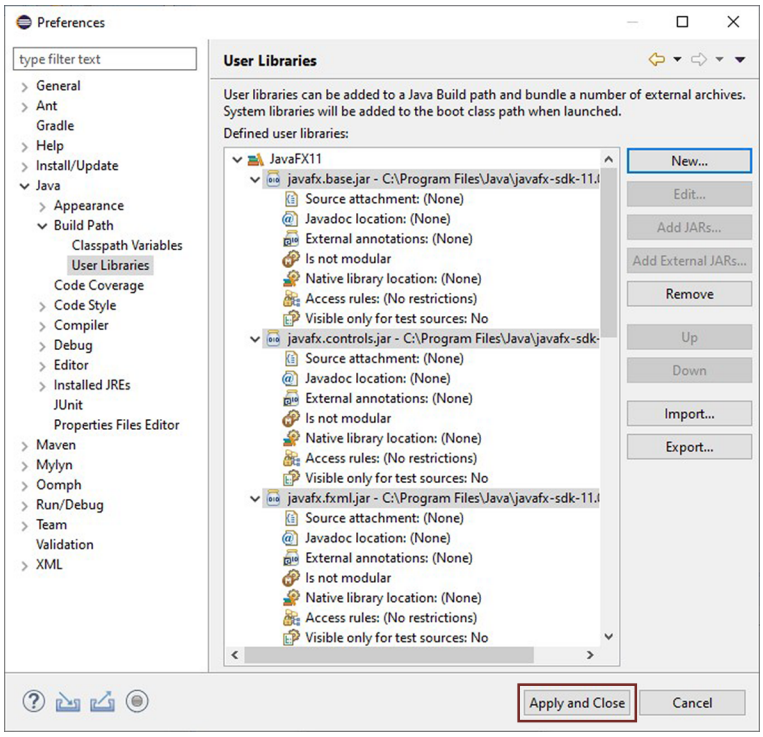
7) A continuació, fem clic en el botó «Add External JARs...», busquem la ruta fins al directori de JavaFX11 que hem descomprimit anteriorment. Hi ha tres subdirectoris. Cal entrar a «lib» i triar tots els fitxers «jar» i **no** seleccionar el fitxer «**src.zip**». Un cop seleccionats, cliquem «Obrir».

Figura 37



8) El resultat de fer els passos anteriors ha de ser el que es mostra en la figura 38. Si és així, fem clic en el botó «Apply and Close».

Figura 38



9) Després cliquem amb el botó dret el projecte i escollim l'opció «Properties». A la finestra que s'obrirà triem l'opció «Java Build Path» del menú esquerre. En la part central escollim la pestanya «Libraries» i, a continuació, «Module-path». Després cliquem «Add Library...» Tot seguit triem «User Libraries» de la finestra flotant, cliquem «Next», escollim «JavaFX11» i, per acabar, «Finish». Després cliquem «Apply and Close».

10) Perquè Eclipse, és a dir JDK, detecti la llibreria JavaFX, cal anar important els paquets que utilitzem de JavaFX en el fitxer especial `module-info.java`. Si no fem servir aquest fitxer per a res, el més senzill és fer el següent: anem al fitxer `module-info.java` i comentem el codi:

Vegeu també

S'ha parlat del fitxer especial `module-info.java` en l'apartat «Mòduls» d'aquesta guia.

```
/*
module nomModul{
}
*/
```

11) En aquest punt, els errors relacionats amb l'ús de classes de la llibreria JavaFX haurien de desaparèixer.

12) No obstant això, encara no podem executar correctament l'aplicació, ja que cal configurar l'execució. Per a això, anem a «Run configurations → Arguments» i escrivim a «VM arguments» el que es mostra a continuació.

```
--module-path "C:\Program Files\Java\javafx-sdk-11.0.2\lib" --add-modules
javafx.controls,javafx.fxml
```

El que està ressaltat amb vermell s'ha de canviar per la ruta en la qual s'ha descomprimit JavaFX.

13) Ara anem al menú superior «Help → Install New Software...». En el camp «Work with» de la finestra que apareixerà escrivim l'adreça: https://download.eclipse.org/efxclipse/updates-released/3.5.0/site/ i premem «enter» perquè Eclipse busqui paquets en aquesta adreça. Una vegada acabada la cerca, marquem tant «e(fx)clipse – install» com «e(fx)clipse – single components».

14) Cliquem el botó «Next». Apareixerà una altra finestra en la qual també hem de fer clic en el botó «Next».

15) A continuació, apareixerà una pantalla perquè acceptem la llicència. Marquem «I accept the terms...» i premem «Finish».

16) Eclipse instal·larà els paquets. Quan acabi és possible que ens demani reiniciar l'IDE. Si és així, ho fem.

17) A partir d'ara, podrem crear projectes que utilitzin JavaFX. En fer «File → New → Other» ens apareixerà una carpeta anomenada «JavaFX» amb plantilles.

5.2.3. Model en JavaFX

Un dels avantatges de JavaFX és que utilitza el patró MVC (Model-Vista-Controlador) per crear els programes, la qual cosa és molt interessant. En aquesta guia (i assignatura) no treballarem la part «model» a l'estil JavaFX, és a dir,

per treure partit a tot el potencial de JavaFX. Això us ho deixem a vosaltres, per si voleu ivestigar JavaFX. Si hi esteu interessats o interessades, busqueu els conceptes `properties` i `binding` de JavaFX.

5.2.4. Vistes en JavaFX (part visual)

Pel que fa a les vistes, cal assenyalar que, o bé es creen programant en Java, o bé es creen mitjançant uns fitxers anomenats FXML, que no deixen de ser uns fitxers que contenen un codi XML *ad hoc* per a JavaFX. La primera opció només s'ha d'utilitzar per crear components (o tota la vista si fos necessari) en temps d'execució (per exemple, mostrar un botó després de marcar un *checkbox*). La segona forma, la més habitual, s'assembla a altres tipus de desenvolupament, com ara la creació d'*apps* a Android (en què la part visual de la vista és un fitxer XML i la part d'interacció de la vista és una classe codificada en Java) o el *front-end* d'un web (la part visual feta amb HTML+CSS i la part interactiva que es comunica amb el controlador feta en Javascript).

Perquè es vegi més clar, penseu que cada pantalla/vista del vostre programa serà un fitxer FXML (part visual) més un fitxer Java (part interactiva i de comunicació amb el controlador). Gràcies a això s'aconsegueix separar/desacoblar la part visual (és a dir, la forma, l'estil i la ubicació d'un botó) de la lògica (és a dir, el funcionament/comportament del programa en fer clic en un botó). Així mateix, com que cada vista té un fitxer Java per a la part interactiva, des d'ell podrem afegir, mitjançant codi, components a la vista si calgués, combinant així totes dues maneres de creació de components (és a dir, via codi i via FXML).

Segur que penseu que deu ser molt laboriós fer una interfície si l'hem de fer escrivint codi FXML (que és un XML *ad hoc*). Teniu tota la raó, és ardu crear interfícies visuals mitjançant codi FXML i, a més, «a cegues». Per això hi ha programes que ajuden a realitzar els fitxers FXML. Un d'aquests programes és SceneBuilder.[26] SceneBuilder permet crear cada pantalla/vista del nostre programa de manera visual, és a dir, arrossegant i deixant anar components en l'escena. Una vegada tinguem la pantalla/vista amb l'aparença que vulguem, només hem de demanar a SceneBuilder que generi el fitxer FXML que codifica la pantalla/vista que hem creat. Així de simple!

(26) http://gluonhq.com/products/scene-builder/

Per instal·lar Scenebuilder, anem al web següent: https://gluonhq.com/products/scene-builder/#download i instal·lem SceneBuilder for Java 11 (serveix per a JDK 11 i superiors). Per a Windows recomanem utilitzar la versió instal·lador, no el `.jar`.

Podem treballar amb SceneBuilder en paral·lel amb Eclipse. És a dir, creem els fitxers FXML amb SceneBuilder i després els copiem en el nostre projecte d'Eclipse. Però, també podem fer una cosa millor, que SceneBuilder i Eclipse s'entenguin. És a dir, que des d'Eclipse creem un fitxer FXML nou, l'obrim i el

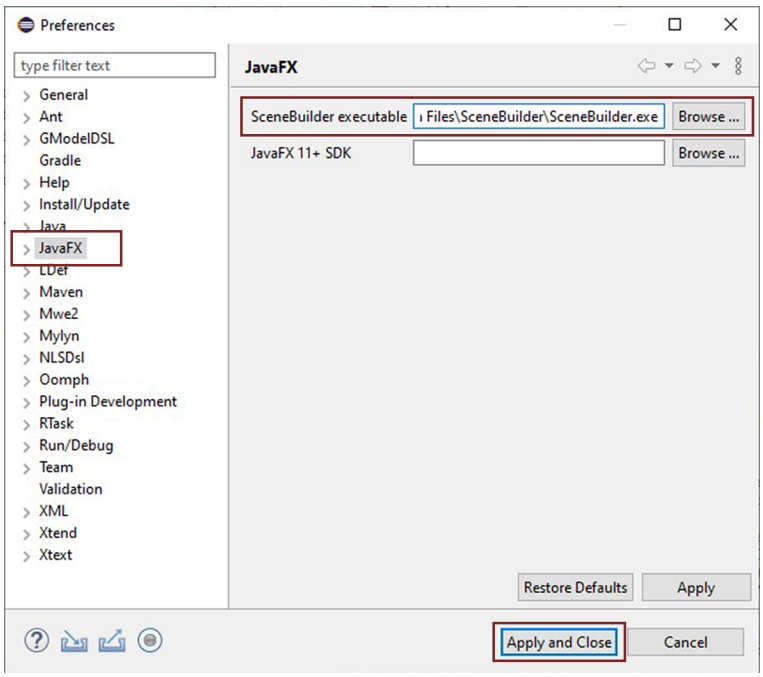
modifiquem amb SceneBuilder des d'Eclipse, i els canvis es vegin reflectits automàticament en el projecte. Per aconseguir que SceneBuilder i Eclipse treballin de manera col·laborativa, cal seguir els passos que detallem a continuació:

1) Dins d'Eclipse anem a l'opció «Window» del menú superior i escollim l'opció «Preferences».

2) A la finestra que apareixerà, busquem a la llista de l'esquerra l'opció «JavaFX».

3) La seleccionem. A la dreta hauria d'aparèixer un únic camp en el qual podem indicar on està situat l'executable de SceneBuilder (vegeu la figura 39). Ho indiquem i fem clic en el botó «Apply and Close».

Figura 39



Gràcies a aquesta configuració podrem clicar amb el botó dret qualsevol fitxer FXML a Eclipse i triar l'opció «Open with SceneBuilder» perquè s'obri aquest fitxer al programari d'edició SceneBuilder. A més, si desem els canvis a SceneBuilder, aquests seran reflectits en el nostre projecte Eclipse. De vegades, cal refrescar el projecte (per exemple, fent F5 a Eclipse). Si en comptes d'obrir un fitxer FXML amb SceneBuilder, l'obrim fent doble clic, veurem el codi FXML a l'editor d'Eclipse.

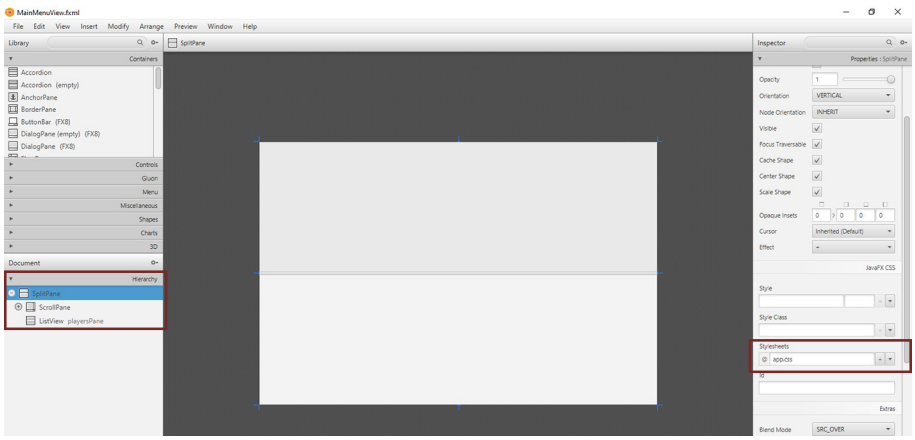
Les vistes, és a dir, cada FXML, poden ser personalitzades mitjançant un fitxer CSS (o fins i tot diversos FXML poden compartir un mateix fitxer CSS). Un fitxer CSS[27] és un document en el qual s'especifiquen diferents estils, des de generals fins a concrets (per exemple, per a un botó determinat). Aquests fitxers són essencials quan es programen webs, ja que faciliten l'escalabilitat, el manteniment i la gestió de possibles canvis. JavaFX fa servir CSS encara que

[27] Acrònim de *cascade style sheet*.

utilitza les seves pròpies propietats, malgrat que moltes són similars a les propietats web, per la qual cosa és senzill fer-les servir si es coneix CSS web. Això nosaltres no ho tractarem aquí.

Si en un fitxer FXML seleccionem a SceneBuilder un element per mitjà del lateral esquerra («Document → Hierarchy») i després mirem el panell «Properties» que hi ha a la dreta del programa (vegeu figura 40), dins trobarem un apartat anomenat «JavaFX CSS». Aquest apartat permet indicar estils directament (apartat «Style»), assignar una classe d'un fitxer CSS i afegir un fitxer CSS. Justament a l'apartat «Stylesheets», hem posat «app.css», que és el nom del fitxer CSS de l'aplicació. En el cas del color i la mida d'un text (per exemple, `label`) o un botó (per exemple, `button`), SceneBuilder ja té un apartat per a ells dins de «Properties» sense necessitat d'utilitzar propietats CSS.

Figura 40



5.2.5. Vistes a JavaFX (part interactiva)

Fins ara només hem vist com crear la part visual d'una interfície/vista/pantalla; però, com li donem vida? Com hi interactua l'usuari? Doncs bé, això es fa mitjançant **esdeveniments** (tant en JavaFX com en Swing i AWT). Això implica una nova forma de programar que s'anomena ***programació orientada a esdeveniments*** **(POE)**. En la POE, el flux d'execució del programa és definit per les accions/esdeveniments (per exemple, clic, doble clic, pressionar una tecla, etc.) realitzats per un agent extern al programa, per exemple, un usuari, un altre programa, etc. La POE no és incompatible amb la POO. De fet, amb Java, continuarem treballant amb objectes que són capaços de respondre a esdeveniments. Així doncs, POO i POE es complementen. Per exemple, tindrem un objecte de tipus `Button` que escoltarà (amb el que s'anomena un *Listener*) que ocorri un esdeveniment. Quan ocorri un esdeveniment concret, com pot ser fer clic en el botó, si té codificat un mètode associat a «fer clic», aquest mètode s'executarà en produir-se l'esdeveniment. És a dir el *Listener* invocarà el mètode associat.

Com ja hem dit, una pantalla/interfície/vista tindrà, d'una banda, el seu fitxer que defineix la part visual (és a dir, un fitxer FXML) i, d'altra banda, un fitxer Java que serà l'encarregat de gestionar els esdeveniments produïts sobre la vista (és a dir, fitxer FXML) i seguir la lògica del programa segons aquests esdeveniments. Normalment, els fitxers Java encarregats de la part interactiva d'un fitxer visual FXML, se solen anomenar igual que el fitxer FXML al qual «donen vida», però finalitzats amb l'afegitó `Controller`. Així sabem que el fitxer Java encarregat de la part interactiva d'un fitxer FXML es diu igual que el fitxer FXML però amb l'afegitó `Controller`. A partir d'ara, aquests fitxers els anomenarem controladors.

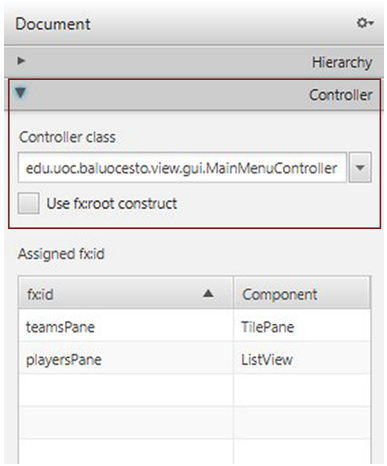
És important destacar que SceneBuilder només detecta/troba les classes (és a dir, fitxers Java) que hi ha al mateix paquet que el fitxer FXML que s'edita.

Com es pot intuir, és essencial vincular un controlador/part interactiva (fitxer Java) a una vista (fitxer FXML). Això es pot fer de dues maneres:

1) Editant directament el codi XML del fitxer FXML. Obrim el fitxer FXML amb Eclipse. En el tag/etiqueta arrel es pot afegir un camp/atribut anomenat «fx:controller», el valor del qual és el nom de la classe controladora (incloent-hi el paquet en el qual està situat).

2) Mitjançant SceneBuilder. Obrim el fitxer FXML amb SceneBuilder. Hi ha una opció anomenada «Controller» dins de l'opció «Document» del menú esquerre (vegeu figura 41). Hi ha un camp anomenat «Controller class», en el qual podem assignar la classe controladora. SceneBuilder detectarà totes les classes que estiguin en el mateix paquet que el fitxer FXML que s'edita. Per a cada FXML, hem d'escollir en el desplegable l'opció (és a dir, el controlador) que correspongui.

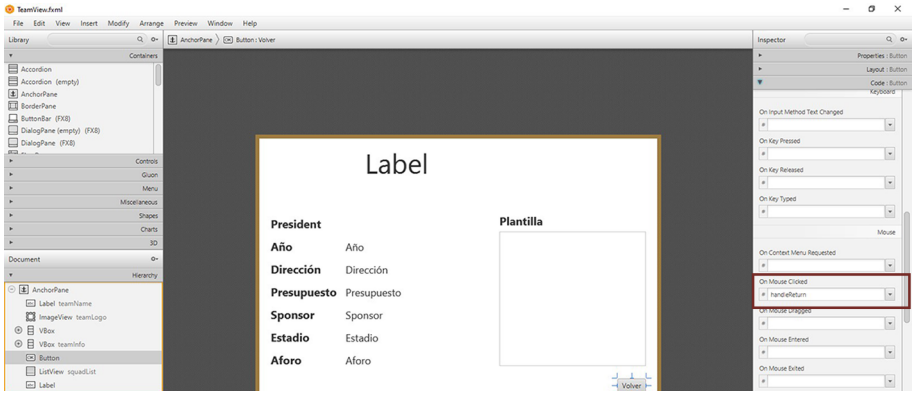
Figura 41



A la classe controladora, crearem tots els mètodes i atributs que volem que interactuïn amb la part visual de la vista. És una bona pràctica precedir tots els atributs i mètodes del controlador que vulguem que siguin accessibles des de les vistes amb l'anotació `@FXML`. Els atributs i mètodes precedits amb `@FMXL`, encara que siguin privats, seran visibles per a la vista en què utilitzem el controlador que els conté. Si un atribut o mètode és públic, no cal fer servir `@FXML`, però sí que és recomanable.

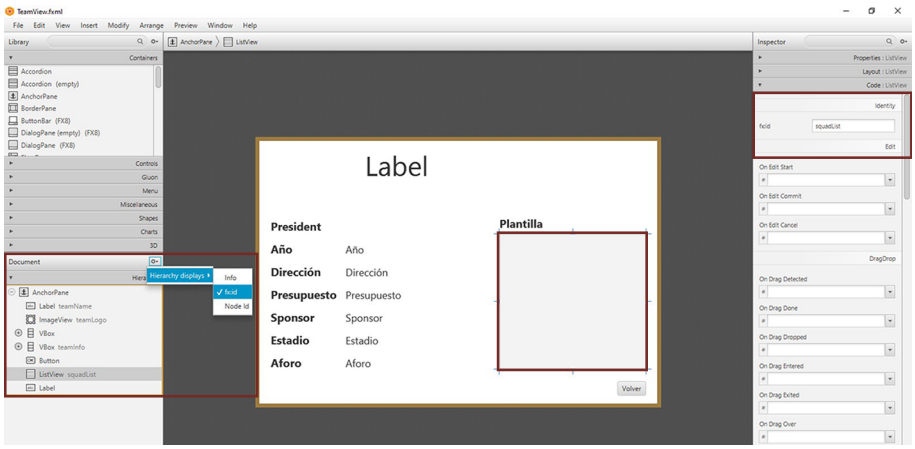
Imagineu que creem un mètode anomenat `handleReturn` que serveix per tornar a la vista/pantalla anterior a la qual estem. En aquest moment, hem de vincular un element de la interfície amb el mètode `handleReturn`. Per fer aquesta assignació/vinculació, el més còmode és fer servir SceneBuilder. Obrim el fitxer FXML amb SceneBuilder, seleccionem un `Button` que hagueu posat a la interfície i, a l'apartat «Code» del lateral dret, busquem l'opció «On Mouse Released» (vegeu figura 42). Hi hem escrit/seleccionat el nom de mètode que cal executar, en aquest cas, `handleReturn`. Aquesta assignació també la podeu veure reflectida, òbviament, en el codi FXML.

Figura 42



Finalment, si volem que un element de la interfície sigui accessible des d'un controlador (és a dir, codi Java), hem d'assignar-li un identificador. Per a això, a SceneBuilder seleccionem un dels elements de la interfície gràfica, i en el menú dret, en l'opció «Code», trobarem un camp anomenat «fx:id». El més habitual és posar-li com a nom un que també fem servir com a atribut en el controlador (precedit d'`@FXML`). D'aquesta manera, el controlador pot accedir a aquest component per modificar-lo o consultar-lo, i la vista, a la informació de l'atribut de la classe de tipus controladora. O dit d'una altra manera, l'atribut de la classe controladora fa referència al component de la vista perquè tenen el mateix nom i perquè a la classe controladora hem posat `@FXML` davant de l'atribut.

Figura 43



En l'exemple que podem veure en la figura anterior, el nom que hem donat a la `ListView` és `squadList`. En el codi Java del controlador tindrem un atribut anomenat `squadList` de tipus `ListView` i precedit per l'anotació `@FXML`.

5.3. JUnit 5

La llibreria JUnit permet als desenvolupadors fer tests unitaris. En aquest apartat, veurem què és un test unitari i com utilitzar JUnit amb Eclipse.

5.3.1. Test unitari

Abans de res, definirem què és un test unitari.

> Els **tests unitaris** (*unit tests*) són les proves de nivell més baix en la programació. Amb elles es comprova el funcionament de les unitats lògiques independents, normalment els mètodes. Són proves que es duen a terme de manera molt ràpida i les que els desenvolupadors passen constantment per verificar que el seu programari funciona adequadament.

Aquests són els tests més importants. Cada test unitari és un pas que caminem en el camí de la implementació correcta del programari. Els desenvolupadors utilitzen els tests unitaris per assegurar-se que el codi funciona com esperen que funcioni, de la mateixa manera que el client fa servir els tests d'acceptació/client per assegurar-se que els requisits de negoci es compleixin amb el programari com s'espera que ho faci. Tot test unitari ha de ser:

- **Atòmic**. Significa que el test unitari prova la mínima quantitat de funcionalitat possible. És a dir, provarà un únic comportament d'un mètode d'una classe. Com ja sabeu, un mètode pot presentar diferents respostes davant diferents entrades o d'un context diferent. El test unitari s'ocuparà exclusivament d'un d'aquests comportaments, és a dir, d'un únic camí d'execució.

- **Independent**. Significa que un test no pot dependre d'altres per produir un resultat satisfactori. No pot ser part d'una seqüència de test que s'hagi d'executar en un determinat ordre. Ha de funcionar sempre igual independentment del fet que s'executin altres tests o no.

- **Innocu**. Aquesta propietat significa que no altera l'estat del sistema. En executar-lo una vegada, produeix exactament el mateix resultat que en executar-lo vint vegades. A més, no altera la base de dades, ni envia missatges electrònics, ni esborra fitxers, ni els crea (i si els ha de crear, els ha d'esborrar). Un test, una vegada executat, ha de comportar-se com si no s'hagués executat.

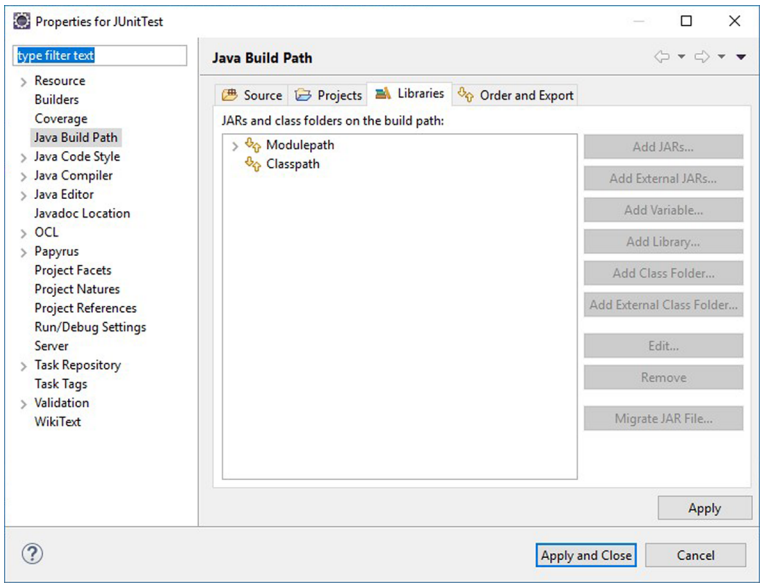
- **Ràpid**. Atès que executem un gran nombre de tests cada pocs minuts, resultaria molt poc productiu haver d'esperar uns quants segons cada vegada. Un únic test ha d'executar-se en una petita fracció de segon.

5.3.2. Configuració del *plugin* JUnit

En instal·lar Eclipse IDE, es realitza la instal·lació de JUnit, una biblioteca que ve per defecte a Eclipse. No obstant això, a continuació s'il·lustren els passos necessaris per configurar un entorn de treball amb proves. Alguns aspectes o noms poden canviar entre el que s'explica en aquest tutorial i el que us podeu trobar en el vostre entorn de treball. En aquest cas, heu de ser capaços de fer les analogies pertinents.

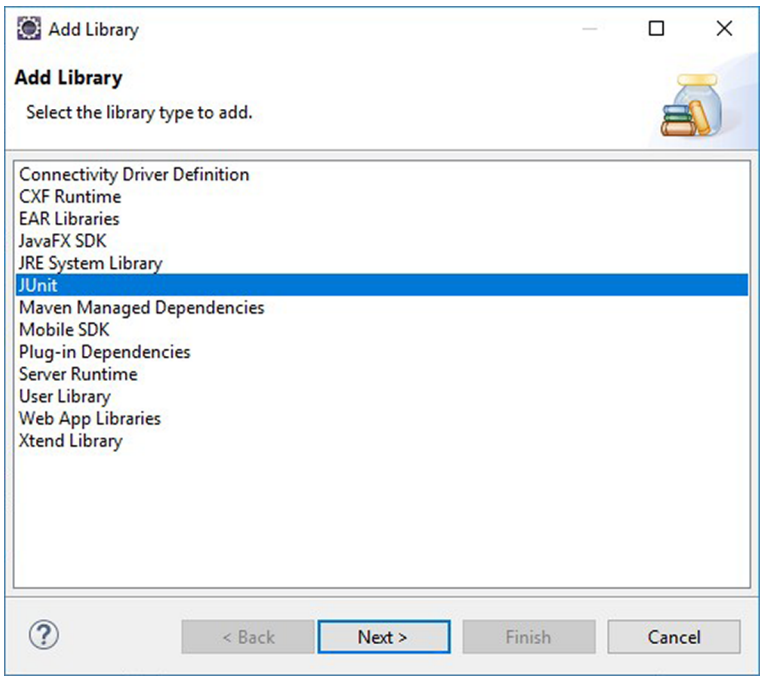
1) Imaginem que tenim un projecte ja creat anomenat «JUnitTest» que està buit, és a dir, no conté cap fitxer `.java`. Si ara volem indicar que aquest projecte ha de permetre JUnit, hem de clicar amb el botó dret el nom del projecte en el «Package Explorer» i escollir «Properties». A la finestra que ens apareixerà hem d'anar a l'opció «Java Build Path» del menú de l'esquerra i, tot seguit, escollir la pestanya «Libraries» a la zona central i seleccionar «Classpath». A continuació, fem clic en el botó «Add Library» (vegeu figura 44).

Figura 44



2) A la finestra que ens apareixerà seleccionem «JUnit» de la llista i premem «Next» (vegeu figura 45).

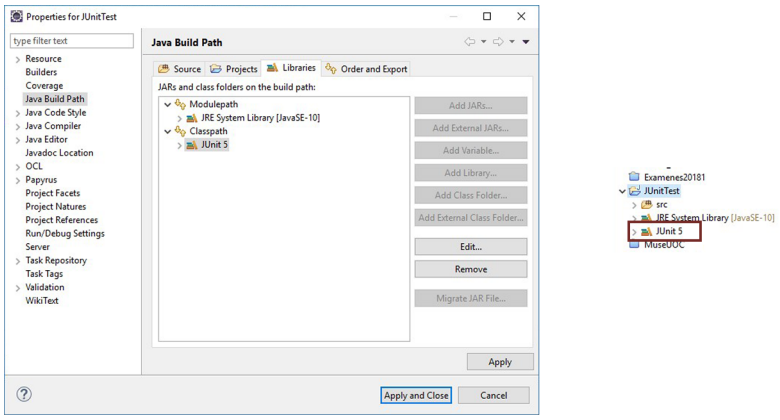
Figura 45



3) En la pantalla següent escollim la versió més alta de JUnit[28] i diem «Finish» (vegeu figura 46). A continuació, veiem la pantalla de la figura 44 amb JUnit afegida. Premem «Apply and Close». Ara, en el «Package Explorer» veiem que s'ha afegit la llibreria JUnit 5. A partir d'aquí ja podem utilitzar JUnit per fer tests unitaris.

[28]En el moment de la creació d'aquesta guia, versió 5.

Figura 46

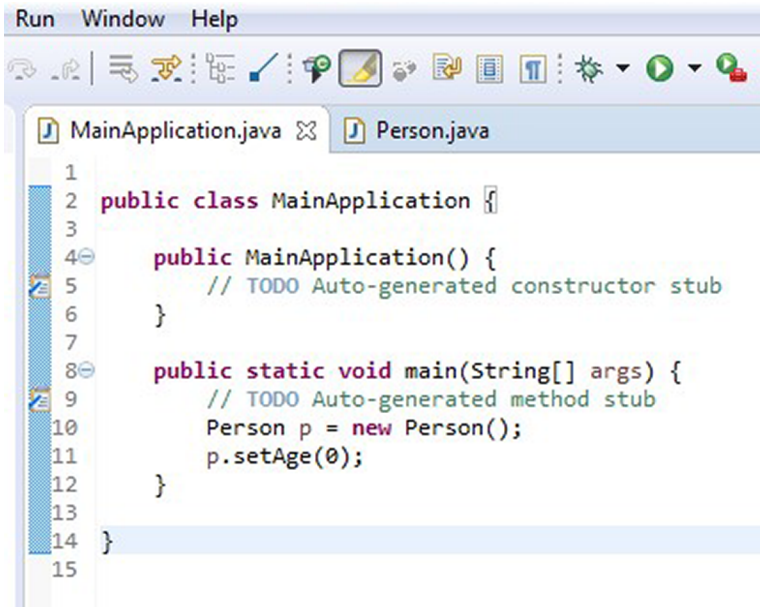


5.3.3. Utilitzar JUnit

Veurem un exemple dins del projecte «JUnitTest» esmentat a l'apartat anterior.

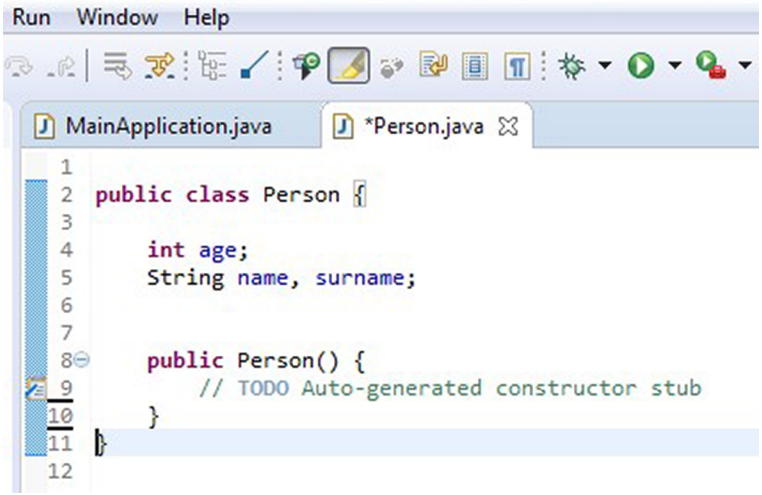
1) Ara afegim una nova classe anomenada `MainApplication` que contindrà un mètode `main`. El codi del fitxer `MainApplication.java` és el que es mostra en la figura 60.

Figura 47



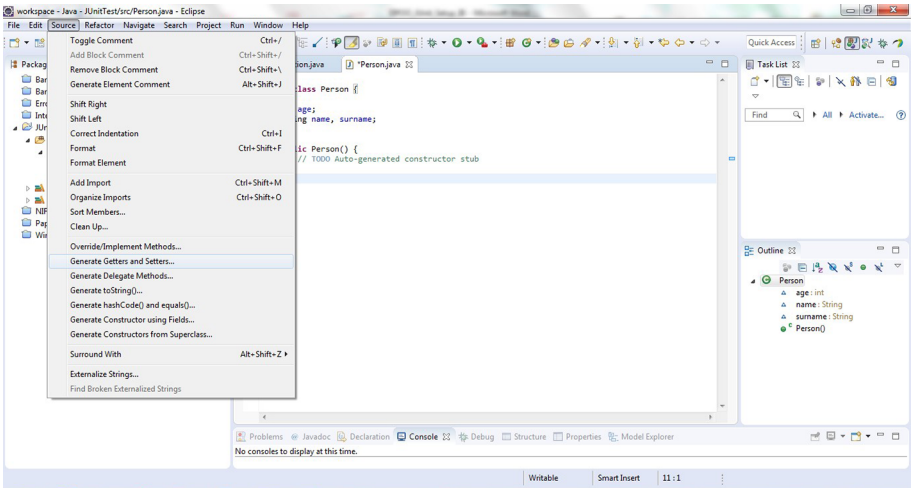
2) Ara afegim una nova classe anomenada `Person`. Una vegada creat el fitxer `Person.java`, escrivim els atributs, com es mostra en la figura 48.

Figura 48



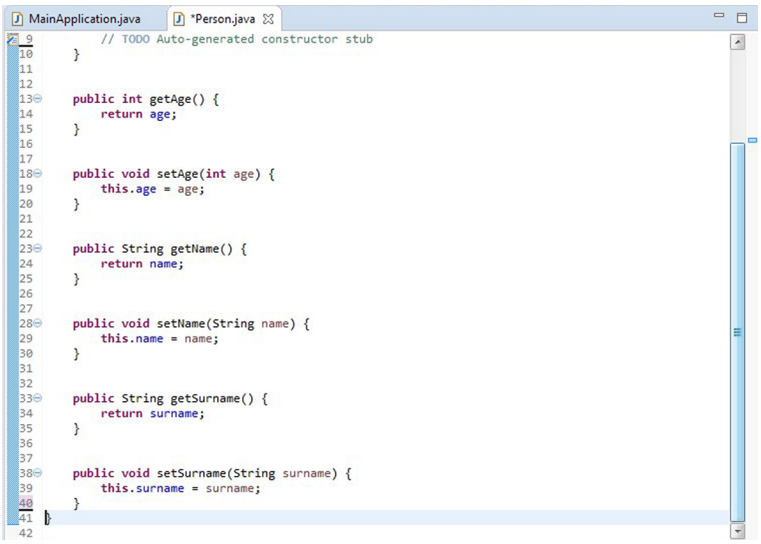
3) Per generar els *getters* i *setters* de la classe `Person` farem servir una drecera d'Eclipse. Amb el fitxer `Person.java` activat en l'editor de codi, anem a l'opció «Source» del menú superior d'Eclipse i seleccionem l'opció «Generate Getters and Setters...» (vegeu figura 49). A la finestra que ens apareixerà marquem tots els atributs i premem «OK». Ràpid, oi que sí?

Figura 49



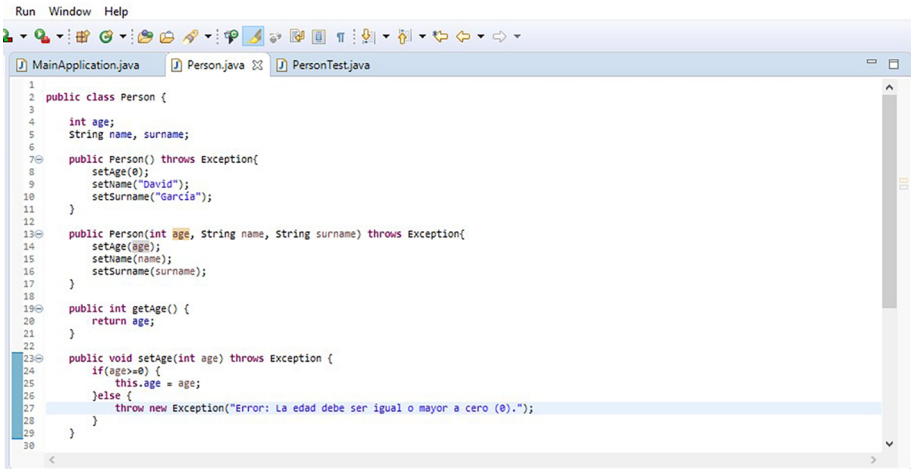
4) El resultat és el que es mostra en la figura 50.

Figura 50



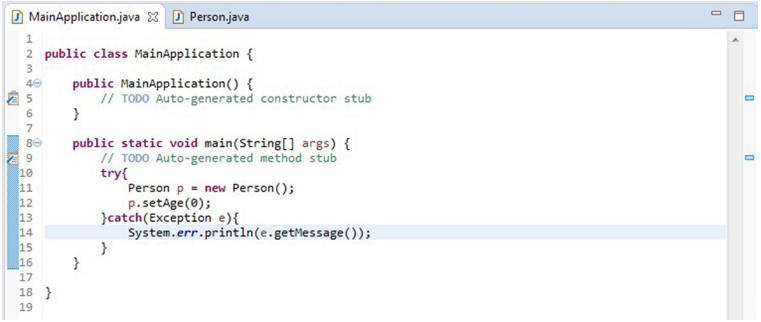
5) Afegim codi en el constructor per defecte, afegim un constructor amb arguments i canviem el codi de `setAge` tal com es mostra en la figura 51.

Figura 51



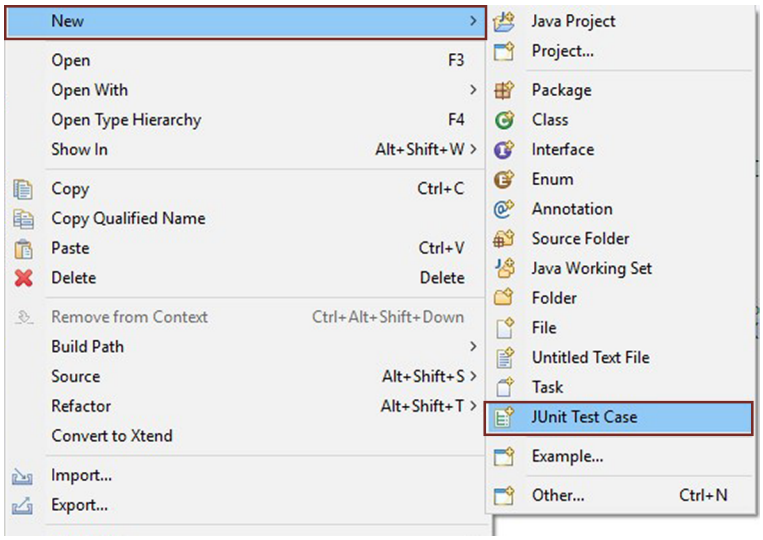
6) També canviem el codi de `MainApplication` perquè capturi l'excepció de `Person` (vegeu la figura 52).

Figura 52



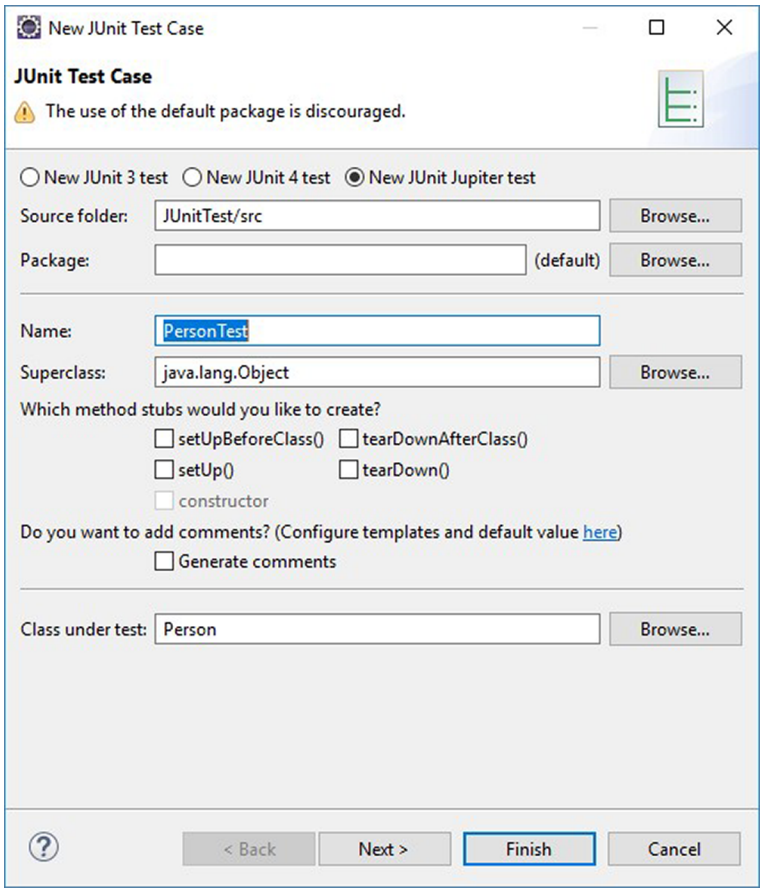
7) Ara, a la finestra «Package Explorer», cliquem amb el botó dret del ratolí `Person.java` i fem «New → JUnit Test Case» (vegeu figura 53).

Figura 53



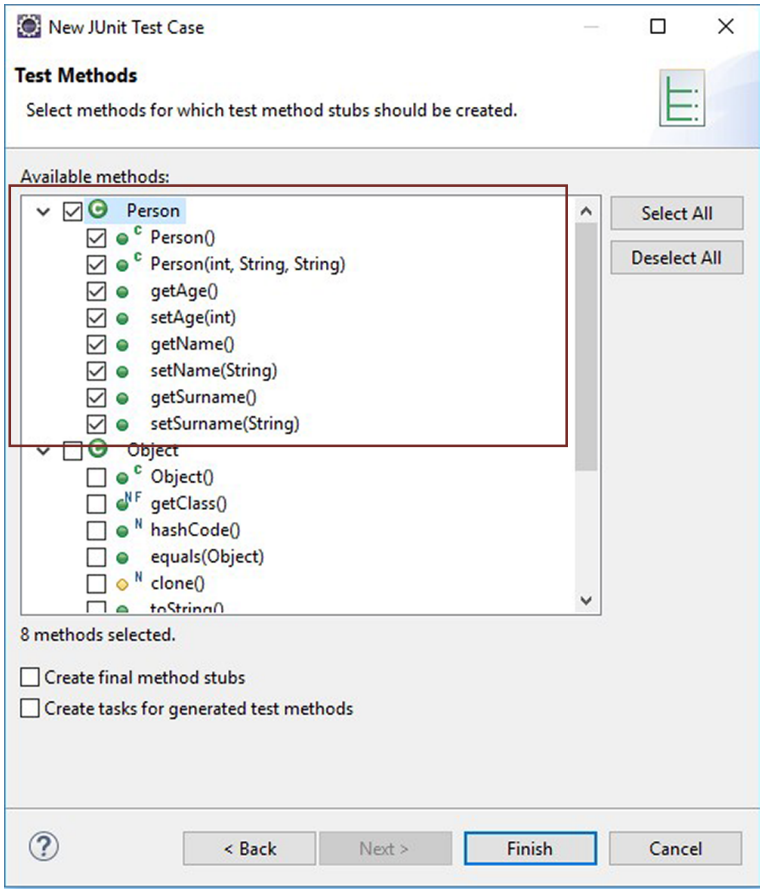
8) D'aquesta manera, ens sol·licita els mètodes del codi pels quals es vol generar la plantilla de proves (vegeu figura 54). En aquesta finestra fem clic a «Next» (l'opció «New JUnit Jupiter Test» és el mateix que dir «JUnit 5»).

Figura 54



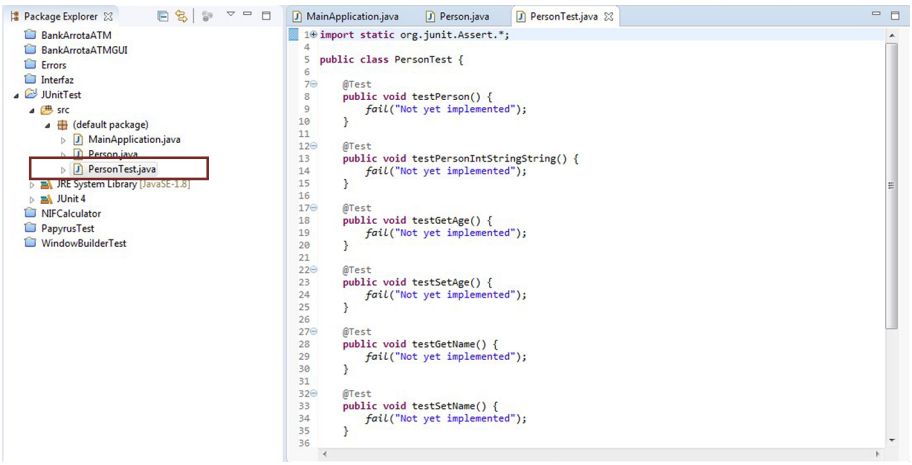
9) A la finestra que ens apareixerà seleccionem els mètodes que volem provar. Per exemple, seleccionem la classe `Person` per seleccionar així, d'una vegada, tots els seus mètodes (vegeu figura 55). Un cop ho tinguem, fem clic en el botó «Finish».

Figura 55



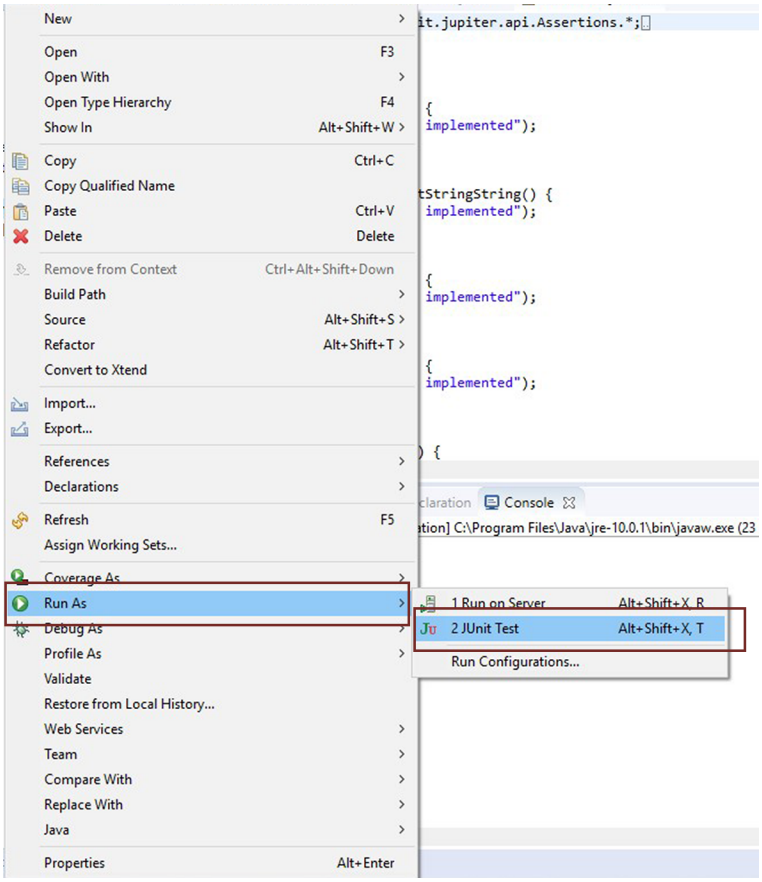
10) Ara, en el «Package Explorer» ens ha aparegut una nova classe anomenada `PersonTest.java` (vegeu figura 56) amb l'avís d'error. Una manera de treure els errors és eliminant el fitxer `module-info.java`. Si mirem el codi de `PersonTest.java`, veurem que són proves relacionades 1: 1 amb els mètodes de la classe `Person` i que, inicialment, seran fallides per recordar-nos que han de ser implementades. De fet, surt el text `"Not yet implemented!"`. Podrem afegir més mètodes de prova anteposant l'anotació `@Test` davant de la signatura d'aquest mètode. Gràcies a `@Test`, aquest mètode és considerat prova unitària. Fixeu-vos que si un mètode està sobrecarregat, com pot ser el constructor, per diferenciar-lo li afegeix com a afegitó els tipus dels arguments que rep. Per exemple, `testPersonIntStringString` és el mètode de prova que crea JUnit per al constructor amb arguments, `Person (int age, String name, String surname)`.

Figura 56



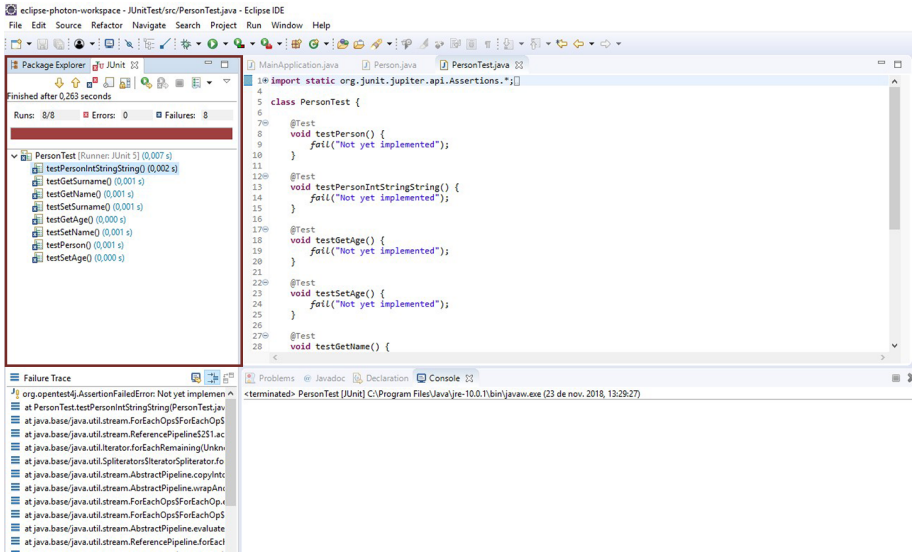
11) A continuació, s'han d'executar les proves per comprovar que NO s'han superat (la qual cosa és lògica ara mateix). En aquest cas, s'ha de marcar el fitxer de test (en el nostre exemple, `PersonTest.java`) i amb el botó dret del ratolí prémer «Run As → JUnit Test» (vegeu figura 57).

Figura 57



12) Ara ens apareix una pantalla que ens mostra els tests que NO s'han superat i el nombre de tests que han fallat. En aquest cas, hem configurat vuit tests per a vuit mètodes dels quals volíem comprovar la funcionalitat i tots són fallits (vegeu figura 58). És possible que en Windows 10 us salti una pantalla d'avís del Firewall que haureu d'acceptar.

Figura 58



13) Ara codificarem el mètode `testSetAge` (vegeu figura 59).

14) Si mirem el codi de la figura 59, podem veure que el primer pas consisteix a canviar la signatura del mètode perquè llanci/propagui les excepcions que rep el seu codi, en aquest cas, el mètode `setAge` llança `Exception`. Ja en el cos del mètode, el primer és crear un objecte que disposi del mètode que es vol comprovar. Per tant, es crea un objecte de la classe `Person`.

```
Person instance = new Person();
```

Una vegada es disposa d'un objecte `Person` vàlid, es comença a invocar els mètodes que es volen comprovar per comprovar que els valors de què disposa són correctes. Observem que sobre un objecte (`instance`) de la classe `Person` s'invoca el mètode `setAge` (que es vol provar). En aquest cas, s'ha invocat amb un valor de deu anys.

```
instance.setAge(10);
```

La nostra persona, per tant, hauria de tenir una edat de deu anys. Per comprovar que és així, el *framework* JUnit aporta mètodes especials anomenats *asserts* que ens han de retornar valors booleans (`true` o `false`) per poder comprovar que els mètodes funcionen adequadament. Per exemple, es fa ús del mètode `assertEquals`, el qual comprova que el valor `10` és igual que el valor obtingut per `instance.getAge()`.

Més endavant, es vol comprovar si el mètode `setAge` llança l'excepció que té programada quan el valor introduït és negatiu. Per especificar aquesta operació a JUnit 5 es fa ús del nou mètode `assertThrows` que retorna un objecte de tipus `Exception` el missatge del qual, com ja hem vist, es pot comparar amb el mètode `assertEquals`.

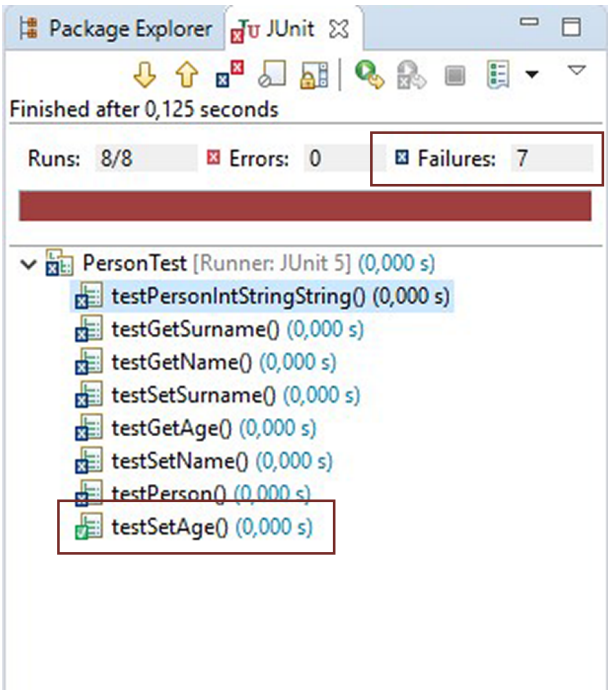
```
Exception exception = assertThrows(Exception.class, () -> instance.setAge(-1));
assertEquals("Error: La edad debe ser igual o mayor a cero (0).", exception.getMessage());
```

Com podem observar, el primer paràmetre d'`assertThrows` és el tipus d'excepció que s'espera. En aquest exemple, esperem la classe `Exception.class`. En cas que es disposi d'una classe específica per a excepcions (per exemple, `PersonException`), n'hi hauria prou d'especificar-hi la classe de l'excepció que s'espera (per exemple, `PersonException.class`). El segon paràmetre és una expressió lambda. Finalment, en la segona sentència, l'excepció retornada per `assertThrows` és comparada mitjançant el seu missatge (`getMessage`) amb el missatge esperat. Perquè això funcioni, cal afegir el corresponent `import`:

```
import static org.junit.jupiter.api.Assertions.*;
```

15) Si tornem a executar «Run As → JUnit Test», veurem que ara només tenim set tests fallits, en comptes de vuit. A la llista, `testSetAge` apareix amb un vist/*check* verd, la qual cosa significa que ha passat la prova correctament.

Figura 60



16) Si fem clic en algun dels mètodes fallits a la finestra JUnit (figura 58), a la part inferior anomenada «Failure Trace», ens indica on ha fallat aquest mètode/test. Quan tots els mètodes de test de la classe `PersonTest` superin les proves, la franja vermella es tornarà verda i tots els mètodes tindran el vistiplau/*check* verd al costat. A més, hi dirà `Runs: 8/8, Errors: 0, Failures: 0`. Gràcies a l'ús dels test sabem que les classes es comporten com s'espera que ho facin. Si anem codificant i passem els tests i un que no fallava ara falla, podrem detectar-ho a temps.

Bibliografia

Java Tutorials Learning Paths [en línia] [consulta: 3 de setembre de 2020]. Disponible a: https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html.

Niemeyer, Patrick; Knudsen, Jonathan (2002). *Learning Java (Java Series)* (2a ed.). Estats Units: O'Reilly Media, Inc.

Niemeyer, Patrick; Knudsen, Jonathan (2013). *Learning Java (Java Series)* (4a ed.). Estats Units: O'Reilly Media, Inc.

Sierra, Kathy; Bates, Bert (2009). *Head First Java. A Brain-Friendly Guide* (2a ed.). Estats Units: O'Reilly Media, Inc.

Van der Linden, Peter (2004). *Just Java 2* (6a ed.). Estats Units: Prentice Hall, PTG.