Breu introducció a Java

Fernando García, Joan Bartrina

http://deic.uab.cat/~jbartrina

Dpt. d'Enginyeria de la Informació i de les Comunicacions (dEIC)
Universitat Autònoma de Barcelona

Setembre 2008

Índex

1 Introducció

Aquest document ha estat obtingut aplicant lleus modificacions al document *Introducció a Java* creat pel Sergi Robles i pel David Gavilán. Els professors de pràctiques de *teoria de la informació* volem agrair la seva gentilesa per haver-nos facilitat les fonts de manera desinteresada.

Breu introducció a Java no preten esdevenir un manual d'aprenentatge i/o consulta del llenguatge de programació Java. El seu principal objectiu és servir de material de suport per a la pràctica de *teoria de la informació*. Apart de trobar tota la informació sobre Java que us cal per tal d'implementar l'algorisme de la codificació aritmètica tal i com es demana a l'enunciat de la pràctica, també podreu trobar una breu descripció de l'arquitecuta de Java, del llenguatge de programació Java i de les eines de desenvolupament.

Com hem dit abans, totes les seccions presentes diferents aspectes de Java, però no totes elles són necessàries pel desenvolupament de la pràctica. Els punts 3 (llevat de 3.2.5) i 4 contenen la informació bàsica per realitzar la pràctica.

2 L'arquitectura Java

El nom "Java" és una marca registrada de *Sun Microsystems*, i es refereix al llenguatge de programació desenvolupat per aquesta empresa i publicat al 1995 en les seves primeres versions. Java s'utilitza per a crear continguts executables susceptibles de ser distribuïts a través de xarxes. De manera més genèrica, la denominació Java fa al·lusió a un conjunt d'eines de programari/maquinari dissenyades per a crear i implementar uns continguts executables emprant el llenguatge de programació Java.

2.1 Conceptes generals

Molta gent associa **Java** amb una eina per a poder crear pàgines interactives o amb animacions per a la *World Wide Web*. Si bé és cert que amb Java es poden fer aquestes coses, aquest llenguatge va molt més enllà. Java és un llenguatge de programació de propòsit general, totalment orientat a objecte, simple, complert, segur i multi-*thread*, indicat per a construir una gran varietat d'aplicacions. Java incorpora mecanismes per facilitar les comunicacions i la programació distribuïda, resultant idoni per a programar en entorns de xarxa com ara Internet.

Un dels objectius en el disseny de Java era aconseguir un llenguatge de programació amb una portabilitat total. Quan es parla de programació d'aplicatius destinats ja no només a una petita comunitat, sinó a tota *Internet*, cal tenir en compte que les diferències culturals dels usuaris i dels programadors podria arribar a ser un greu problema. Java va incorporar des del principi la internacionalització del programari. Això es va aconseguir utilitzant un estàndard de codificació de caràcters que contingués a la resta: l'UNICODE.

L'objectiu principal, però, era assegurar la portabilitat a nivell de codi executable. És a dir, el programa que ha fet un programador i ha compilat a la seva plataforma local, ha de córrer en totes les plataformes d'igual manera, sense haver de recompilar el codi font. Els programes executables Java es podrien aleshores transmetre per xarxa al igual que ara ho fan les imatges o el text. Aquest ambiciós objectiu pretén revolucionar la forma en la que s'escriu i es distribueix el programari. Java podria ser la manera de compartir tots un únic llenguatge.

El lema de Java és "*Write once, run everywhere*". De cada programa hi hauria d'haver un únic codi font i un únic codi executable. Evidentment, les avantatges d'un llenguatge així serien espectaculars: les companyies de programari no haurien de dedicar-se a diferents plataformes per separat, la programació seria molt més senzilla donat que s'hauria aïllat definitivament el programari del maquinari, tots els usuaris podrien utilitzar tot el programari, etc.

2.2 La Màquina Virtual de Java

Per aconseguir la transparència programari/maquinari que es necessitava, la solució triada va ser dissenyar una arquitectura neutra. Cada plataforma ha de tenir, llavors, una interfície entre la seva pròpia arquitectura i aquesta arquitectura Java. El programari s'executarà realment sobre aquesta nova arquitectura (figura 1).

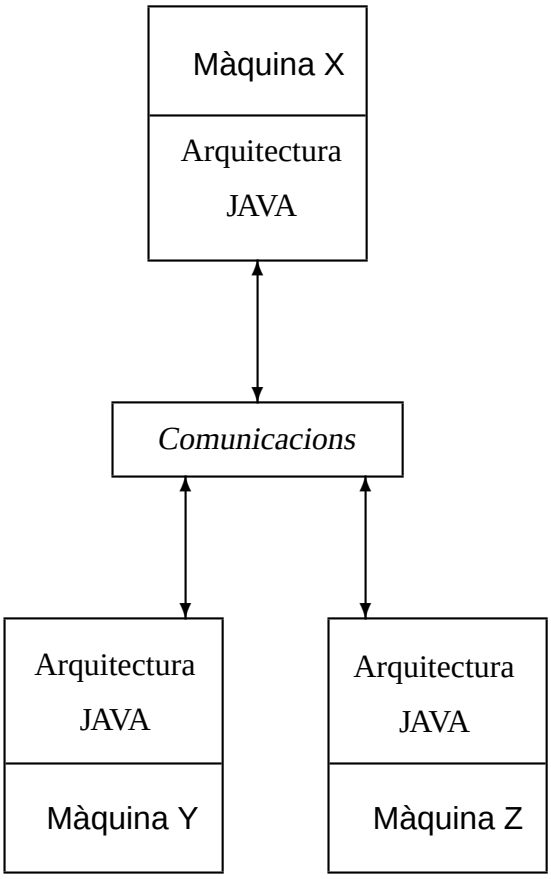

Figura 1: Esquema de la idea de Java

Així va néixer el que es coneix com la Màquina Virtual de Java (*Java Virtual Machine* o JVM), una màquina abstracta, no existent físicament, perfectament definida i que hauria d'estar present a totes les màquines "reals". El codi "executable" Java passaria a la Màquina Virtual en lloc d'anar directament a la CPU local. Allà s'aniria traduint (interpretant) de codi Java a codi natiu de la màquina a on seria finalment executat.

Per tant, tindrem un codi intermedi amb el que estaran construïdes les aplicacions, anomenat **codi bytecode**, i un intèrpret a cada plataforma, la **màquina virtual**. Cada programador necessitarà un compilador que faci *codi bytecode* a partir de codi font Java a la seva màquina i d'aquesta manera podrà assegurar la portabilitat.

El disseny d'aquesta arquitectura era la clau de l'èxit de Java. Per aconseguir la ubicuitat s'ha d'especificar de quina manera es comportarà exactament l'entorn en temps d'execució de Java.

D'una banda ha de ser de molt baix nivell, per tal de que la màquina anfitriona del programa Java el pugui "traduir" al seu propi llenguatge màquina de manera ràpida i eficient; però no tant baix com per tenir un conjunt d'instruccions massa gran que dificulti la transmissió per xarxa.

Aquesta màquina virtual executa un conjunt especial d'"instruccions", anomenades **bytecodes**, que són simplement una sèrie de bytes amb format. Cadascun d'aquests conté una especificació precisa de l'efecte que exerceix sobre la màquina virtual. La màquina virtual s'encarrega també de determinades funcions fonamentals de Java, com ara la creació d'objectes o el retorn de recursos.

Java havia de ser un llenguatge orientat a *Internet*, per tant segur en quant a la gestió dels recursos de la màquina quan són usats per codi provinent de fora de l'àmbit local. Per aquest motiu la seguretat a Java va ser dissenyada des de la mateixa màquina virtual. Per transportar els bytecodes de forma segura a través de la xarxa, és necessari un model de seguretat a tota prova, així com un format precís per enviar aquesta sèrie de bytecodes d'una màquina virtual a una altra.

3 El llenguatge de programació Java

Alguns programadors utilitzen Java com un llenguatge de propòsit general, no només quan volen assegurar la portabilitat del codi o utilitzar alguna altra particularitat de l'arquitectura Java. Java, el llenguatge, ofereix una gran facilitat per programar (reutilització de codi, seguretat d'accés a memòria, ...), resultant molt fàcil, per tant, generar ràpidament codi depurat. Incorpora mecanismes per evitar els errors més comuns en altres llenguatges, com la no reserva explícita de memòria (*garbage collection*), les referències de tipus segur (cast controlat) o les excepcions.

Encara que les eines que porta integrades Java són molt potents, el llenguatge en si resulta molt senzill i no és difícil assolir en poc temps un grau alt d'habilitat programant.

3.1 Un llenguatge Orientat a Objecte

Java és un llenguatge orientat a objecte. L'objectiu d'aquesta secció no és explicar qué és un llenguatge orientat a objecte, sinó donar les característiques específiques de Java.

3.1.1 Classes

Els programes Java estan construïts a partir de *classes*. A partir de cada classe es generarà un fitxer de codi *bytecode*. Normalment aquests fitxers tenen l'extensió *.class*. A partir de la definició d'una classe, podem crear qualsevol nombre d'*objectes*, coneguts com *instàncies* de la classe.

A Java, totes les classes són subordinades de la classe *object*, és a dir, una classe no pot començar mai una nova jerarquia, sempre tindrà com a superclasse la classe *object*.

La sintaxi de la declaració de classes a Java és la següent:

```
class Identificador {
  CosClasse
}
```

L'*Identificador* indica el nom d'una nova classe, que derivarà predeterminadament d'*object*. En el cos de la classe aniran els *membres*. Existeixen dos tipus de membres en les classes de Java, els *mètodes* i els *camps* (*variables membre*). Els camps es defineixen dins de la classe tal i com es fa a C (*structs*) o a $C++$. Els mètodes serien els equivalents a les funcions d'altres llenguatges, però lligades a

una certa classe. A diferència d'altres llenguatges, Java no permet la declaració de 'funcions' fora d'una classe. En el següent exemple veiem la declaració de la classe *Persona* amb dos camps, *nom* i *edat* i un mètode *identifica* que oferirà les dades de la persona.

```
class Persona {

  private String nom;    // nom de la persona
  private int    edat;   // edat de la persona

  public Persona(String n, int e) {
    nom = new String( n );
    edat = e;
  }

  public String identifica() {
    return new String ( nom + new Integer(edat).toString() );
  }

}
```

Noteu la similitud amb *JavaScript*. Tal com veieu al retorn del mètode *identifica*, s'utilitza un operador sobrecarregat: '+'. A Java podem sobrecarregar mètodes, però no podem sobrecarregar operadors com es fa a $C++$, per exemple. Hi ha alguns operadors, com el '+', que Java ja proporciona sobrecarregats.

3.1.2 Herència

Java té un model d'herència simple, és a dir, no suporta l'herència múltiple directa. A Java, una nova classe pot estendre exactament una classe com a molt. Estendre una classe no significa només heretar la seva interfície, sinó també la seva implementació. En el model d'herència múltiple, una nova classe pot tenir dos o més superclasses.

L'herència múltiple directa és útil quan una classe vol crear un nou comportament a partir dels comportaments de més d'una classe. Però en aquest cas apareixen problemes, ja que no existeix una manera única d'heretar el comportament.

Java utilitza el model d'herència simple per evitar aquests problemes. L'ús aquest model afavoreix el correcte disseny de les aplicacions. El problema del model múltiple és per a la implementació, no per a la interfície. Per aquest motiu, Java proporciona una manera per heretar la interficie però no la implementacio: el tipus *interface*.

Tot i així, Java permet l'herència múltiple indirecta, que resol el problema de l'herència múltiple directa afitant l'àmbit amb subclasses. És a dir, Java permet definir classes dins d'altres classes (*Inner Classes*) per tenir totalment especificada la implementació.

3.1.3 *Packages*

Els *packages*, o paquets, són agrupacions de classes, interfícies i *subpackages*. Els membres d'un package tindran certs privilegis entre ells. L'equivalent als paquets en altres llenguatges serien les biblioteques o *libraries*. Els packages són necessaris per diverses raons:

- Agrupen classes i interfícies relacionades entre elles.
- Creen un entorn de noms, que permet a les classes i interfícies utilitzar noms "locals" amb la seguretat que no entraran en conflicte amb noms iguals d'altres packages.
- Els packages poden tenir membres només accessibles des del propi package, però inaccessibles des de fora.

Per indicar al compilador que utilitzarem les classes de dins d'un package, utilitzarem la sentència

```
import nomPackage;
```

al principi de l'arxiu de codi font.

Els membres dins d'un package són referenciats com `nomPackage.membre`. Cada package té associat un directori al disc, a on són guardats tots els seus membres. Aquest directori té el mateix nom del package.

La API (*Application Programming Interface*) de Java consta d'un gran nombre de packages que són utilitzats per crear noves aplicacions. Entre aquests paquets destaquem:

- **java.io.** Amb els membres d'aquest package es poden utilitzar les funcions d'entrada i sortida, com per exemple la manipulació de fitxers.
- **java.lang.** Inclou totes les classes per treballar amb *Threads* i amb tipus especials.

- **java.applet.** Per construir Applets.
- **java.awt.** *Abstract Window Toolkit*. Té totes les classes necessàries per tractar amb finestres.
- **java.net.** En aquest packet es troben totes les classes que fan possible l'ús de telecomunicacions amb TCP/IP.
- **java.rmi.** *Remote Method Invocation*. Per poder utilitzar el sistema d'objectes distribuïts de Java.
- **java.security.** Paquet amb les classes per utilitzar signatures digitals, gestió de claus i altres funcions criptogràfiques.
- **java.util.** Paquet amb elements com vectors, dates, diccionaris, etc.

3.1.4 *Crear Packages*

Podem crear la nostra propia *biblioteca de classes* o *package* afegint la instrucció

```
package meva_llibreria
```

al començament de cada fitxer font. Qui la vulgui fer servir la inclourà amb

```
import meva_llibreria.*
```

3.2 Programació en Java

Una aplicació Java està integrament composta per classes. Com ja hem apuntat abans, Java no permet la inclusió de variables o funcions fora d'una classe (com succeeix a $C++$).

A Java es defineixen aquests tipus primitius:

tipus	valors
boolean	true, false
char	caràcter UNICODE (**16 bits!!**)
byte	8 bits, complement a 2
short	sencer 16 bits
int	sencer 32 bits
long	sencer 64 bits
float	punt flotant 32 bits
double	punt flotant 64 bits

El control de flux és com el del llenguatge C: blocs `if - else`, `switch`, `while`, `do - while`, `for`, i un afegit, `label - break - continue`, que tot i que pugui semblar una estructura *goto*, serveix només per sortir d'un cert grau de nidació en bucles.

3.2.1 Accés als membres d'una classe

Java permet controlar l'accés als camps i mètodes de les classes. Tots els membres d'una classe estan sempre disponibles per la mateixa classe, però l'accés des d'altres classes, així com per herència, es pot controlar mitjantçant aquest modificadors:

- **Public:** Els membres declarats amb aquest modificador són accessibles des de qualsevol lloc, i són heretats per les subclasses.
- **Private:** Aquest modificador farà que el membre sigui accessible només des de la mateixa classe que el conté.
- **Protected:** Quan un membre és declarat amb aquest modificador és accessible i heretat per les subclasses de la classe a on està declarat.
- **Package:** Els membres que no tinguin cap modificador tindran una accessibilitat de *package*, i.e seran accessibles i heretats només per les subclasses en el mateix package.
- **Static:** El valor de la variable és compartit per totes les instàncies de la classe. En el cas d'un mètode, no caldrà crear una instància de la classe per emprar aquell mètode. No podran tenir accés a variables i mètodes no estàtics.
- **Final:** L'efecte varia segons a què s'aplica. En el cas d'una variable, el seu valor no es pot modificar: és una constant. En el cas d'un mètode, cap subclase podrà redefinir-lo.
- **Abstract:** Un mètode es pot declarar i no fer-ne la implementació: la deixem per a les subclasses. Una classe amb aquest tipus de mètodes serà també de tipus abstract i no podrà esser instanciada. La combinació, doncs, *final abstract* és ilůlegal.
- **Native:** Una funció d'aquest tipus no està feta en Java, sino en algun altre llenguatge. Per exemple, emprant JNI (*Java Native Interface*) podem ïncrustar"en el nostre programa funcions fetes en C.
- **Volatile:** Una variable d'aquest tipus no pot ser optimitzada pel compilador.

- **Synchronized:** Amb aquest modificador l'execució d'aquest mètode serà exclusiva entre cadascun dels fils d'execució d'un objecte. En parlarem a l'apartat de *threads*.

3.2.2 Crear i utilitzar classes

En definir una classe haurem de declarar un mètode amb el mateix nom que ella del qual en direm el *constructor de la classe*. De fet, podrem definir diferents constructors ja que els mètodes de Java es poden sobrecarregar.

```
public class Point3D {
        // coordenades 3D del punt
        public float x;
        public float y;
        public float z;

        // constructors
        public Point3D() {
                this(0,0,0); // crida a un altre constructor
        }

        public Point3D(Point3D p) {
                this(p.x,p.y,p.z);
        }

        public Point3D(float x,float y,float z) {
                this.x=x;
                this.y=y;
                this.z=z;
        }
}
```

La instrucció per crear una nova instància (un objecte) d'una classe és el **new**. Quan creem un objecte s'assigna automàticament la memòria que necessita. El programador NO haurà d'alliberar mai aquesta memòria (no existeix `delete`), serà un sistema de *garbage collection* qui l'alliberarà automàticament quan calgui.

Els mètodes només reben paràmetres per valor. Això és conseqüència directa de que a Java **no es treballa amb punters**. De fet, això no és del tot cert. Els punters no es poden operar aritmèticament. Els únics punters amb els que es treballa són les referències a objectes, i conceptualment es poden considerar valors de tipus objecte. Això, com veurem programant, no és cap limitació, sinó tot el contrari.

Les classes tenen un constructor, que s'invocarà cada vegada que es faci una nova instància de la classe.

Si declarem els membres `static`, seran compartits per totes les instàncies de la classe.

Dintre de la classe tindrem dos objectes especials, `this`, que fa referència a la instància de la classe actual, i `super`, que fa referència a la superclasse.

3.2.3 Un programa d'exemple

Ja hem dit que a Java només tenim classes i no podem tenir funcions. Però, a on posem el nostre *main()*? Al igual que en C les aplicacions Java han de tenir un mètode `main` per poder ser executades.

En Java, la pròpia aplicació és una classe. Per poder ser executada ha de tenir un mèdote `main` amb una signatura concreta. Veiem-ho en aquest exemple:

```
1  class Nombre {
2    static final double PI = 3.14;
3    private int n;

4    Nombre (int n) { this.n = n; }
5    Nombre () { n = 0; }

6    public int suma (int x) { return n += x; }
7    public int getN () { return n; }
8  }

9  public class Exemple {

10 public static void main( String[] arguments ) {

11   Nombre a = new Nombre(3);

12   System.out.println( " 3 + 6 = " + a.suma(6) );
13  }

14 }
```

Cal remarcar que una classe pública ha d'estar en un fitxer amb el seu mateix nom i ha de tenir extensió `.java`. Aquest exemple ha d'estar en el fitxer `Exemple.java`.

El compilarem amb `javac Exemple.java` (es generarà el fitxer de *bytecode* `Exemple.class`) i l'executarem amb `java Exemple`.

A la línia [1] declarem la classe Nombre. La nostra classe tindrà un element PI ([2]) de tipus `double` amb valor 3.14. Aquest valor serà comú a totes les instàncies que tinguem de la classe, per això li afegim el modificador `static`. Volem que no sigui modificable, és a dir, el seu valor serà constant al llarg de l'execució. Per aconseguir-ho declarem la variable `final` (equivalent al `const` de $C++$).

A continuació, declarem una variable privada sencera n (línia [3]).

A les línies [4] i [5] definim dos constructors. Un amb un paràmetre, que posarà valor a n, i l'altre sense paràmetres, que posarà a n el valor 0.

A [6] definim un mètode $suma$ de la classe Nombre. Rebrà un paràmetre i el sumarà a la variable privada n i també el retornarà. Tenim un altre mètode a [7], $getN$, que retornarà simplement el valor n.

Finalment declararem la classe pública que serà l'aplicació ([9]). Per a que aquesta classe sigui executable haurà de tenir una funció $main$ ([10]) amb exactament aquesta signatura:

```
public static void main( String[] arguments )
```

El paràmetre que rep és un array d'Strings amb els arguments de la invocació.

A la línia [11] creem un objecte de la classe Nombre, cridant al seu constructor d'un paràmetre.

A [12] veiem la utilització d'un objecte d'ús molt freqüent, `System.out`, que serà la sortida estàndard del nostre programa. El seu mètode `println` seria l'equivalent a la funció $printf$ de C. Com a paràmetre aquest mètode necessita un String. Si li passem un altre tipus de dades, invocarà al seu mètode `toString` per convertir-lo a String. Veiem aquí també l'ús de l'operador sobrecarregat '+', concatenador de Strings.

Veiem que el nom de les classes comença sempre amb majúscules i el dels mètodes amb minúscules per convenció.

3.2.4 *Arrays*

En declarar una variable de tipus array cada element estarà inicialitzat al seu valor per defecte o `null` en el cas de les classes.

```
int[] a;         // crea una variable que pot guardar un array
a=new int[10];   // ara crea l'array, inicialitzat a 0
int[] b={1,2,3}; // el declarem i inicialitzem alhora

String[] s=new String[10]; // crea 10 strings, ara a null
```

```
for(int i=0;i<10;i++) s[i]=new String();
```

Un `a[-1]` o un `a[100]` provoca l'excepció `ArrayIndexOutofBoundsException` i posar dades de tipus equivocat provoca un `ArrayStoreException`. Vegeu el capítol de les *Excepcions*.

També podrem declarar arrays multidimensionals:

```
String[][] a=new String[10][];
for(int i=0;i<a.length;i++)
    a[i]=new String[5];    // cada element és un array de strings
for(int i=0;i<a.length;i++)
    for(int j=0;j<a[i].length;j++)
        a[i][j]=new String();

double[][] matriu={ {1.0, 0.0, 2.2}
                    {2.5, 3.4, 1.7}
                  };
```

3.2.5 *Subclasses i interfaces*

Les subclasses hereten les variables i mètodes de la superclasse (protected i publics). Totes les classes que definim seràn subclasses de la classe *Object*.

Si redefinim un mètode `f()` en una subclasse, en cridar `f()` o `this.f()` ens referirem al nou mètode, i amb `super.f()` cridarem el mètode que es va definir a la superclasse.

Ja hem dit que Java no suporta directament l'herència múltiple. Però el que podem fer es definir una *interface*, amb variables i mètodes sense instrumentar, i fer que altres classes, essent subclasse de qualsevol altre, els implementin. Vegem un exemple:

```
public interface QualitatsLiquides {
     float viscositat;
     public void servir (float velocitat);
}

// De la mateixa manera podriem definir altres qualitats, i
// ara emprar-les en una classe, que es preocuparà de la
// implementació dels mètodes.
```

```
class SucDeTaronja extends Begudes implements QualitatLiquides,
   QualitatsSabor, QualitatsColor {

      public void servir (float velocitat) {
          // instrumentació
      }

      // s'han d'instrumentar tots els mètodes de les interfaces
}
```

3.3 *Threads*

Cada procés d'execució en Java es diu *thread* (fil d'execució), i direm, doncs, que Java suporta multi-threading. Cal tenir en compte que per aquest tipus de programació s'han de prendre unes precaucions adddicionals, com ara l'exclusió mútua. Per sort, Java permet controlar aquestes qüestions amb molta facilitat.

S'ha de diferenciar entre els threads i els processos dins de Java. Els threads són la manera que té Java d'implementar la programació concurrent. Java permet també executar processos del sistema operatiu de la màquina amfitriona de la màquina virtual. Això s'aconsegueix amb el mètode `Runtime.getRuntime().exec(cmd)`, de manera molt similar a la funció `exec` de C.

3.3.1 L'ús dels threads a les aplicacions

Usar threads a Java és molt senzill. Cada thread que s'executa és la instància d'un objecte, per tant el que primer haurem de fer serà crear una classe. Per a que els objectes d'aquesta classe puguin ser llençats com threads tindrem dues opcions:

1. Extenent la classe *Thread*
2. Implementant la interfície *Runnable*

Ambdues maneres són equivalents. La única diferència està en que si triem la primera estarem extenent una classe, i, tal com hem vist en seccions anteriors, les classes Java només poden derivar d'una classe. Triarem aquesta opció només quan no necessitem extendre cap altre classe. La segona opció, implementant la interfície *Runnable*, permet que extenguem una classe i implementem altres interfícies. La manera de començar els threads serà també una mica diferent.

Tant si triem una o altra opció l'únic que haurem de fer és implementar el mètode `public void run()`. Per llençar els threads, si hem triat la primera opció, farem:

```
NovaClasse a = new NovaClasse();
a.start();
```

on `NovaClasse` és la classe que hem creat, que estén la classe *Thread* i que té definit el mètode `public void run()`.

Si haguéssim triat la segona opció, implementant la interfície *Runnable* faríem:

```
Runnable a = new NovaClasse();
new Thread(a).start();
```

Un cop "engegat" el thread existeixen maneres de controlar-lo. Aquests són alguns dels mètodes declarats a la classe *Thread* per controlar els threads:

- **start()** i **stop()** Comença i acaba l'execució del thread.
- **join()** Espera la finalització del thread.
- **sleep(long milis)** "Dorm" al thread els milisegons indicats.
- **yield()** Cedeix la execució a un altre thread.
- **suspend()** i **resume()** Suspen momentaniament i continua la execucióde thread.
- **wait()** i **notify()** Envia senyals als threads.

3.3.2 L'exclusió mútua

Com a tot llenguatge que suporti la programació concurrent, és necessària la senyalització de zones d'exclusió mútua, i.e. zones que pot estar executant un únic thread alhora, per a garantir la integritat i la coherència de les dades. Java incorpora un mecanisme de blocatge d'objectes que resol aquest problema. Quan un objecte està blocat per un thread només aquest thread pot accedir a l'objecte.

Aquest blocatge s'aconsegueix declarant el mètode que executarà el thread amb el modificador `synchronized`.

La “sincronització” força a que l’execució de dos threads que cridin a aquest mètode sigui mútuament excloïent en el temps. Niuar mètodes sincronitzats no comporta cap problema: si el que fa una crida al mètode és el propi thread que té blocat a l’objecte, el codi s’executa i no es desblocarà fins que retorni el mètode sincronitzat del nivell de niuament més exterior.

Aquí tenim un exemple on es pot veure la utilitat de la sincronització:

```
class A {

  int i , x=0;

  synchronized void f() {
    for ( i=0 ; i<10 ; i++ ) {
      x++;
      Thread.yield();
    }
    System.out.println(
      "i: " + i + " x: " + x );
  }
}
```

```
class T extends Thread {

  A a;

  T( A b ) { a=b; }

  public void run() {
    a.f();
  }

}
```

```
public class C {

  public static void main( String[] arguments ) {

    try {

      A a = new A();
      T t1 = new T( a );
      T t2 = new T( a );

      t1.start();
      t2.start();

    } catch( Exception e ) {
      System.out.println( e );
    }

  }
```

```
}
```

El resultat de l'execució és

```
i:  10 x:  10
i:  10 x:  10
```

Si no haguessim posat el modificador `synchronized` al mètode `void f()` el resultat hauria estat

```
i:  10 x:  11
i:  11 x:  11
```

3.4 Excepcions

Durant l'execució de les aplicacions es poden produir molts tipus d'errors. Quan els mètodes d'un objecte són invocats poden haver problemes d'estat intern (valors inconsistents a les variables), detectar-se errors manipulant les dades (no es pot resoldre una adreça de xarxa), o d'altres similars.

És molt difícil controlar totes les possibles condicions d'error (el codi esdevé ininteligible si després de cada invocació a un mètode comprovem totes les condicions d'error)

Java utilitza un mecanisme d'excepcions per controlar les condicions anormals del programa. La incorporació d'aquest mecanisme d'excepcions fa que el codi no excepcional (i.e. el que no s'encarrega de manipular els errors ni altres excepcions) es vegi més clar i sigui més fàcil de llegir i més fàcil de mantenir.

A Java, una excepció és qualsevol objecte que és instància de la classe *Throwable* o de qualsevol de les seves subclasses. Quan un mètode arribi a un cas excepcional (una divisió per zero, per exemple), podem dir-li que llenci una excepció. Aquesta excepció podrà ser després capturada i tractada en un bloc de codi diferent.

Quan un mètode pot llençar alguna excepció, s'ha d'indicar a la seva signatura, després de la paraula reservada `throws`. El mecanisme de captura d'excepcions és molt similar al del llenguatge C++, amb blocs de *try-catch*. Les execpcions que es produeixin dins d'un bloc *try* són tractades a la secció *catch* del bloc. Un bloc

try-catch pot tenir més d'una clàusula *catch*. Cada clàusula *catch* està associada a una classe d'excepció. En el següent exemple, es suposa que el mètode `llegir` produiex la excepció *FiFitxer*:

```
...

  try {

    while (true) {
      buffer = llegir( 100 , fitxer );
      procesar(buffer);
    }

  catch( FiFitxer exc ) {

    gestionarExcepcio();

  }
```

Les excepcions van pujant per la pila del sistema (sempre que els mètodes hagin declarat l'excepció a la seva clàusula de *throws*), fins que algun la capturi (*catch*). Per llençar una excepció s'utilitza *throw* seguit de l'excepció que es llença (recordem que l'excepció és una instància de la classe de l'excepció). Una línia de la funció `llegir` a l'exemple anterior hauria de ser:

```
...
  throw new FiFitxer();
...
```

Podrem també declarar les nostres pròpies excepcions construint subclasses de la classe `Exception`. Podrem enviar un string com a missatge a recollir amb el `getMessage`. Per exemple:

```
class laMevaExcepcioDeBD extends Exception {
    public laMevaExcepcioDeDB(String missatge) {

      super(missatge);
    }
}
```

Les excepcions són realment una forma molt potent de dividir l'espai de totes les possibles condicions d'error en parts tractables per separat. De totes les clàusules *catch*, s'executarà la primera que coincideixi.

Existeix una clàusula especial que es pot utilitzar al final del bloc *try-catch*, *finally*. El codi d'aquesta clàusula s'executarà "passi el que passi". Normalment s'utilitza per alliberar algun recurs extern després d'haver-lo adquirit, tancar un fitxer després d'haver-lo obert o situacions similars.

3.5 Entrada/Sortida

Un flux d'entrada és una font (o productor) de bytes, i un flux de sortida és un destí (o consumidor) de bytes. Tots els fluxes d'entrada són subclasses de la classe abstracta `InputStream`, i els de sortida ho són de `OutputStream`. La única excepció és `RandomAccessFile`, que és tant font com destí.

Aquestes classes es troben dins del paquet `java.io.*`. També trobarem la classe `File`, amb la que podrem obtenir informació d'un arxiu, de forma independent a la plataforma. Així, per obrir un flux d'entrada podriem fer:

```
File f=new File("fitxer.txt");
if (f.exists()) {
    FileInputStream istream=new FileInputStream(f);
}
```

De fet, els fluxes tenen diferents constructors, així que podem passar directament com a paràmetre la cadena que conté el nom del fitxer. El següent exemple fa una copia d'un fitxer:

```
import java.io.*;

public class Copia {

  public static void fer_copia(String desti, String origen) {

     try {
         FileInputStream fis=new FileInputStream(origen);
         FileOutputStream fos=new FileOutputStream(desti);
         int c;

         while((c=fis.read())!=-1) {
             fos.write(c);
```

```
 }

         fis.close();
         fos.close();

      } catch (FileNotFoundException e) {
         System.out.println(e);

      } catch (IOException e) {
         System.out.println(e);
      }

  }

  public static void main(String[] args) {
      if (args.length != 2) {
          System.out.println("Copia <origen> <desti>");
      } else {
          fer_copia(args[1],args[0]);
      }
  }
}
```

Però recordeu que un flux no necessàriament ha de ser d'un arxiu de disc. Per exemple, si mireu `java.net.URL`, podreu veure aquest mètode:

```
URL u = new URL("http://localhost/el_meu.html");
InputStream ris = u.openStream();
```

3.6 Applets

Una aplicació de Java es un programa independent que pot executar-se com qualsevol altre programa, però fent servir la màquina virtual Java. Un *applet*, però, és un programa que s'executa dins d'un navegador web. Així, ja compta amb un contexte d'una finestra i pot respondre a events de la intefície d'usuari del navegador. Com que els applets estan dissenyats per a executar-se a través de la xarxa, Java restringeix en gran part els accessos que poden fer els applets als arxius del client.

Ara no disposarem d'un mètode `main`, sino que haurem de instrumentar el mètode `paint` de la nostra subclasse de la classe `Applet`. Vegem un exemple senzill:

```
import java.applet.*;
import java.awt.Graphics;

public class Hola extends Applet {

   public void paint(Graphics g) {
   // aquest mètode es cridat automàticament

        g.setColor(Color.red);
        g.drawString("Hola! Benvingut a la meva web",5,10);
   }
}
```

Recordem que haurem de dir al fitxer amb aquest codi font `Hola.java`. Un compilat, necessitarem crear un fitxer $HTML$ que carregarem al nostre navegador, o amb l'`appletviewer` del JDK.

```
<APPLET CODE="Hola.class" WIDTH=300 HEIGHT=300></APPLET>
```

4 Les eines de desenvolupament

Les eines per al desenvolupament d'aplicacions Java que farem servir al laboratori venen donades pel JDK (*Java Development Kit*). Podeu aconseguir el JDK per a una gran quantitat de plataformes a:

```
http://java.sun.com/
```

Amb el JDK es poden crear immediatament aplicacions Java. El JDK inclou un intèrpret en temps d'execució de Java —*java*—, un compilador —*javac*—, un depurador —*jdb*—, l'API (Aplication Program Interface) i altres eines d'utilitat.

Cal anar amb compte amb les eines RAD (Rapid Application Development) de Java (M$ Visual J++, Borland JBuilder, etc). Algunes d'aquestes eines no generen codi portable ja que utilitzen biblioteques pròpies.

4.1 Compilar i executar un programa

El codi font d'una classe pública X ha d'estar dintre d'un fitxer amb el mateix nom de la classe i extensió `.java`. En aquest cas, `X.java`. Per compilar-lo simplement farem

```
javac X.java
```

i es generarà, si no han hagut errors, el fitxer `X.class` que contindrà el *byte-code* de la classe. Aquest és de fet el codi executable Java. Si la classe utilitzés altres classes, aquestes es compilarien automàticament. Si el codi font de classes utilitzades hagués variat, el compilador de Java ho detectaria i les recompilaria.

Per executar codi Java, és a dir, una classe Java, només cal utilitzar l'intèrpret en temps d'execució `java`. Aquesta és la màquina virtual que executarà la nostra classe.

```
java X
```

Per a que Java executi aquesta classe cal que estigui dintre d'un directori de classes. Java sap quins són els directoris de classes gràcies a la variable d'entorn CLASSPATH. Per defecte, en aquesta llista de directoris hi ha l'entrada `$HOME/java`. Per tant, heu de tenir les vostres classes en aquest directori o bé afegir en aquesta variable un nou directori de classes.

4.2 Documentació

Java basa la seva potència en un conjunt de classes estàndard. Aquestes classes, agrupades en *packages* formen el que es coneix per API (*Aplication Programming Interface*). Teniu la API de Java disponible al servidor de Web de la unitat:

`http://deic.uab.cat` (secció docència/recursos/manuals Web)

D'altra banda, a partir dels fitxers font es pot generar automàticament una documentació associada en $HTML$, amb el programa `javadoc`. Al document, a més de la jeràrquia i els mètodes de les classes, apareixen els blocs de comentaris que hagueu marcat amb `/** ... */`. No valen blocs del tipus `/* ... */` ni comentaris amb `//`. Dins del bloc podeu posar alguns tags de $HTML$. També podeu emprar etiquetes especials com aquestes:

- **@see** Crea una referència "miri també".
- **@param** Documenta els paràmetres d'un mètode.
- **@return** Documenta el valor que retorna un mètode.
- **@exception** Documenta les excepcions llançades per un mètode.

Aqui teniu un exemple:

```
/**
 * Aquest mètode localitza un element en un <B>arbre binari</B>.
 * @see GLlistaEncadenada
 * @see DLlistaEncadenada
 * @param obj element a buscar
 * @return retorna l'element trobat
 * @exception NoSuchElementException element no trobat
 */
```

Referències

[API] Sun MicroSystems. *API specifications*
`http://java.sun.com/reference/api/index.html`

[JGos96] Ken Arnold, James Gosling. *The Java Programming Language*.
The Java Series. Addison-Wesley 1996.

[MCam97] Mary Campione, Kathy Walrath. *The Java Tutorial, Object-Oriented Programming for the Internet*
The Java Series. Addison-Wesley 1997.
`http://java.sun.com/docs/books/tutorial/index.html`

[MMor98] Mike Morgan. *Descubre Java*.
Prentice Hall 1998.

[MChan98] Mark C.Chan, Steven W. Griffith, Anthony F.Iasi. *1001 Tips para programar con Java*
McGrawHill 1998.