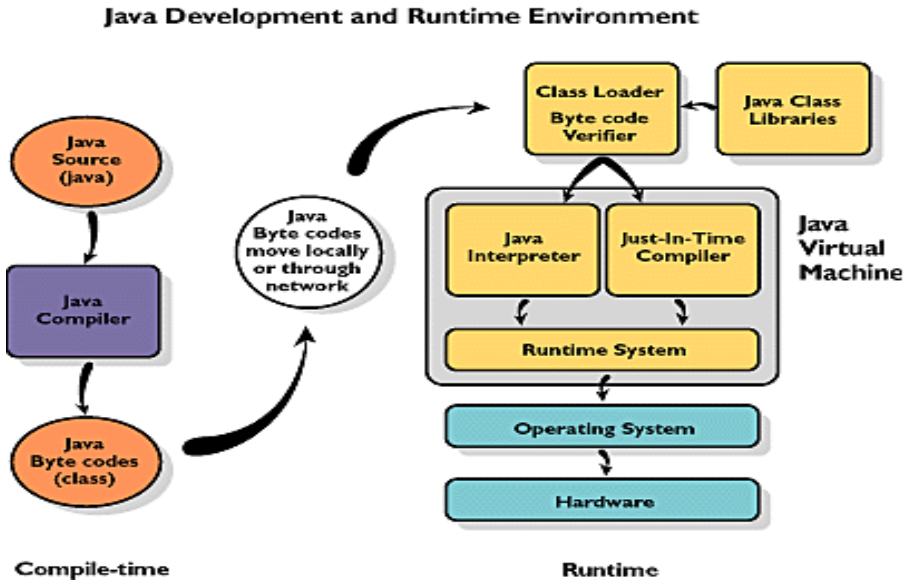
U.T. 2.- PROGRAMACIÓ MODULAR I PROGRAMACIÓ ORIENTADA A OBJECTES



1. MODULARITAT I FUNCIONS

Programació modular: tècnica sorgida en els anys 60s que proposa fer ús de la tècnica de "divideix i venceràs" (en diversos nivells). Suposa que cada programa ha de dividir-se en mòduls, i que cada mòdul ha de tornar a dividir-se sempre que es considere que realitza més d'una tasca. En la pràctica, l'experiència ens diu que cada mòdul no hauria d'excedir 50 línies o l'anvers d'un A4. Aquesta tècnica va començar a utilitzar-se per a la programació amb llenguatges d'alt nivell propis de l'època, de 3ª generació, juntament amb la programació estructurada, i es continua utilitzant en altres paradigmes de programació.

Els mòduls, segons el llenguatge de programació, són anomenats mòduls, submòduls, subprogrames, rutines, subrutines, etc. En Pascal, per exemple, tenim 2 tipus de mòduls segons retornen, o no, un valor (són anomenats, respectivament, funcions i procediments). En altres, com C, tots els mòduls són

anomenats funcions. És propi de la programació orientada a objectes (POO) cridar-los **mètodes**, que vindrien a significar funcions no independents, sinó incloses sempre a l'interior d'una classe.

Cada programa, en C o en Java, ha de tenir sempre una funció (o mètode) principal: main. Però hi han altres classes que es dissenyen, no per a fer un programa, sinò per a definir nous tipus de dades. Aquestes classes seran anomenades classes instanciables.

La sintaxi general de **definició d'una funció** o mòdul en Java és:

```
tipusValorRetornat nomFuncio( tipus1 param1, ..., tipoN paramN)
/*Si no hi han paràmetres, els parèntesi també es posen */
{
    //definició de variables locals

    //algorisme de la funció

    return expressió; /* return és opcional*/
}
```

Vegem un exemple:

```
    // DEFINICIÓ DE LA FUNCIÓ mitjana
double mitjana(double n1, double n2)
{
    return (n1 + n2) / 2;   // retorna la mitjana aritmètica
}
```

Quan una funció no retorna cap valor, la indiquem de tipus **void**. Per exemple:

```
void saluda(String s)
{
    System.out.println("Hola " + s);
}
```

return és la paraula reservada del llenguatge per tornar un valor. La utilitat de return és doble: retorna el control a la funció des de la qual es va cridar, i retorna un valor del tipus indicat. Quan la funció no retorna valor (funció de tipus *void*) es pot utilitzar *return*, sense cap valor, únicament per abandonar la funció. Exemple:

```
void meuaFuncio(...)
{
    ...
    if (condicio) /* eixir de la funció si es compleix la condició */
        return;
    /* Si la condició era certa, la resta d'instruccions no s'executen */
    ...
}
```

Encara que no és habitual, una funció pot estar definida dins d'una altra. Això suposa que és funció local i que, per tant, només pot ser cridada des de dins d'ella.

Exemple: Realitzar una funció en Java que mostre tants asteriscos com indique el valor passat com a paràmetre:

```
void imprimeixAsteriscos(int num)
{
```

```
    int i;
    for ( i = 1; i <= num; i++ )
        System.out.print('*');
}
```

Quan una funció fa un càlcul per a obtindre un valor, la millor opció serà que aquesta funcione retorne el valor en lloc de mostrar-lo a pantalla. En el primer cas, eixa funció la podré reutilitzar en tots els casos; en el segón no ens resultarà útil en aquells casos en els que només interesse obtindre el valor i no que es mostre. I recorda que la funció ha de fer una única tasca, no vàries.

```
   // Aquesta versió de la funció mitjana no retorna el valor, sino el mostra ...
void mitjana(double n1, double n2)
{
    System.out.println((n1 + n2) / 2);
}
/* ... per tant no sempre ens serà útil:
  és millor opció la versió de mitjana de tipus "double" vista anteriorment */
```

Ja sabem què és la definició d'una funció. En ella, com acabem de veure indiquem el tipus de valor retornat, el nom de la funció, els seus paràmetres i el bloc de codi corresponent a la funció, però què és el **prototip** d'una funció? És la primera línia de la definició, en la que indiquem el tipus, nom i paràmetres. Per exemple, el prototip de la funció *mitjana* anterior seria:

```
// Prototip de la funció: indica la informació básica per a usar la funció
void mitjana(double n1, double n2);
```

2. ÚS O CRIDA A LES FUNCIONS

Distinguirem 2 casos (funcions que no retornen un valor, i funcions que si el retornen).

En el primer cas, la crida a la funció s'efectuarà generalment des de *main*, o també pot ser des d'alguna altra funció que s'haguera cridat previament. L'efecte serà que es passarà a executar el codi de la funció trucada i, quan aquesta acabe, es continuarà amb la següent instrucció del mètode inicial. La sintaxi d'aquesta crida serà:

```
nomFuncio(parametre1, ..., parametreN);
```

En el segon cas, quan retorne un valor, la diferència consistirà en què la funció ha d'assignar el valor retornat a una variable

```
variable = nomFuncio(parametre1, ..., parametreN);
```

També serà possible imprimir-lo o utilitzar-lo des de la crida a una altra funció. Vegem un exemple en el que cridem a la funció *mitjana* des d'un *println*:

```
System.out.println( "La mitjana de tots dos valors és" + mitjana(n1, n2));
```

Observa com, en ambdós casos, en la crida no s'indiquen els tipus dels paràmetres, en la definició si.

Tot i que la funció no admeta paràmetres, es requereixen parèntesi tant en la definició com en la crida.

3. PAS DE PARÀMETRES A FUNCIONS

- Paràmetres actuals: són els que apareixen en la crida a la funció
- Paràmetres formals: apareixen en la definició de la funció

Ha d'existir una correspondència (o coincidència), en nombre i tipus, entre els paràmetres actuals i els formals d'igual posició, ja que els paràmetres formals recullen un a un els valors de cada paràmetre actual. Exemple :

```
/*Crida a la funció: conté els paràmetres actuals*/
funció (3.5, 'm', 8);

/*Definició de la funció: paràmetres formals*/
tipus funcio (double x, char c, int n)
{
    ...
}
```

ATENCIÓ! No caiguem en 2 errades molt comuns en principiants. Per a això:

1. en la crida no s'han d'indicar els tipus, ni de la funció ni dels paràmetres.
2. si la funció és de tipus *void*, la crida no anirà en una assignació.

4. ÀMBIT DE LES VARIABLES

Variable local: aquella definida dins d'una funció (incloent els paràmetres formals). A més, hi han les variables internes a un bloc de codi (delimitat per les claus {}, com podria ser un *if*, *else*, *while*, *for*, *do-while*, etc.). Aquestes variables només ocupen memòria, i per tant només poden ser utilitzades, quan s'executa el codi al seu interior (tota la funció o tot el bloc). Fora d'eixa porció de codi desapareixen de la memòria RAM. Això és el que coneixem com a àmbit de la variable.

Variable global: aquella definida fora de qualsevol funció. En altres llenguatges poden existir encara que no s'aconsella el seu ús. En Java no poden existir perquè hauran d'estar incloses dins d'una classe com a atributs. Es veuran posteriorment.

Exemple :

```
... main (...)
{
    int n2; /* Variable local a main */
    ...
}

void meuaFuncio (...)
{
    int n3; /* Variable local a meuaFuncio */
    ...
}
```

En general, és important **reduir l'àmbit** de les variables en la mesura que siga possible. Per tant, és preferible treballar amb variables locals a fer-ho amb variables globals. Quan ampliem l'àmbit d'una variable augmentem la probabilitat d'errors i dificultem la seua depuració.

5. DIVISIO MODULAR

En general, la divisió modular ha de procurar que cada funció faça únicament una cosa. Observa el següent programa. Demana valors enters (fins a acabar amb un zero) i mostra, per a cada valor la suma des d'1 fins al propi valor (per exemple, donat un 5 mostra el resultat de la suma 1 + 2 + 3 + 4 + 5 = 15).

```java
import java.util.Scanner;

public class modularitatIncorrecta
{
	private static Scanner ent = new Scanner(System.in);

	public static void main (String args [])
	{
		int num1;

		System.out.println("Introduix un enter positiu (0 per a acabar)") ;
		num1 = ent.nextInt();
		demanarNum(num1);
	}

	// DIVISIÓ MODULAR INCORRECTA: LA FUNCIÓ HO FA TOT
	public static void demanarNum(int n1)
	{
		int suma, aux;

		while (n1 != 0)
		{
			if (n1 < 0)
				System.out.println("Valor no vàlid, s'ignorarà");
			else
			{
				suma = 0;
				aux = n1;
				while (aux > 0)
				{
					suma = suma + aux;
					aux--;
				}
				System.out.println("La suma és " + suma);
			}
			System.out.println("Introduix un enter positiu (0 per a acabar)") ;
			n1 = ent.nextInt();
		}
		System.out.println("Fi del programa");
	}
}
```

La solució mostrada és perfecta funcionalment, també és eficient, però peca d'una modularitat incorrecta. Compara-la amb la següent solució alternativa.

```java
import java.util.Scanner;
public class modularitatCorrecta
{
	public static void main (String args [])
	{
		Scanner myScanner = new Scanner(System.in);
		int num1;

		System.out.println("Introduix un enter positiu (0 per a acabar)") ;
		num1 = myScanner.nextInt();
		while (num1 != 0)
		{
			System.out.println("La suma és " + sumaValors(num1));
			System.out.println("Introduix un enter positiu (0 per a acabar)") ;
			num1 = myScanner.nextInt();
		}
	}
```

```
        public static long sumaValors(int n1)
        {
                int suma = 0, aux = n1;

                while (aux > 0)
                {
                        suma = suma + aux;
                        aux--;
                }
                return suma;
        }
}
```

Aquesta divisió modular sí es correcta perque la funció només fa una cosa: retorna la suma a partir de l'enter que actua com a valor d'entrada. Observa que tampoco el mostra, només el retorna. Normalment és prefrible retornar el valor a mostrar-lo, això augmenta les probabilitats de reutilitzar la función: en altra ocasió podem estar interessats en obtindre el resultat, sense que es mostre per pantalla, per a un càlcul posterior. Si la funció estiguera feta per a mostrar el valor no la podríem reutilitzar en eixe cas.

Altre exemple. Imagina que volem fer un programa que demane un enter (per exemple, 4) i mostre per pantalla el següent triangle:

1

1 2

1 2 3

1 2 3 4

Podriem fer algo com:

```
... main (...)
{
    int num;

    // ... demana un enter (num) i el llig des de teclat
    mostraTriangle(num);    // num és l'enter introduït
}

public void mostraTriangle(int num)
{
    for (int i = 1 ; i <= num ; i++)
    {
        for (int j = 1 ; j <= i; j++)
            System.out.print(j + "\t");
        System.out.print('\n');
    }
}
```

Esta solució seria correcta funcionalment, però tampoc seria la divisió modular més apropiada. Seria millor esta altra:

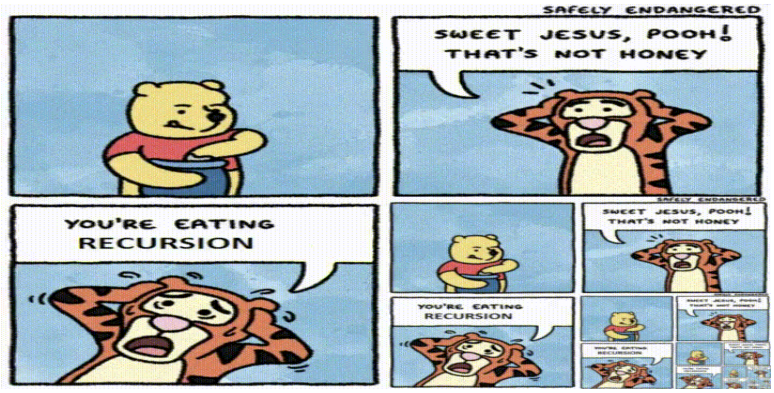
```
... main (...)
{
        int num;
```

```
        // ... demana un enter (num) i el llig des de teclat
        for (int i = 1 ; i <= num ; i++)
        {
                mostraLinia(i);
                System.out.print('\n');
        }
}

public void mostraLinia(int limit)
{
        for (int i = 1 ; i <= limit ; i++)
                System.out.print(i + "\t");
}
```

6. RECURSIVITAT

Per entendre la recursivitat, primer s'ha d'entendre la recursivitat. (Anònim)



Funció recursiva: aquella que, en la seva definició es crida a si mateixa :

```
tipus funcio (...)
{
    ...
    if (condicio)
     funcio(...); // aquí es produeix la recursivitat
    ...
}
```

Exemple típic de recursivitat: el factorial d'un sencer. En matemàtiques:

5! = 5*4*3*2*1 = 120

Aquesta operació factorial (en matemàtiques s'utilitza l'operador !) compleix la següent propietat per a qualsevol valor positiu de n:

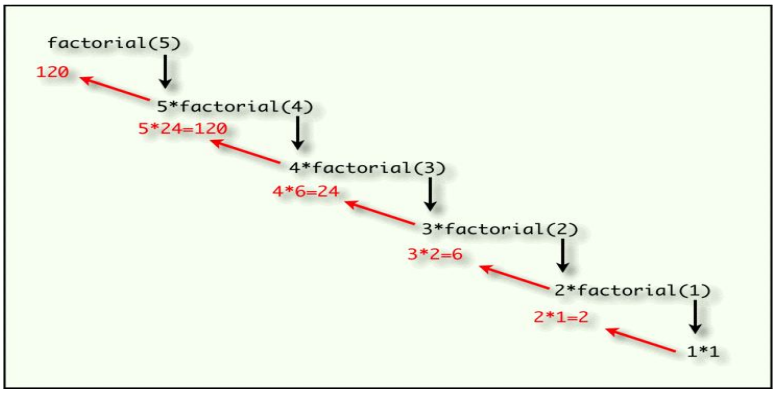
n! = n * (n-1)!

En Java serà :

```
int factorial(int num)
{
    if (num > 1)
        return num * factorial(num-1);      // crida recursiva
    else // per a num igual a 1 o inferior
```

```
        return 1; // Fi de la recursivitat
}
```

Quan es crida a una funció recursiva, tindrem diferents instàncies de la mateixa funció. L'última instància creada serà la primera a acabar i la primera instància l'última a acabar.



Perquè una funció recursiva estiga ben dissenyada ha d'haver un límit a la recursivitat (una eixida). ¡¡ La recursivitat no ha de ser mai infinita !!.

Exercici: fer un programa que utilitze aquesta funció.

La modularitat sempre té un cost en l'eficiència del programa. Sempre que cridem des d'una funció a una altra suposa com un canvi d'escenari (s'han de retirar unes variables, les locals, per a posar en el seu lloc a altres). Igual quan retornem des de la funció cridada: s'han de retirar les variables locals de la funció per a recuper les de la funció de partida, etc.

Per això mateixa, la recursivitat és especialment "cara" pel que fa a l'eficiència del programa, però en ocasions algunes funcions que serien molt complexes sense la recursivitat, queden molt senzilles amb ella. Per a solucionar la pèrdua d'eficiència de les solucions recursives tenim els optimitzadors de codi inclosos en els compiladors. En ocasions, aquests optimitzadors són capaços de transformar una solució recursiva (més còmoda per al programador) en una solució iterativa (més eficient).

Per exemple, el factorial en solució iterativa, no recursiva, seria:

```
int factorial(int num)
{
    int i, result = 1;

    for (i = 2 ; i <= num ; i++)
        result = result * i;
    return result;
}
```

7. CARACTERÍSTIQUES DE LA POO

La filosofia o paradigma de la programació orientada a objectes (POO), per contraposició a la programació estructurada clàssica, no estarà tan basada en el disseny dels algorismes, sinó en el de les dades o objectes que intervenen en el programa. Alguna de les propietats fonamentals que caracteritzen la POO seran:

- **abstracció**. Cada tipus de dada s'ajustarà a les nostres necessitats, és a dir, farem una representació a la mida de les nostres necessitats.
- **encapsulació** de dades. Les dades contingudes en una classe estaran protegits enfront de canvis indeguts que puguen dificultar el manteniment i depuració dels programes.
- **polimorfisme**. Les dades i les funcions poden adoptar diferents formes segons ens interesse.

- **herència**. Permetrà definir uns tipus de dades com a extensió d'altres prèviament existents. Per a això, ambdós tipus hauran de tenir una relació entre ells de el tipus "és un" (per exemple, un rectangle és una figura).

8. L'ENTORN DE DESENVOLUPAMENT JAVA

El JDK (*Java Development Kit*), tot i que no conté cap eina gràfica per al desenvolupament de programes, sí que conté aplicacions de consola i eines de compilació, documentació i depuració. El JDK inclou el JRE (Java Runtime Environment), que consta dels mínims components necessaris per executar una aplicació Java, com són la màquina virtual i les llibreries de classes.

És moment de tornar al nostre primer exemple, el *HolaMon* :

```
public class HolaMon
{
    public static void main(String args [])
    {
        // mostra el missatge per pantalla
        System.out.println( "Hola món!");
    }
}
```

A la vista de tan escàs codi podem fer algunes puntualitzacions:

- en Java no existeixen funcions independents. Totes les funcions estaran incloses en alguna classe i, com a tal, han de ser mètodes d'aquestes classes. És per això que, a diferència de C++, es tracta d'un llenguatge orientat a objectes pur o amb un sentit més estricte de l'orientació a objectes. En el cas de C++ sí que hi han funcions independents, la mateixa *main()*, per exemple, no s'inclou en cap classe.
- els comentaris (// per a una sola línia i / *... */ multilínia) són els ja coneguts de C i C++.

En Java generalment cada classe és un fitxer diferent. Si hi ha diverses classes en el fitxer, la classe que s'anomena amb el nom del fitxer hauria de portar el modificador *public* i és la que es pot utilitzar des de fora de l'arxiu. Les classes tenen el mateix nom que el fitxer *.java* i cal que majúscules i minúscules coincidisquen.

El mètode **main** té les següents propietats:

- és públic, per poder cridar-lo des de qualsevol costat.
- és estàtic, per poder cridar-lo sense haver d'instanciar la classe.
- no torna cap valor (modificador *void*).
- admet opcionalment una sèrie de paràmetres *(String args[ ])*, que en aquest exemple no són utilitzats.

9. INTRODUCCIÓ ALS OBJECTES I LES CLASSES

La classe vistes als anteriors exemples eren classes orientades a l'execució del codi del nostre programa (les instruccions de *main()). En Java podem crear classes com l'anterior (no instanciables), les quals contindran un mètode *main()* i potser alguns més, però també classes definides per a posteriorment ser instanciades repetides vegades. Aquestes últimes són les que anomenarem **classes instanciables**, definides per a poder crear objectes d'aquests tipus. Un exemple d'una d'aquestes classes seria :

```
// rectangle.java
class rectangle
{
    // ATRIBUTS o propietats de la classe
```

```
    private double ample;
    private double alt;

    // MÈTODES
    // constructor sense paràmetres
    public rectangle() {ample = 1; alt = 1; }
    // constructor general
    public rectangle(double a1, double a2) {ample = a1; alt = a2; }
    // mètode setter
    public void setDimensions(double a1, double a2) {ample = a1; alt = a2; }
    // mètodes getters
    public double getAmple() {return ample; }
    public double getAlt() {return alt; }
    // resta de mètodes
    public double area() {return ample*alt; }
}
```

Com pots veure en la definició de la classe rectangle, aquesta es compon d'**atributs** (o propietats, que actuaran com a variables locals de la classe) i **mètodes** que, a diferència de les funcions del llenguatge C (com molts altres), que eren independents, aquestes sempre estaran dins d'una classe.

L'anterior classe és un bon exemple d'aquest segon tipus de classes: les classes instanciables. Definim el codi de la classe i això ens permetrà posteriorment, en la mateixa aplicació o altres, crear diferents o repetides instàncies (objectes) que seran variables locals o atributs, segons on es defineixen:

```
// programa.java
public class programa
{
    public static void main (String args[ ])
    {
        // cree un rectangle
        rectangle r1 = new rectangle ();
        // cree altre rectangle
        rectangle r2 = new rectangle (3,4);
        // mostre les seves àrees
        System.out.println ( "L'àrea del primer és " + r1.area () + " i del segon " + r2.area ());
    }
}
```

10. TIPUS DE DADES I OPERADORS DEL LLENGUATGE

Els tipus de dades s'utilitzen generalment al declarar variables i són necessaris perquè l'intèrpret o compilador conega previament el tipus d'informació que contindrà una variable i la grandària en bytes de memòria a reservar. Els tipus de dades primitius en Java són els següents.

Tipo	Significado	Cantidad de bytes por cada valor
byte	Números enteros cortos [-128..0..127]	1
short	Números enteros cortos [-32768..0..32767]	2
int	Números enteros	4
long	Números enteros largos	8
float	Números con coma flotante o decimales [de 7 dígitos decimales]	4
double	Números con coma flotante o decimales [de 14 dígitos decimales]	8
char	Caracteres simples	2
boolean	Valores lógicos [true;false]	1

Com en C, és usual en Java utilitzar majúscules per als identificadors de les constants, mentre que les variables utilitzen minúscules. Les constants es declaren seguint el següent format

```
[final] [static] tipus identificador = valor;
```

El qualificador *final* identifica la dada com a constant, mentre que *static* es veurà el seu significat més endavant. Un exemple seria:

```
final static double PI = 3.14159;
```

Els literals en Java poden ser una expressió:

- de tipus de dada simple
- amb el valor null
- amb una cadena de caràcters ("Hola món", per exemple).

L'àmbit de cada dada, variable o constant, com ja hem dit, depèn d'on siga declarat. En general, cada dada podrà ser utilitzada al llarg de tot el bloc on ha estat declarat, entenent com a bloc la porció de codi delimitada per les claus d'obertura i tancament {} dins de les quals ha estat declarat. D'aquesta manera, una variable que es definisca, per exemple, dins d'un bucle només podrà utilitzar-se des de l'interior d'aquest i no una vegada estiguem fora. Les variables locals a un mètode es poden utilitzar al llarg de tot el mètode, mentre que els atributs podran ser accedits des de qualsevol localització interior a la classe, en general en qualsevol dels seus mètodes.

A part de les dades primitives, qualsevol altra classe (pròpia de Java o creada per nosaltres) actuarà com a tipus de dada igualment utilitzable en els nostres programes.

11. MEMBRES D'UNA CLASSE

Cada classe definida en Java va a poder contenir atributs i mètodes. Observa, com a exemple, l'anterior implementació de la classe *rectangle*.

Els **atributs** d'una classe defineixen les propietats que van a tenir cadascun dels objectes que es instancien. Constitueixen el que, generalment, podríem definir com a part oculta de la classe. Això és així perquè els atributs han de estar protegits front a canvis des de l'exterior de la classe; per això els declarem **privats** (o, al menys, no públics). Amb això es redueix dràsticament la possibilitat d'errors per

canvis indeguts. Qualsevol atribut d'una classe podrà ser utilitzat des de l'interior de la classe com *this.atribut* o simplement *atribut*. La paraula reservada *this* és sempre una referència a l'objecte amb el qual treballem.

Per la seva banda, els **mètodes** a les classes defineixen la part visible d'aquestes. Són normalment declarats **públics** i ens permeten definir el comportament de la classe, és a dir, el conjunt d'operacions que voldrem fer amb els objectes de la classe. A major nombre de mètodes inclosos en la classe, millor o més variat comportament tindran els objectes.

En la majoria de classes tindrem tres tipus de mètodes:

- *constructors*: són els utilitzats per instanciar o crear els objectes. Si a la classe definisc diferents constructors podré crear-los de diferents formes, amb diferents inicialitzacions per als seus atributs.
- *setters/getters*: són els mètodes utilitzats per modificar (*setters*) o llegir (*getters*) el valor de cadascun dels atributs. Aquests mètodes permeten interactuar, des d'altres classes, amb els atributs de la pròpia classe sense haver de fer-ho directament (encapsulació). Això redueix dràsticament la possibilitat d'errors per canvis indeguts.
- mètodes de comportament: són la resta de mètodes que permeten realitzar qualsevol altra operació.

Membres estàtics

Un membre estàtic a Java en general significa que es tracta d'un membre, atribut o mètode, que pertany a una classe i no a les instàncies individuals. Per tant, no cal accedir a un membre o mètode estàtic a Java amb una referència a l'objecte, sinó directament amb el nom de la classe.

El membre estàtic de la classe no ocupa memòria per instància, és a dir, tots els objectes comparteixen la mateixa còpia del membre estàtic. El membre estàtic es pot utilitzar independentment de qualsevol objecte d'aquesta classe. Podem accedir al membre estàtic de la classe abans de crear qualsevol objecte. El millor exemple de membre estàtic és el mètode *main()*, es declara estàtic de manera que es pot invocar abans que existisca qualsevol objecte.

Per poder invocar un mètode cal fer-ho sobre un objecte. Però com és possible cridar *main*, si en iniciar l'execució del programa encara no existeix cap objecte? Els objectes es creen precisament dins el *main*! Aquest problema seria un peix que es mossega la cua. La resposta està a fer-lo static, de manera que és possible fer-ne la crida sense la necessitat que hi haja cap objecte existent prèviament.

Lal forma general per accedir a un membre estàtic és:

```
nomClasse.membreStatic          // accedint al membre estàtic de la classe
```

Al codi anterior, nomClasse és el nom de la classe en què es defineix membreStatic, que pot ser, com hem dit, atribut o mètode.

Atributs de tipus objecte

Quan tinguem un atribut que no siga de tipus primitiu o bàsic i, per tant, siga també objecte, si volem accedir als seus membres hem d'usar la sintaxi "objecte.membre.submembre" (concatenant punts) com mostra el següent exemple.

```
class Tennista
{
        private String nom;
        public void setNom(String nouNom) { nom = nouNom; }
        ...
}
```

```
class partitTennis
{
        private Tennista t1;
        private Tennista t2;
        public void setPrimerTennista(Tennista t) { t1 = t; }
        public void setSegonTennista(Tennista t) { t2 = t; }
        public Tennista getPrimerTennista() { return t1; }
        public Tennista getSegonTennista() { return t2; }
        ...
}
```

Si volem establir o canviar el nom del primer tennista d'un partit existent fariem:

```
partitTennis pt1 = new partitTennis();
// NO ÉS POSSIBLE ACCEDIR DIRECTAMENT A t1 PER SER PRIVATE
//pt1.t1.setNom("Rafa Nadal");
// SI ÉS POSSIBLE ACCEDIR AL GETTER PER SER PUBLIC
pt1.getPrimerTennista().setNom("Rafa Nadal");
```

En aquestos casos serà necessari evitar l'excepció *NullPointerException*. Per això s'ha d'evitar que *getPrimerTennista()* retorne el valor *null* per no tindre encara l'objecte *Tennista* corresponent assignat.

12. CONSTRUCTORS I DESTRUCTORS

Els constructors són mètodes especials de classe que s'utilitzen per inicialitzar els objectes en l'instant de la seva creació. Tenen una sintaxi especial, diferenciada de la resta de mètodes, així com també característiques especials.

Quant a la seva sintaxi, el seu nom ha de coincidir amb el de la classe, i no han de retornar cap valor, tot i que no són de tipus *void* per això:

```
public rectangle()
{
    ample = alt = 1;
}
```

La característica especial que el diferencia de la resta de mètodes és que són cridats automàticament al definir l'objecte mitjançant l'ús de la paraula reservada **new**. Per exemple:

```
rectangle r1 = new rectangle();
```

No són mètodes que puguen ser cridats explícitament, en qualsevol moment, com succeeix amb la resta de mètodes.

El segon tipus de constructor és el que accepta paràmetres, generalment un per a cada valor dels atributs, permetent inicialitzar l'objecte als valors desitjats:

```
public rectangle(double a1, double a2)
{
    ample = a1; alt = a2;
}
```

Com a l'anterior exemple de la classe *rectangle*, usualment definim al menys dos constructors: un que inicialitza a uns valors fixos decidits per nosaltres com a valors per defecte (en aquest cas la unitat, tant

per l'ample com per a l'alt), i un segon (d'ús més general) que ens permetrà triar els valors inicials al admetre aquests com paràmetres. Cada vegada que es crea un nou objecte de la classe amb *new* s'està cridant a un dels constructors existents a la classe. La forma de determinar quin constructor s'utilitza en cada cas serà pels paràmetres utilitzats al fer ús de *new*. Per exemple

```
rectangle r2 = new rectangle(2,3);
```

En aquest últim cas estem creant l'objecte mitjançant el segon constructor, el qual hem anomenat "general". Mentre que l'anterior objecte *r1* s'havia creat amb el primer constructor, el qual no admet paràmetres.

Un tercer tipus de constructor és el constructor **de còpia**. Aquest constructor admet com a paràmetre un objecte de la mateixa classe, ja creat anteriorment, i crea el nou objecte com a còpia del primer, amb els mateixos valors per als seus atributs :

```
public rectangle(rectangle r)
{
    ample = r.ample; alt = r.alt;
}
```

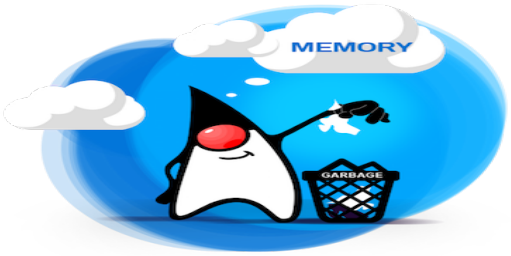
La definició dels constructors dins de cada classe no és obligatòria per al programador. Si no incloem cap constructor a la classe, el compilador ens inclourà automàticament un que inicialitze els atributs amb valors nuls (zeros per als atributs numèrics). Això sí, quan hem inclós un constructor amb paràmetres i es necessite un constructor per defecte, sense paràmetres, el compilador no el genera automàticament sinò que generaria una errada de compilació al crear un objecte que intentara fer ús d'aquest constructor.

Inicialitzadors statics. A diferència dels inicialitzadors d'instància vistos, també poden existir un o més inicialitzadors de classe (inicialitzadors static). Son mètodes especials, sense paràmetres ni nom, que s'executaràn quan es carrega en memòria per primera vegada una classe (generalmente al crear el primer objecte de la classe). La seua sintaxi és:

```
static
{
        ...
}
```

Un inicialitzador static és similar a un mètode (un bloc {...} de codi, sense nom i sense arguments, precedit per la paraula *static*) que es crida automàticament al crear la classe (al utilitzar-la per primera vegada). També es diferencia dels constructors en què no és cridat per a cada objecte, sinó una sola vegada per a tota la classe. Podríem dir que és una cosa així com un constructor de classe, davant dels constructors d'objectes coneguts fins ara. Entre altres usos, se solen utilitzar per inicialitzar variables estàtiques de la classe.

Per acabar, hem de destacar que Java, a diferència de C++ i altres llenguatges de programació orientats a objectes, no utilitza destructors per alliberar l'espai de memòria ocupat per l'objecte quan es deixa d'utilitzar. En aquests llenguatges és responsabilitat de l'programador alliberar (generalment fent ús de la paraula reservada *delete*) la memòria prèviament reservada amb *new*. En canvi, en Java el programador es veu descarregat de la necessitat de fer això mateix, atès que la JVM inclou un component que automatitza aquesta tasca (generalment, amb alguna periodicitat, va comprovant objectes ja no utilitzats i alliberant la seva memòria). Aquest és el conegut com *Garbage collector* o recol·lector d'escombraries.



13. CONTROL D'EXCEPCIONS

El control d'excepcions permet el programador preveure la resposta del programa enfront de diversos problemes que puguen ser previstos, evitant que el programa falle de manera brusca.

Bàsicament, fa ús de les paraules reservades *try* i *catch* de manera que el programa intentarà protegir les sentències situades dins d'un bloc tancat per *try*. En el cas que es produeixca un error s'intentarà controlar l'excepció mitjançant els blocs *catch* (depenent de l'excepció s'executarà un bloc o un altre). El bloc *finally* és opcional, però en el cas d'existir s'executarà sempre.

```
try {
        // Bloc de codi on pot aparéixer un error
}
catch (Exception e)
{
        // Bloc de codi on es tracta el problema (resposta al problema)
}
finally
{
        // Bloc de codi que s'executa sempre
}
```