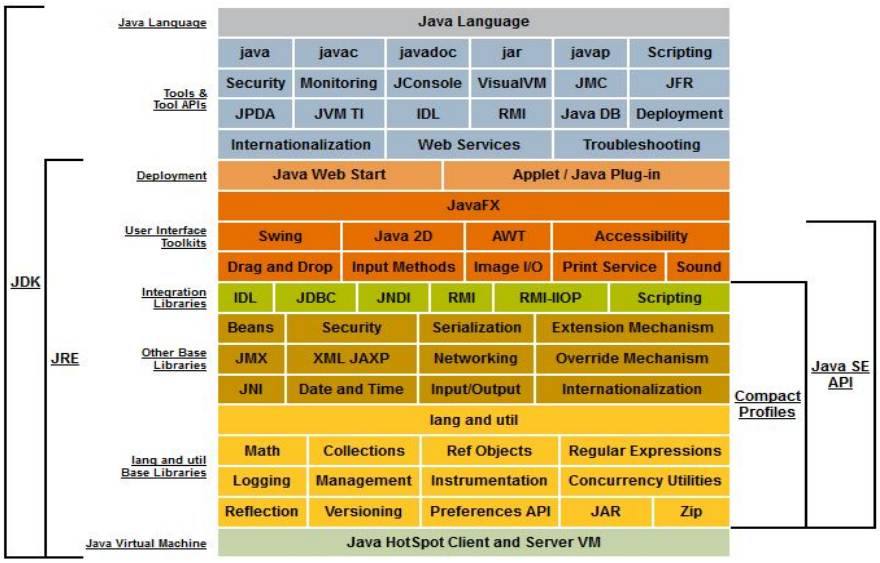
U.T. 4.- PROGRAMACIÓ AVANÇADA AMB JAVA



1 - HERÈNCIA

L'herència és una de les propietats fonamentals de la POO. Ens permet definir una nova classe (classe **derivada**) a partir d'una classe prèviament existent (classe **base**). La classe derivada pot afegir els seus propis atributs i mètodes, però tindrà a més, sense necessitat de declarar-les explícitament, els atributs i mètodes definits en la classe base:

```
class Jugador // classe base
{
    // atributs
    String nom;

    public void setNom (String n) {nom = n; }
    public String getNom () {return nom; }
}

class Futbolista extends Jugador // classe derivada
```

```
{
    int gols;

    public void setGols (int g) {gols = g; }
    public int getGols () {return gols; }
}
```

En l'exemple anterior, es defineix la classe *Futbolista* per extensió (herència) de la classe *Jugador* prèviament definida. Això vol dir que qualsevol objecte de la classe derivada, qualsevol futbolista, tindrà dos atributs (nom i gols) i els quatre mètodes definits. Observa que per a això es fa ús de la paraula reservada **extends**.

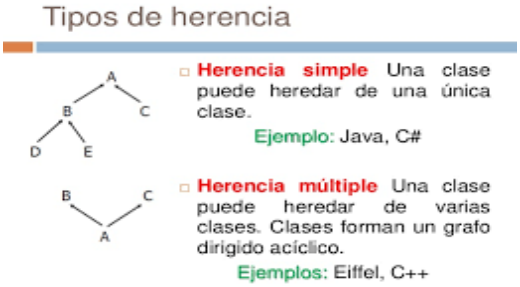
Una cosa fonamental a tenir en compte, per a utilitzar l'herència en la definició de les nostres classes, és que entre la classe derivada i la classe base ha d'existir una relació del tipus "**és un**" en el sentit indicat. És a dir, un futbolista és un jugador (així és), encara que un jugador no ha de ser necessàriament un futbolista, pot ser un jugador d'un altre esport o joc. Altres exemples de classes que podrien complir aquesta relació podrien ser:

CLASSE BASE	CLASSE DERIVADA
Animal	Elefant
Dispositiu	Impressora
Electrodomèstic	Televisor

¿Hi hauria una relació d'herència entre les classes EquipFutbol i Futbolista? La resposta és clara: no. Un equip de futbol contindrà objectes futbolistes, però no és un futbolista (i a la inversa, un futbolista no és un equip).

L'herència, com ja hem dit, és una de les característiques fonamentals i, per tant, present en tots els llenguatges orientats a objectes. En el cas de Java, a més presenta una característica important que el diferencia d'altres llenguatges d'aquest paradigma: **no** permet l'herència múltiple. Això el que vol dir és que una classe només pot fer *extends* directament d'una altra classe (classe base), no de dos o més com, per exemple, sí que ho permet C++. Sí que és possible que una classe C herete d'una classe B que, al mateix temps, haja heretat d'una altra classe A. Podem tenir quants nivells d'herència vulguem, però cada classe derivada hereta directament només d'una classe base (encara que indirectament, estiga heretant per tant de les classes que estigan en els nivells d'herència superiors).

Això es va fer així pels problemes que sabem ocasiona l'herència múltiple en altres llenguatges. Per contra, sí que es permet en Java la implementació múltiple de més d'una interfície.



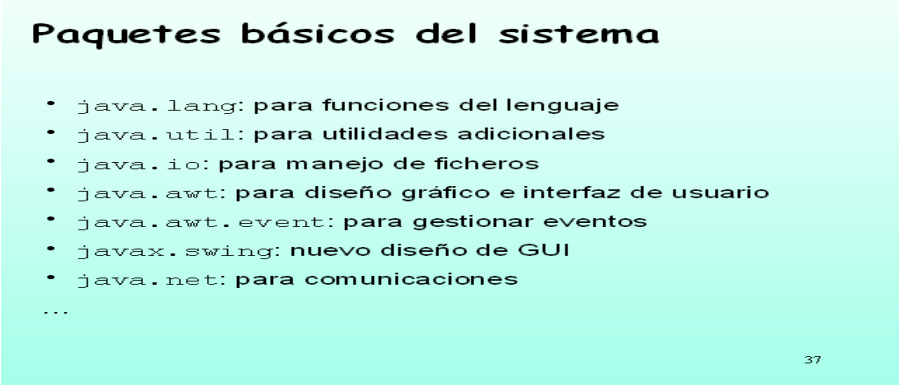
Si la paraula reservada *this* ens serveix per fer referència a la instància actual de l'objecte, Java també té la seva paraula reservada per a fer referència a la instància de la classe base corresponent. Aquesta paraula és **super**, la qual ens serveix per fer el mateix amb la classe pare del nostre objecte (la classe de la que es fa l'*extends*); a més ens permet accés als constructors (amb *super()*) i als mètodes i atributs de la classe de la que hereta:

```java
    public class Fitxer {
            String nom;
            String ruta;

            public Fitxer(String nom) {
                    this.nom = nom;
                    this.ruta = "";
            }
            public Fitxer(String nom, String ruta) {
                    this(nom);
                    this.ruta = ruta;
            }

            public String getNom() {
                    return nom;
            }
            public void setNom(String nom) {
                    this.nom = nom;
            }
            public String getRuta() {
                    return ruta;
            }
            public void setRuta (String ruta) {
                    this.ruta = ruta;
            }

            public String toString () {
                    return ruta + nom;
            }
    }

    public class Imatge extends Fitxer {
            private int ample;
            private int alt;

            public Imatge (String nom, String ruta) {
                    super (nom, ruta); // crida a constructor de la classe base
                    this.ample = 1920;
                    this.alt = 1080;
            }
            public Imatge (String nom, String ruta, int ample, int alt) {
                    super(nom, ruta);
                    this.ample = ample;
                    this.alt = alt;
            }

            public int getAmple() {
                    return ample;
            }
            public void setAmple(int ample) {
                    this.ample = ample;
            }

            public int getAlt() {
                    return alt;
            }
            public void setAlt(int alt) {
                    this.alt = alt;
            }

            public String toString () {
                    // amb super cridem al toString() de la classe base
                    return super.toString() + ", dimensions:" + this.ample + "X" + this.alt;
            }
}
```

En aquest exemple veiem com al mètode *toString()* fem servir *super* per fer la crida al mètode del pare. Encara que en aquest cas estem fent la crida al mètode amb el mateix nom del pare, no hi ha res que ens impedisca cridar a un altre dels seus mètodes o a tots els que vulguem.

2 - MODIFICADORS D'ACCÉS

De manera similar a com succeix en el sistema de fitxers, en què els fitxers s'agrupen en directoris, les classes Java s'agrupen en paquets. Quan no fem ús de la paraula reservada *package* en la definició d'una classe, el directori en el qual creem la classe actua com a paquet per defecte per a la classe. Quan la nostra classe es pretenga utilitzar des d'altres classes, podem limitar els accessos a ella segons els 4 modificadors d'accés que veurem en aquest apartat.



La imatge anterior enumera algun dels paquets principals de classes incloses en el JDK. El primer d'ells, *java.lang*, conté les classes principals de el llenguatge, com *String*, que no requereixen incloure's amb *import*. La resta de paquets sí que ho requereixen.

Apart dels paquets propis del JDK, nosaltres també podem crear els nostres propis paquets tenint en compte que:

- un paquet també pot contenir subpaquets
- Java manté la seva biblioteca de classes en una estructura jeràrquica
- quan ens referim a una classe d'un paquet, el seu nom queda definit per la successió de paquets jeràrquica a la qual pertany, per exemple *java.io.File*, excepte si ja vam fer el corresponent *import*.

Els modificadors d'accés disponibles en Java són, de menys a més restrictius:

- public
- protected
- (default, per defecte), quan no indiquem res
- private

Per definir les classes utilitzem public i *default*, mentre que per als seus membres (atributs o mètodes) podem utilitzar tots ells.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	X
no modifier	Y	Y	X	X
private	Y	X	X	X

Així, per exemple qualsevol altra classe del mateix paquet podrà accedir a qualsevol atribut definit com *protected*, però no si es va definir *private*. Com es pot veure, *public* ho fa accessible sempre, mentre que *private* només des de la pròpia classe.

3 - LA CLASSE OBJECT

Totes les classes Java comparteixen alguns elements en comú. La millor manera de dur a terme això és mitjançant una classe base, en l'arrel de l'herència, que defineisca allò que ha d'incloure qualsevol objecte de la resta de classes. La classe *Object* és aquest arrel jeràrquic de Java.

Això suposa que totes les classes, fetes per nosaltres o no, hereten d'ella (les seves instàncies són objectes *Object*) i, per tant, hereten els seus mètodes. Entre ells podem destacar:

protected Object clone ()

Crea i retorna una còpia de l'objecte. Es deixa a l'alumne llegir i interpretar en la documentació fins a quin punt l'objecte creat és independent de l'objecte original.

boolean equals(Object obj)

Avalua si un objecte passat com a paràmetre és igual a aquest. Realment no compara el contingut intern sinó les adreces en memòria.

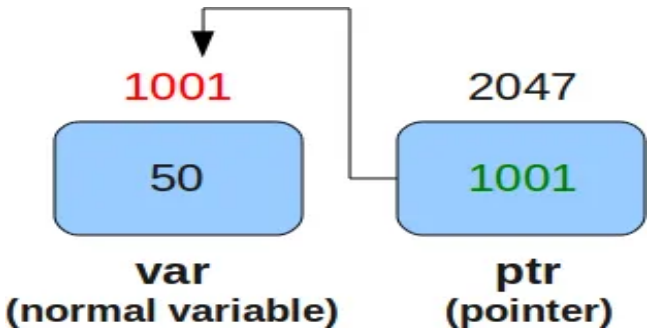
protected void finalize()

Invocat pel recol·lector d'escombraries (*garbage collector*, component de la màquina virtual) sobre un objecte quan aquell determina que no hi ha més referències al mateix objecte.

String toString ()

Retorna en un String una representació del codi *hash* de l'objecte.

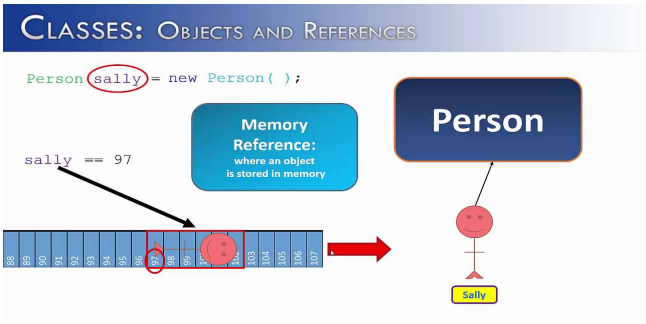
4 - REFERÈNCIES



En Java anomenem referències als tipus de dades que en altres llenguatges, com C, són coneguts com punters o dades indirectes. Es tracta de dades que contenen una adreça de memòria. Aquesta adreça indica on un objecte és allotjat en memòria.

```
rectangle r = new rectangle();
```

Quan creem un objecte d'alguna classe instanciable, el que fins ara en dèiem l'objecte (*r* a l'exemple anterior), realment és una referència. Cada objecte és referenciat per una, o vàries, referències.



En general, no només en Java, en la programació modular els paràmetres a les funcions poden ser passats de dues formes:

-**per valor**. En aquest cas, la funció cridada treballa amb una còpia de la dada original. Si modifiquem la còpia des del codi de la funció cridada, el paràmetre formal, no canvia el paràmetre actual, és a dir la dada original.

-**per referència**. En aquest cas, la funció treballa directament amb la dada original. Si aquest és modificat en la funció estem canviant el paràmetre actual també.

Mentre que altres llenguatges, com C, permeten passar una mateixa dada per valor (directament la dada) o per referència (a través d'un punter), en Java **tots els valors de tipus bàsics o primitius es passen sempre per valor**, mentre que els objectes es passen per referència (a través de la seva referència), tret de comptades excepcions com la classe *String*. Això permet que els canvis realitzats en una funció sobre un objecte es reflecteixen directament en l'objecte original, mentre que en les dades bàsiques no.

En general, assignar una referència a una altra realitza una **còpia superficial** de l'objecte. L'efecte que s'aconsegueix és tenir una referència més al mateix objecte:

```
rectangle r1 = new rectangle(2,3);
rectangle r2 = r1;  // r2 es una còpia superficial de r1
r2.setAmple(4);
System.out.println(r1.getAmple()); // mostraria 4, no 2
```

En aquest cas, r2 i r1 són les dues referències al mateix objecte. Puc accedir al mateix objecte a través d'una o altra referència, és a dir, si modifique el contingut de l'objecte a través d'una de les referències estic modificant el contingut de l'altra, ja que ambdues són referències al mateix objecte.

Per contra, quan fem servir *clone()* d'*Object*, o un constructor de còpia, fem una còpia **en profunditat**, és a dir, crea un nou objecte clon del primer amb vida independent (si modifique un, no es modificarà l'altre):

```
rectangle r1 = new rectangle(2,3);
rectangle r2 = new rectangle(r1);
r2.setAmple(4);
System.out.println (r1.getAmple()); // mostraria 2, no 4
```

Realment, la còpia no sempre és tant independent con podem pensar. Per a la classe *rectangle* sí que ho seria, ja que els seus atributs són tots de tipus bàsics. Els atributs d'aquests tipus, els bàsics o primitius, són diferents per a un objecte o un altre; en canvi, aquells atributs de la classe que siguen referències, es

a dir objectes, són copiats per referència (ja que són referències), la qual cosa suposa que l'objecte origen de la còpia i el nou compartirien, en principi, aquests valors i, per tant, no tindrien vida independent. Això és així per a totes les classes, excepte alguna escassa excepció com (altra vegada) la classe String.

5 - CLASSES I MÈTODES ABSTRACTES I FINALS

Una classe abstracta és una classe que no permetrà generar instàncies, és a dir, objectes. Generalment, aquest tipus de classes s'utilitzen com a base d'altres classes, derivades d'elles, evitant crear instàncies de la classe base, només de la classe derivada.

Les classes abstractes generalment contenen algun mètode abstracte, és a dir, mètodes en que el codi està per definir (la pretensió, generalment, és fer-ho en les classes derivades); aquestos mètodes només són prototipats (definits només el seu nom, tipus i llista de paràmetres amb els seus tipus). No obstant això, una classe abstracta pot no tenir cap mètode abstracte. En aquest sentit, l'única implicació és que si la classe conté algun mètode abstracte, necessàriament haurà de ser abstracta, però no al contrari.

Una classe derivada, d'una classe abstracta amb algun mètode abstracte, estarà obligada a implementar el codi restant (prototipat en la classe base), és a dir, definir aquests mètodes abstractes si no vol ser igualment abstracta:

```
public abstract class Figura
{
    protected String color;

    public Figura(String color)
    {
        this.color = color;
    }
            // mètode abstracte: obliga a que la classe siga abstracta
    public abstract double calcularArea();

    public String getColor()
    {
        return color;
    }
}

public class Quadrat extends Figura
{
    private double costat;

    public Quadrat(String color, double costat)
    {
        super (color);
        this.costat = costat;
    }

    public double calcularArea()
    {
        return costat*costat;
    }
}
```

Com es pot observar, el mètode *calcularArea* ha estat definit abstracte en la superclasse abstracta *Figura*, indicant-se només la seva signatura o prototipus

```
public abstract double calcularArea ();
```

D'altra banda, en cadascuna de les subclasses (en aquest cas *Quadrat*, però ho seria també en *Triangle* o qualsevol altra classe derivada) s'ha d'implementar aquest mètode.

Per la seva banda, una classe marcada com *final* no podrà actuar com a classe base d'altres, és a dir, no podrem definir una altra classe fent *extends* d'ella. És per tant, com posar un límit a l'herència. Recorda que l'ús més conegut de *final* és el de definir constants. En aquest cas, aplicat a una classe també té el sentit de mantenir-la constant, és a dir, no permetre fer subtipus que redefineixin el concepte de la classe base:

```
final class A
{
     // mètodes i camps
}
// La següent classe és il·legal.
class B extends A
{
    // COMPILACIÓ-ERROR!
}
```

També els mètodes poden ser marcats com *final*. El seu significat serà que el mètode no pot ser sobreeescrit (és a dir, redefinit) en qualsevol classe derivada de la classe on es va definir el mètode com *final*. Si s'intenta redefinir s'obté un error de compilació.

6 - INTERFÍCIES

Tant la paraula *class* com *interfície* s'utilitzen en Java per crear nous tipus per a la posterior creació d'objectes. Com ja sabem, una classe es compon d'un conjunt d'atributs i mètodes que operen sobre un cert tipus d'entitats.

Per la seva banda una interfície comunament només inclou mètodes abstractes, és a dir, mètodes sense codi definit, només prototipats. També podria contenir atributs, però no és freqüent, ja que **una interfície no pot tenir estat**. Qualsevol atribut que incloguera hauria de ser constant.

Una interfície és sintàcticament similar a una classe, però hi ha una diferència semàntica important entre classe i interfície: una classe es pot instanciar, però una interfície no. Sí que es poden instanciar objectes de classes que implementen la interfície, com vorem a continuació. Una interfície mai pot tenir un constructor, pel motiu anterior i per no haver de inicialitzar atributs. La seva sintaxi general és:

```
interface nomInterficie
{
    tipus varName = valor;
    tipus metode1(llista de paràmetres);
    tipus metode2(llista de paràmetres);
    . . .
}
```

Si una interfície té algun atribut, l'ha de inicialitzar en el moment de la seva declaració. Atès que els seus mètodes són abstractes, una interfície només defineix el que ha de fer una classe en lloc de com ha de fer-ho. Amb la interfície ja creada, pot ser implementada per una o més classes usant com a paraula clau **implements**. Les classes que implementen la interfície hauran de definir tots els mètodes inclosos en la interfície o es generaria un error de compilació:

```
interface figura
{
```

```
    public double area();
}

class rectangle implements figura
{
    ...
    public double area() {return ample*alt; }
}
```

Una altra característica a destacar en les interfícies és el fet de que una classe, com ja hem vist, només pot heretar d'una classe base (recorda, en Java no permet l'herència múltiple) però sí que pot implementar més d'una interfície. La sintaxi, en aquest cas, és mitjançant l'ús de la mateixa paraula reservada *implements* seguida de la llista d'interfícies separades per comes.

A partir de la versió 8 de Java, s'incorporen al llenguatge característiques pròpies del paradigma de programació funcional. Les principal novetats que incorpora el llenguatge són l'ús d'expressions anomenades *lambdes*, així com l'ús de *streams* o fluxes de dades. Les primeres d'elles pretenen resoldre la implementació simple de les denominadas **interfícies funcionals**.

Les interfícies funcionals són interfícies caracteritzades per incloure un únic mètode abstracte. Sí que poden contenir altres mètodes amb implementacions per defecte, però només un d'ells podrà estar sense definir, és a dir, prototipat. Un exemple és *Comparable* i el seu mètode *compareTo* implementat per *String* o *StringBuffer* entre altres classes.

Expressions Lambdes

Donada, per exemple, la interfície *Collection* observa, consultant la documentació, com *ArrayList* o *Vector* són exemples de classes que implementen aquesta interfície. En ella tenim el següent mètode:

```
default boolean removeIf (Predicate<? super E> filter )
```

El paràmetre que accepta es un *Predicate*, el qual és igualment altra interfície. Esta és una interfície funcional, interfície que obliga a implementar només un mètode com ja hem dit:

```
boolean test(T t) // Evalua el predicat per al paràmetre t.
```

Es tracta d'un mètode que comprova si un paràmetre compleix una determinada condició. Per exemple:

```
Vector<String> v = new Vector<String>();

v.add( ...);
...  // afegim més elements al Vector
v.removeIf( s -> s.isEmpty() )   // expressió LAMBDA que elimina elements sense text
```

Observa que fàcil resulta amb l'expressió lambda esborrar aquells elements que estiguen buits. Sense una expressió lambda, hauriem de fer una classe que implementara la interfície *Predicate*, instanciar un objecte i passar-lo com a paràmetre al mètode *removeIf*:

```
class Predicat implements Predicate<String>
{
    public boolean test(String s)
    {
        return s.isEmpty();
    }
}
```

```
Predicat<String> p = new Predicat();
v.removeIf(p);
```

Observa que hem hagut de crear una classe que, molt probablement, no tornarà a ser útil; és a dir, no la tornarem a instanciar. Altra possibilitat hauria segut fer una classe anònima:

```
v.removeIf(new Predicate<String>()
    {
            public boolean test(String s)
            {
                 return s.isEmpty();
             }
    });
```

Com pots veure una classe anónima es aquella que, sense nom, defineix la classe al mateix temps que la instancia.

Però ¿què és una expressió lambda i com es defineix? Podem dir que una expressió lambda és una funció sense nom. Les expressions lambda proporcionen una forma ordenada d'implementar una classe que normalment té una sola funció (la classe que implementa una interfície de només una funció, anomenada interfície funcional), facilitant la implementació del mètode en qüestió i evitant fer l'esforç de definir el mètode amb tota la seua sintaxi completa. A més, les expressions lambda ajuden a l'execució més eficient de programes Java que treballen amb fils (*threads*) en màquines de múltiples nuclis o en paral·lel.

El **format de definició** per a una expressió lambda és:

> paràmetres -> codi de la funció

La fletxa separa els paràmetres d'entrada del cos o codi de resposta. En molts casos, el cos és curt, inclús pot ser només una línia de codi. Vegem un exemple, el d'una funció que no pren paràmetres i retorna el número 44:

> () -> {return 44; }

O també una expressió lambda que retorne la suma de dos enters:

> (int x, int y) -> {return (x + y); }

En molts casos, Java pot inferir el tipus de paràmetres, i en aquest cas podem deixar d'especificar el tipus de dades, inclús el *return*. El següent codi és equivalent a l'exemple anterior:

> (x, y) -> x + y

Una vegada implementada la expressió lambda, per al seu ús es defineix una referència a la interfície funcional assignant-li la expressió lambda. En l'exemple anterior de *Predicate* seria:

> Predicate<String> p = s -> s.isEmpty();

En aquest cas evitem generar una nova classe que implemente *Predicate* amb la definició del mètode *test*. Així queda el codi molt més compacte. Inclús no es requereix especificar el paràmetre de tipus *T*, en l'exemple *String*, perquè el compilador de Java el pot deduïr.

Un programa complet que fera ús de la lambda per a la suma anterior seria:

```
interface acumula
{
        public double suma(double x, double y);
}

public class programa
```

```
{
        public static void main(String args[])
        {
                double n1 = 5.1, n2 = 3.4;

                acumula ac = (x,y) -> x + y;
                double resultSuma = ac.suma(n1,n2);
                System.out.println("La suma és " + resultSuma);
        }
}
```

7 - POLIMORFISME

El polimorfisme, qualitat per la qual un objecte pot prendre diverses formes, en la POO permet considerar un objecte com a pertanyent a diferents classes a través de l'herència, les classes abstractes i les interfícies.

Una primera forma de polimorfisme és la referida a funcions o mètodes (polimorfisme de funcions o, també, **sobrecàrrega** de funcions), la qual permet que tinguem en una mateixa classe dos o més mètodes amb el mateix nom, distingint-se en el nombre o els tipus dels paràmetres. Un exemple d'això és la mateixa funció *println* (o també *print*) de la classe *PrintStream*. Això és conegut com *overloading* o sobrecàrrega, i hem de tenir en compte que:

- Els mètodes sobrecarregats han de diferenciar-se en la llista d'arguments obligatòriament.
- Un mètode pot estar sobrecarregat a la mateixa classe o en una subclasse.
- Al sobrecarregar un mètode es poden utilitzar les mateixes o diferents excepcions.
- Els mètodes sobrecarregats poden canviar el tipus de retorn o el modificador d'accés.

Similar, però no igual a la sobrecàrrega, és la **sobreescriptura (overriding, també overwriting)** de mètodes. La sobreescriptura permet modificar el comportament de la classe base en una classe derivada:

```
class A {
    int i, j;
    A (int a, int b) {
        i = a;
        j = b;
    }
    // S'imprimeixen i i j.
    void show() {
        System.out.println ( "i i j:" + i + " " + j);
    }
}

class B extends A {
    int k;
    B (int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // S'imprimeix k sobreescrivint el mètode
    @Override
    void show() {
        super.show();
        System.out.println( "k:" + k);
```

```
    }
}
```

Vegem un altre exemple similar. Creem dues classes *Gat* i *Gos*, que hereten de la superclasse *Animal*. La classe *Animal* té el mètode *parla()*, que es sobreescriu de forma diferent en cada subclasse (els gats miolen i els gossos borden). Llavors, un altre objecte (un programa, per exemple) pot enviar el missatge d'emetre el seu so a un grup d'objectes *Gat* i *Gos* a través d'una referència *Animal*, fent així un ús polimòrfic d'aquests objectes respecte del missatge parlar:

```
class Animal {
    public void parla() {
        System.out.println( "Grr ...");
    }
}
class Gat extends Animal {
    @Override
    public void parla() {
        System.out.println( "Miau");
    }
}
class Gos extends Animal {
    @Override
    public void parla() {
        System.out.println( "Guau");
    }
}
```

Com tots els objectes Gat i Gos són objectes Animals, podem fer el següent :

```
public static void main (String [] args) {
    Animal a = new Gos();
    Animal b = new Gat();
    a.parla(); // mostraria "Guau"
    b.parla(); // mostraria "Miau"
}
```

Creem dues variables de referència de tipus *Animal* i les fem apuntar als objectes *Gat* i *Gos*. Al cridar a *parla()* en cada cas és el tipus d'objecte, i no el de la referència, el que determina quin mètode és l'anomenat.

Selecció dinàmica de mètodes. L'anterior exemple de les classes *Gos* i *Gat* és un cas de selecció dinàmica de mètodes. Es tracta d'un mecanisme mitjançant el qual la crida a un mètode sobreescrit es resol durant el temps d'execució i no en el de compilació.

Quan es crida a un mètode **sobreescrit** a través d'una referència a una superclasse, Java determina quina versió d'aquest mètode s'ha d'executar en funció del tipus d'objecte a què es fa referència en el moment de fer-se la trucada.

Quan es fa referència a diferents tipus d'objectes, podem fer referència a diferents versions del mètode sobreescrit. En aquest cas, el que determina la versió del mètode que serà executat és el tipus d'objecte a què es fa referència, i no el tipus de variable de referència. Per tant, si una superclasse conté un mètode sobreescrit (*overriding* o *overwriting*) per una subclasse, quan es faça referència a diferents tipus d'objectes, mitjançant una variable de referència d'una superclasse, s'executaran diferents versions del mètode.

Imaginem aquest altre exemple:

```java
class A {
    void hola() {
        System.out.println("Estic en A");
    }
}

class B extends A {
    @Override
    void hola() {
        System.out.println("Estic en B");
    }
}

class C extends A {
    @Override
    void hola() {
        System.out.println("Estic en C");
    }
}

class Saluda {
    public static void main (String args []) {
        A a = new A();
        B b = new B();
        C c = new C();
        A r; // Obtenció d'una referència a un objecte A
        r = a;
        r.hola();
        r = b;
        r.hola();
        r = c;
        r.hola();
   }
}
```

L'eixida del programa seria

```
Estic en A
Estic en B
Estic en C
```

8 - CONVERSIONS ENTRE OBJECTES (CASTING)

Donada la següent relació de classes:

```java
class Esfera {
    double radi;

    double superficie() {
        return 4*Math.PI*radi*radi;
    }
}

class Planeta extends Esfera {
```

```
    int numSatelits;
}
```

la següent declaració seria vàlida

```
Esfera e = new Planeta();
```

Convé diferenciar els conceptes de tipus estàtic d'una variable amb el tipus dinàmic. Anomenem **tipus estàtic** al tipus amb el qual es declara una variable referència, i **tipus dinàmic** al tipus de l'objecte a què apunta aquesta referència. En l'exemple mostrat, s'ha declarat una variable de tipus *Esfera* (tipus estàtic), però, l'objecte creat és de tipus *Planeta* (tipus dinàmic).

Com ha quedat reflectit, és possible declarar una referència d'un tipus estàtic determinat i instanciar un objecte, no d'aquesta classe, sinó d'una subclasse de la mateixa. És a dir, és possible que el tipus dinàmic siga una subclasse del tipus estàtic. Aquesta manera de referenciar objectes en els quals el tipus referència correspon a una superclasse de l'objecte referenciat és el que anomenem **conversió cap amunt** (*upcasting*).

Quan s'utilitza la conversió cap amunt és important tenir en compte una limitació: si creem un objecte d'un tipus determinat i fem servir una referència d'una superclasse per accedir a la mateixa, a través d'aquesta referència únicament podrem "veure" la part de l'objecte que es va definir en la superclasse. Dit d'una altra manera, el tipus estàtic limita la interfície, és a dir, la visibilitat que es té dels atributs i mètodes d'un objecte determinat.

Prenguem com a exemple les següents classes:

```
class A {
    public void m1() {
        // Aqui es fa alguna cosa ...
    }
}

class B extends A {
    public void m2() {
        // Aqui es fa una altra cosa ...
    }
}
```

Suposem ara que en algun punt del nostre programa es creen els següents objectes:

```
B obj1 = new B (); // Tipus estàtic: B Tipus dinàmic: B
A obj2 = new B (); // Tipus estàtic: A Tipus dinàmic: B
B obj3 = new A (); // Tipus estàtic: B Tipus dinàmic: A. ¡ERROR!
```

En aquest cas, a partir de la referència *obj1* es té accés tant al mètode *m1()* com a *m2()*. No obstant això, amb la referència *obj2*, tot i que referència a un objecte de tipus *B*, únicament es té accés als mètodes definits en *A*, ja que el tipus estàtic imposa aquesta limitació. La referència *obj3* no és possible, ja que si es permitera, s'hauria de tenir accés a elements que realment no han estat creats (com *m2()*).

A tall de resum direm que el tipus estàtic dictamina QUÈ operacions poden realitzar-se, però és el tipus dinàmic el que determina COM es realitzen. Aquesta característica té una enorme transcendència en la programació orientada a objectes ja que, com es veurà més endavant, permet realitzar certes abstraccions sobre els tipus de dades (classes) amb els quals es treballa.

En la conversió cap amunt es guanya generalitat però es perd informació sobre el tipus concret amb el qual es treballa i, conseqüentment, es redueix la interfície. Si es vol recuperar tota la informació del tipus amb què es treballa (és a dir, recuperar l'accés a tots els membres de l'objecte) serà necessari canviar el

tipus de la referència (tipus estàtic) perquè coincideisca amb el tipus real de l'objecte (tipus dinàmic) o, al menys, amb algun altre tipus que permeta l'accés al membre de l'objecte desitjat. Aquesta conversió a un tipus més específic és el que es coneix com conversió cap avall i es realitza mitjançant una operació de càsting.

```
Figura f = new Cercle (); // Conversion cap amunt
Cercle c = (Cercle) f; // Conversion cap avall mitjançant canvi de tipus (càsting)
// Ara mitjançant c podem accedir a TOTS els membres
// de Cercle. Amb f NOMÉS podem accedir a membres de Figura
```

També es pot fer un canvi de tipus temporal sense necessitat d'emmagatzemar el nou tipus en una altra variable referència:

```
Figura f = new Cercle (); // Conversion cap amunt
((Cercle) f) .radi = 1; // Accés a membres de la subclasse
// A continuació f segueix sent de tipus Figura
```

En aquest cas mitjançant la sentència *((Cercle) f)* es canvia el tipus de *f* únicament per a aquesta instrucció, el que permet accedir a qualsevol membre definit en *Cercle* (per exemple, a l'atribut *radi*). En sentències posteriors, *f* seguirà sent de tipus *Figura*.

Les conversions de tipus s'han de fer amb precaució: la conversió cap amunt sempre és segura (per exemple un cercle, amb tota certesa, és una figura, de manera que aquest canvi de tipus no generarà problemes), però la conversió cap a baix pot no ser segura (una figura pot ser un cercle, però també un rectangle o qualsevol altra figura). Dit d'una altra manera, en la conversió cap amunt es redueix la "visibilitat" que es té de l'objecte (es redueix la interfície) mentre que en la conversió cap avall s'amplia.

9 - NIDIFICACIÓ DE CLASSES

Una classe imbricada o nidificada és una classe que és membre d'una altra classe. No és una cosa que es faça habitualment, i prèviament hauríem de preguntar-nos si realment no és millor optar per fer classes independents, però en ocasions ens pot interessar fer-ho així. Anem a distingir entre dos tipus d'aquestes classes: **niades internes** i **niades estàtiques**. Comencem per les primeres.

La nidificació d'una classe té per objectiu afavorir l'encapsulament. Una classe imbricada es diu que és interna si es declarada dins d'una altra classe però fora de qualsevol mètode de la classe contenidora. Pot declarar amb qualsevol dels modificadors: private, protected o public.

Una característica fonamental és que una classe interna té accés a tots els atributs de la classe que la conté. Vegem un exemple:

```
// exemple de classe nidificada no estàtica

class A
{
        private B b;
        public A() { b = new B(); /* b.num = 0; */ }
        public void setB(B b) { this.b = b;}
        class B
        {
                private int num;
        }
}

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

```
    {
            A a = new A();
            //B b = new B();    // produiria error de compilació
            a.setB( a.new B() );
            A.B b = a.new B();    // necessite la instància de la classe exterior
    }
}
```

Observa com no permetria crear un objecte de la classe B directament, però sí amb una instància de A i, o bé, el corresponent *setter* o utilitzant la notació A.B.

Però en Java també podem definir classes internes amb el modificador *static*, són les CLASSES **niades estàtiques**. En aquest cas la classe interna es comporta com una classe normal de Java amb l'excepció que es troba dins d'una altra. Per crear un objecte de la classe interna hem d'utilitzar la següent sintaxi:

```
class A
{
        private B b;
        public A() { b= new B(); b.num = 5;}
        public void setB(B b) { this.b = b;}
        // ara la classe és estàtica
        static class B
        {
                private int num;
        }
}

class nidificacio2
{
        public static void main(String args [])
    {
                A a = new A();
                //B b = new B();    // continua produïnt error de compilació
                A.B ab = new A.B();
    }
}
```

Observa com ara creem instàncies de la classe amb la notació **A.B**.

10 - WRAPPERS

Anomenem *wrappers* o classes embolcall a les classes que ens permeten treballar amb dades bàsiques com objectes. Exemples d'aquestes classes són *Integer*, *Double*, *Character* i *Boolean*. Consulta-les en la documentació del JDK. Anem a prendre com a exemple la classe *Integer*. La descripció que es fa d'aquesta classe en la documentació és:

> "La classe Integer envolta un valor del tipus bàsic int en un objecte. La classe, a més, proveeix diferents mètodes de conversió a String, així com altres mètodes útils quan treballem amb enters."

Bàsicament, la classe inclou, entre constants, un únic atribut variable que és el valor de tipus *int* i li proveeix d'un conjunt de mètodes de conversió. Entre ells anem a destacar:

int intValue()

Retorna el valor contingut com tipus bàsic.

static int parseInt(String s)

Mètode estàtic que retorna el sencer, com a tipus bàsic, contingut en l'objecte String.

static String toBinaryString(int i)

Retorna la representació en binari com String de l'sencer passat com a paràmetre.

static String toHexString(int i)

Retorna la representació en hexadecimal com String de l'sencer passat com a paràmetre.

static String toOctalString(int i)

Retorna la representació en octal com String de l'sencer passat com a paràmetre.

String toString()

Retorna un String que representa el valor sencer contingut en l'objecte.

static String toString(int i)

Mètode estàtic que retorna un String que representa el valor sencer passat com a paràmetre.

static Integer valueOf(int i)

Mètode estàtic que crea i retorna un Integer que representa el valor enter, de tipus bàsic, passat com a paràmetre.

static Integer valueOf(String s)

Mètode estàtic que crea i retorna un Integer que representa el valor enter passat com a paràmetre String.

11 - LA CLASSE DATE

https://www.unixtimestamp.com/ca/index.php

La classe *java.util.Date* representa un instant específic en el temps, amb precisió de milisegons, indicant el temps transcorregut des del primer instant de l'any 1970 (la mitjanit de l'1 de Gener). En tots els mètodes de la classe (alguns marcats com obsolets) que accepten o retornen valors d'anys, mesos, dies, hores, minuts i segons s'han de tenir en compte les següents consideracions:

- un mes es representa amb un enter entre 0 i 11; Gener és el 0 i 11 és Desembre.
- un dia del mes es representa amb un enter entre 1 i 31.
- una hora es representa amb un enter entre 0 i 23.
- els minuts són valors entre 0 i 59, mentre que els segons estan entre 0 i 61. Els valors 60 i 61 estan introduïts per al seu ús en segons intercalars (no els utilitzarem, per a més informació acudeix a la wikipedia o fes alguna recerca en Internet).
- en tots els casos, els paràmetres passats als mètodes no necessàriament han d'estar en aquests rangs; per exemple, una data podria indicar-se com 32 de Gener i seria interpretada com 1 de Febrer.

Una altra classe útil serà *Calendar*. Aquesta és una classe que ens permet gestionar dates d'una manera diferent de com ho fèiem amb *Date*. Es recomana veure la seva documentació en l'API. Tenim dues maneres de crear un objecte *Calendar*:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
Calendar data1 = Calendar.getInstance();
data1.set(2022,6,3);
Calendar data2 = new GregorianCalendar(2022,7,5);
```

La primera forma és usant *Calendar.getInstance()*. ¡Important!: no podem fer *new Calendar();* per tractar-se d'una classe abstracta. Després de crear-lo, fem servir el mètode *set* per fixar els valors que volem, valors d'any, mes i dia. La segona forma és utilitzant una altra classe anomenada *GregorianCalendar* que és filla de *Calendar* i és instanciable.

En Calendar, podem modificar cada element de la següent manera:

```
Calendar data1 = Calendar.getInstance();
data1.set(Calendar.YEAR, 2014);
data1.set(Calendar.MONTH, 10);
data1.set(Calendar.DATE, 20);
```

Simplement, ens cal indicar quin camp s'ha de modificar (any, mes, dia, etc) i el valor a modificar. Per al camp, necessitem usar les constants de *Calendar*, pots veure-les en l'API.

Podem afegir dies, mesos, anys, hores, etc a una data en concret. Per a això, hem de fer servir el mètode *add*; indiquem el camp a afegir i la quantitat que li afegim. Per exemple:

```
Calendar data1 = Calendar.getInstance();
System.out.println( "Data d'avui:" + data1.getTime());
data1.add(Calendar.DATE, 2);
System.out.println( "Data 2 dies més tard:" + data1.getTime());
```

També podem restar-li si el número és negatiu.

Es deixa a l'alumne la tasca de recerca d'altres classes útils per al treball amb temps i dates, com són les classes *LocalDate*, *LocalTime*, *LocalDateTime*, *Clock*, *Instant*, etc.

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
Instant instant = clock.instant();
Date legacyDate = Date.from(instant);
```