Programació multimèdia
i dispositius mòbils

6. Programació Orientada a Objectes

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2021-22

Continguts

1 Programació orientada a objectes

1.1 Objectes

- Un objecte és una entitat amb certes propietats i determinat comportament.
- En termes de POO, les propietats es coneixen com **atributs**, i el conjunt de valors d'aquestes determinen l'**estat** de l'objecte en un determinat moment.
- El comportament, ve determinat per una sèrie de funcions i procediments que anomenem **mètodes**, i que modifiquen l'estat de l'objecte.
- Un objecte tindrà a més un nom pel que s'identifica.

1.2 Classes

- Una classe és una abstracció d'un conjunt d'objectes, i un objecte ha de pertànyer necessàriament a alguna classe.
- Les classes defineixen els atributs i mètodes que posseïran els objectes d'aquesta classe.
- Un objecte es diu que és una **instància** d'una classe.
- **Exemple**
  - Una classe "Persona", que te dos propietats: el nom complet i la data de naixement, i un mètode que calcula l'edat en funció de la data de naixement.
  - D'aquesta classe tenim dos instàncies: l'objecte Josep, amb any de naixement 1978 i l'objecte Joan amb any de naixement 1981.
  - Quan s'utilitze el mètode per calcular l'edat, tot i que serà el mateix per a tots els elements de la classe, el resultat serà diferent per a cada objecte, ja que els seus atributs són diferents.

Veiem la forma genèrica de definir una classe en Java:

```
/* Definició d'una classe*/

[modeAccés] class nomClasse {

    /* Bloc de definició d'atributs */

    [nivellAccés] tipus atribut_1;
    …
    [nivellAccés] tipus atribut_n;

    /* Definició del Constructor    */
```

```
    nomClasse (llista_de_paràmetres) {
        // Inicialització dels atributs, etc.
    }

    /* Definició dels Mèdotes     */

    [nivellAccés] tipus mètode_1 (llista_de_paràmetres) {
        // Cos del mètode
    }
    …
    [nivellAccés] tipus mètode_m (llista_de_paràmetres) {
        // Cos del mètode
    }
}
```

1.2.1 Modes d'accés de la classe

Es tracta de combinacions entre:

- **public / (res)**:
  - ***public***: Classe pública, accessible des de qualsevol altra classe. Només pot haver una classe pública en cada fitxer font (ext .java), i ha de tindre el mateix nom que el fitxer.
  - Si no indiquem res, la classe només podrà ser accedida des del mateix paquet.
- **abstract / final**: S'utilitzen quan fem ús de l'herència.
  - ***abstract***: La classe no permet la instanaciació d'objectes, sinò la definició d'altres classes que implementen mètodes que aquesta defineix com a abstractes.
  - ***final***: Classe que implementa els mètodes abstractes, i de la que no es podran crear subclasses.

1.2.2 Nivell d'accés a atributs i mètodes

Els atributs i mètodes tenen un tipus de dada (el tipus de dada de l'atribut, i el tipus de dada que retorna, en cas dels mètodes), així com un nivell d'accés, que determina qui pot o no accedir a ells. Aquest nivell d'accés pot ser:

- **public**: Pot accedir-se des de qualsevol lloc.
- **protected**: Permet l'accés des del mateix paquet de la classe i des de les seues subclasses.

- ***[no indicar res]***: Permet l'accés des del mateix paquet de la classe.
- **private**: Només hi pot accedir la pròpia classe.

Algunes consideracions:

- Per regla general, els atributs solen establir-se com a privats, i s'utilitzen mètodes específics (getters i setters), per tal de consultar o establir els valors d'aquells atributs als que volem donar accés de forma controlada.
- Els mètodes serveixen per comunicar-nos amb els objectes i interactuar amb ells, a través de missatges, que no és més que la invocació als seus mètodes.
- El mètode constructor s'executa en crear un objecte (amb new), i s'usa per inicialitzar l'estat de l'objecte. Tota classe hauria de tindre, com a mínim, un constructor.

1.2.3 Nivells d'accés o modificadors de la visibilitat en Kotlin

Kotlin presenta pràcticament els mateixos nivells d'accés que Java, però amb les seues peculiaritats:

- **public**: Qualsevol classe, funció, propietat, intefície o objecte públics poden ser accedits des de qualsevol lloc. És el modificador per defecte.
- **private**: Si s'aplica a funcions de nivell superior, interfícies o classes, significa que només poden ser accessibles des del mateix fitxer. Si s'aplica a una funció o propietat d'una classe, objecte o interfície, aquesta només serà visible per als membres de la mateixa classe, objecte o interfície.
- **protected**: Només pot aplicar-se a propietats o funcions dins una classe, objecte o interfície, i només seran accessibles des de la pròpia classe i les subclasses.
- **internal**: Quan treballem en projectes amb mòduls (Gradle o Maven), si definim una classe, objecte, interfície o funció del mòdul con a *interna*, només serà accessible des del propi mòdul.

1.3 Treballant amb classes i objectes

- Creació d'un objecte

```
NomClasse NomObjecte = new NomClasse(llista_paràmetres_inicialització);
```

- Accés a un objecte

```
// Consulta d'un atribut (forma no recomanada)
Tipus variable = NomObjecte.atribut;

// Assignació d'un atribut (forma no recomanada)
```

```
NomObjecte.atribut=valor;

// Accés a un mètode genèric
TipusRetorn variable=NomObjecte.mètode(llista_paràmetres);

// Accés a atributs mitjançant mètodes
Tipus Variable = NomObjecte.GetAtribut();
NomObjecte.SetAtribut(valor);
```

- Referències a atributs de classe.
  - `this` és una referència al propi objecte, que es pot utilitzar des de dins.
  - Si no hi ha conflicte, podem utilitzar el nom definit a la classe sense problemes.
  - En cas que, per exemple, un paràmetre d'entrada tinga el mateix nom que un atribut de la classe, o que definim una variable local a un mètode amb el mateix nom que l'atribut de la classe, sí que caldrà fer ús del `this` per tal de referenciar el paràmetre de la pròpia classe.
- Eliminació d'objectes
  - Tasca del *Garbage Collector* o recol·lector de fem.
- Encapsulació automàtica amb VS Code:
  - VS Code, quan declarem una variable privada, ens suggereix la creació dels mètodes Getter i Setter per a la variable.

1.4 Classes i objectes en Kotlin

La forma de declarar una classe en Kotlin és semblant a Java, però amb una sintaxi més compacta i versàtil.

En Kotlin podem generar una classe buïda amb:

```
class nomClasse
```

I per crear un objecte o nova instància de la classe ho faríem de forma similar a Java, però sense l'operador new (en Kotlin no és una paraula reservada):

```
val elMeuObjecte=nomClasse()
```

Fixeu-se que utilitzem el nom de la classe com si invocàrem directament a una funció.

Per tal de generar una classe que ja continga propietats, mètodes, etc, ho podem fer de diverses formes, anem a vore'n unes quantes.

1.4.1 Creació d'una classe genèrica amb el *constructor secundari*

```
class nomClasse {
    [nivellAccés] [var | val] propietat1: TipusPropietat1
    ..
    [nivellAccés] [var | val] propietatN: TipusPropietatN

    constructor(propietat1: TipusPropietat1,...,propietatN:
    ↪ TipusPropietatN){
        this.propietat1=propietat1;
        ...
        this.propietatN=propietatN;
    }

    [nivellAccés] fun metodeX(llista_de_paràmetres): TipusRetorn {
        cos_del_mètode
    }
}
```

Veiem algunes peculiaritats:

- Declarem la classe amb `class`, igual que amb Java.
- La classe té diverses propietats, que poden ser *mutables* (de lectura-escriptura) si es defineixen amb `var` o immutables (només de lectura) si es defineixen amb `val`. Aquestes propietats podran ser públiques (per defecte si no es defineix res), privades o protegides.
- El mètode `constructor` es coneix com ***constructor secundari***, i seria l'equivalent al constructor de Java, que s'invoca en crear l'objecte i realitza les tasques d'incicialització d'aquest.
- Una cosa interessant, és que **no hem de definir mètodes *getters* i *setters* ja que aquests són autogenerats pel compilador de Kotlin** per a les propietats públiques, i ens permetran accedir com si accedirem directament a aquestes. Si la propietat és **mutable** (definida amb `var`), es generarà el **getter** i el **setter** corresponent, mentre que si és **immutable**, només es generarà el **getter**.
- Finalment, els mètodes es declaren com si foren una funció local a l'objecte, especificant de manera opcional el mètode d'accés al davant.

1.4.2 Creació d'una classe genèrica amb el *constructor primari*

Kotlin permet declarar un constructor en el mateix encapçalament de la classe, anomenat *constructor primari*. Com que el bloc que definim entre `{}` després és el bloc de la classe, el bloc de codi corresponent a la inicialització de paràmetres en aquest constructor es fa amb un bloc `init` que s'executa només crear una instància de classe:

```
class nomClasse constructor(propietat1: TipusPropietat1,...,propietatN:
↪  TipusPropietatN) {
    [nivellAccés] [var | val] propietat1: TipusPropietat1
    ..
    [nivellAccés] [var | val] propietatN: TipusPropietatN

    init {
        this.propietat1=propietat1;
        ...
        this.propietatN=propietatN;
    }

    [nivellAccés] fun metodeX(llista_de_paràmetres): TipusRetorn {
        cos_del_mètode
    }
}
```

Una manera d'abreviar més aquesta construcció sería la següent:

```
class nomClasse constructor([var | val ] propietat1:
↪  TipusPropietat1,...,[var | val ] propietatN: TipusPropietatN) {
    [nivellAccés] fun metodeX(llista_de_paràmetres): TipusRetorn {
        cos_del_mètode
    }
}
```

En ella hem utilitzat el *constructor primari*, i hem definit les propietats directament dins d'aquest, precedint-les de `var` o `val` en funció de que siguen mutables o immutables. En aquesta construcció també podem incloure valors per defecte amb `propietatX: TipusPropietatX = "Valor per defecte"`, i en cas que la classe no tinga modificador d'accés o alguna anotació, podem, fins i tot ometre la paraula `constructor`.

1.4.3 Treballant amb objectes

Per tal de crear instàncies de la classe ho farem amb:

```
[ val | var ] objecte=Classe(Valor_propietat_1, ... , Valor_propietat_n)
```

Per tal d'accedir a les propietats, ho farem directament, a través de la sintaxi d'accés a la propietat: `objecte.propietat`, sense necessitat d'invocar explícitament el mètode *getter*. Per tal d'estrablir un valor, sempre que aquest s'haja definit com a mutable (amb `var`) no és necessari el *setter*, sinò que es pot moificar directament amb l'operador =.

1.4.4 Personalització dels getters i setters

Kotlin permet també personalitzar els mètodes accessors, de manera que puguem bé validar valors en els *setters* abans de fer l'assignació o bé convertir o formatar els valors abans de tornar-los amb un *getter*.

Per tal de fer això, haurem de definir les propietats dins el cos de la classe, en lloc e fer-ho a la capçalera del constructor, i tot seguit, definir els mètodes get i set corresponents, seguint la següent sintaxi:

```
class nomClasse (propietat1: TipusPropietat1,..., propietatN:
↪  TipusPropietatN) {
    [val | var ] propietat1:TipusPropietat1
    get(){
        // Accedim la propietat
        // amb 'field', no amb el seu nom
        return valor_retorn
    }

    var propietatN:TipusPropietatN
    set(value){
        // Accedim a la propietat amb 'field'
        // Per al valor a actualitzar sol usar-se value
        // Aci fariem comprovacions amb value
        field=value // Fem el set
    }
    init{...}
}
```

Fixem-nos que així com per a fer un `get` personalitzat la propietat pot ser mutable o immutable, per fer el `set`, aquesta ha de ser necessàriament mutable.

1.4.5 Interoperabilitat amb Java

Com hem comentat, quan definim propietats per als objectes de Kotlin, el compilador genera automàticament els mètodes accessors, de manera que podem accedir a ells directament a través de les propietats.

Quan des de Java utilitzem una classe creada amb Kotlin, per a aquelles propietats que hem definit com a mutables, es generaran els *getters* i *setters* corresponents a la propietat (`getPropietat()` i `setPropietat(valor)`). Al cas que la propietat siga immutable, Kotlin només ens generarà el *getter*.

1.4.6 Exemple

A moded'exemple, anem a veure el projetce Gradle `exemplePersones`. En ell veurem com crearíem la classe *Persona* amb tres propietats, el nom, l'any de naixement i la professió. El nom i l'any de naixement en principi seran immutables, mentre que la professió podrà variar. Les classes *Persona2*, *Persona3* i *Persona4* guarden la mateixa informació, però veiem en elles diferents formes de creació de classes.

Al llarg de l'exemple trobareu diferents comentaris autoexplicatius:

```kotlin
package com.ieseljust.dam.exemplePersones

import java.time.LocalDateTime

/*******************************************************
 * Definició d'una classe amb el constructor secundari  *
 *******************************************************/
class Persona {
    // Definim les propietats
    val nom: String
    val anyNaix: Int
    public var professio: String

    // Definim el constructor
    constructor(nom: String, anyNaix: Int, professio: String = "") {
        this.nom = nom
        this.anyNaix = anyNaix
        this.professio = professio
    }
```

```
    // AltresMètodes
    fun printMe(): Boolean {
        println("$nom ${LocalDateTime.now().year - anyNaix} - $professio")
        return true
    }
}

/*******************************************************
 * Definició d'una classe amb el constructor primari    *
 *******************************************************/
class Persona2 constructor(nom: String, anyNaix: Int, professio: String) {
    // Definim les propietats
    val nom: String
    val anyNaix: Int
    var professio: String

    // Bloc init per inicialitzar els objectes
    // El 'constructor' està a la definició
    // de la pròpia classe
    init {
        //println(nom+anyNaix+professio);
        this.nom = nom
        this.anyNaix = anyNaix
        this.professio = professio
    }
    fun printMe(): Boolean {
        println("$nom ${LocalDateTime.now().year - anyNaix} - $professio")
        return true
    }
}

/*******************************************************
 * Definició d'una classe amb el constructor primari  *
 *******************************************************/
class Persona3 constructor(val nom: String, val anyNaix: Int, var professio:
 ↪  String = "") {
    // No cal definir les propietats ni init
    // ja que aquestes es defineixen directament en el
    // constructor, predecides de var o val

    // Altres mètodes
    fun printMe(): Boolean {
```

```
        println("$nom ${LocalDateTime.now().year - anyNaix} - $professio")
        return true
    }
}

/*******************************************************
 * Definició d'una classe amb el constructor primari i  *
 * Accessors (getter i setters) personalitzats.         *
 *******************************************************/
class Persona4 constructor(nom: String, anyNaix: Int, professio: String) {
    // Cal declarar les propietats fora del `consructor`
    // (com hem fet en Persona2)
    val nom: String
    // I immediatament després de definir cada propietat,
    // generem els mètodes get() o set() com desitgem.
    get() {
        // field fa referència al propi camp (atribut)
        // (en aquest cas el retorna en majúscules)
        return field.toUpperCase()
    }
    var professio: String = ""
    var anyNaix: Int = LocalDateTime.now().year

    // El setter rebrà un 'valor' al què actualitzar
    // la propietat. El tipus d'aquest 'value' s'infereix
    // del tipus de la propietat que anem a modificar.
    set(value) {
        // En aquest cas, comprovem que l'any no siga
        // superior a l'any actual. En eixe cas, llaçariem
        // una excepció.
        if (value> LocalDateTime.now().year) {
            throw IllegalArgumentException("L'any de naixement no pot ser
            ↪  posterior a ${LocalDateTime.now().year}")
        }
        // Accedim a la propietat amb l'alias 'field', i al
        // valor a actualitzar amb 'value'
        field = value
    }
    init {
        this.nom = nom
        this.professio = professio
        this.anyNaix = anyNaix
```

```
    }
    fun
    printMe(): Boolean {
        println("$nom ${LocalDateTime.now().year - anyNaix} - $professio")
        return true
    }
}

/*
Si utilitzàrem només aquest fitxer, de forma autònima i aquesta fora la
 ↪  classe
principal, crearíem la funció main com s'expressa aci baix.
Com que anem a crear una altra classe Java per a que llance
l'aplicació, aquesta funció està comentada.
*/

/*fun main(args: Array<String>) {
    val p1 = Persona("Josep", 1978, "Profe")
    p1.printMe()
    var p2 = Persona2("Paco", 1973, "Profe")
    p2.printMe()
    var p3 = Persona3("Maria", 2013, "Estudiant")
    p3.printMe()
    var p4 = Persona4("Pepica", 2016, "Estudiant")
    p4.printMe()        // Fixeu-bos que el nom ens apareixerà
                        // directament en majúscules!!
    p4.anyNaix = 2050   // Llançarà una excepció!
    p4.printMe()
}*/
```

Fieu-vos que a l'exemple de dalt hem comentat la funció principal `main`, i anem a utilitzar una classe en Java com a classe principal, que utilitze la classe persona definida a dalt:

```
package com.ieseljust.dam.exemplePersones;

public class Persones {
    public static void main(String[] args) {
        Persona p=new Persona("Jose", 1978, "Profe");
        p.printMe();

        Persona2 p2 = new Persona2("Paco", 1973, "Profe");
```

```
        p2.printMe();

    Persona3 p3 = new Persona3("Maria", 2013, "Estudiant");
    p3.printMe();

        Persona4 p4 = new Persona4("Pepica", 2016, "Estudiant");
    p4.printMe();        // Fixeu-vos que el nom ens apareixerà
                           // directament en majúscules!!
    //p4.anyNaix = 2050;   // Llançarà una excepció!
        p4.printMe();

    }
}
```

Podeu construïr l'exemple i veure'n el resultat de l'execució.

1.4.7 Ús de múltiples constructors

En Kotlin podem combinar els constructors primaris i secundaris segons les nostres necessitats.

En principi, hi pot haver només un constructor primari, però tants constructors secundaris com desitgem.

Veiem-ho amb un exemple amb la classe *PersonaMC*. Aquesta classe es troba definida dins els mateixos fonts que l'exemple anterior, però no forma part del projecte, pel que haurem de compilar-lo i llançar-lo directament des del mateix directori on es trobe:

```
@file:JvmName("PersonesMC")

import java.time.LocalDateTime



/*******************************************************
 * Definició d'una classe amb diversos constructors     *
 *******************************************************/

// Deinició de la classe amb constructor primari, que defineix
// dos propietats immutables, nom i anyNaix
class PersonaMC constructor(val nom: String, val anyNaix: Int) {
```

```
    // Ara definim propietats mutables
    var professio="";
    var poblacio="";

    // Ara fem ús de diversos constructors secundaris

    // Constructor secundari que rep nom, anyNaix i professio
    // Ha de fer referència al constructor primari, fent ús de
    // la paraula reservada "this"
    constructor(nom:String, anyNaix:Int, professio:String):this(nom,
     ↪  anyNaix){
        // Al fer referència al constructor primari, ja donem valor a nom i
         ↪  anyNaix
        // Només ens queda professio
        this.professio=professio;
    }

    // Veiem altre constructor on passem també la població
    constructor(nom:String, anyNaix:Int, professio:String,
     ↪  poblacio:String):this(nom, anyNaix){
        // Al fer referència al constructor primari, ja donem valor a nom i
         ↪  anyNaix
        // Només ens queda professio
        this.professio=professio;
        this.poblacio=poblacio;
    }

   fun printMe(): Boolean {
        println("$nom ${LocalDateTime.now().year - anyNaix} - $professio -
 ↪  $poblacio")
        return true
    }
}

fun main() {
    val p1 = PersonaMC("Josep", 1978)
    var p2 = PersonaMC("Paco", 1973, "Profe")
    var p3 = PersonaMC("Maria", 2013, "Estudiant", "Tavernes")

    p1.printMe()
    p2.printMe()
```

```
    p3.printMe()
}
```

Per fer la compilació, des del directori `/src/main/kotlin/com/ieseljust/dam/exemplePersones`, llancem:

```
$ kotlinc PersonaMC.kt
```

I executem la classe *PersonesMC*:

```
$ kotlin PersonesMC
Josep 42 -  -
Paco 47 - Profe -
Maria 7 - Estudiant - Tavernes
```

1.4.8 Objectes en Kotlin

Kotlin permet definir objectes sense que hàgen de ser necessàriament instàncies d'una classe, tal i com podem fer amb javascript.

Els objectes són semblants a les classes i tenen les següents caracteristiques:

- Poden tindre propietats, mètodes i un bloc init,
- les propietats i mètodes poden tindre modificadors de visibilitat,
- No tenen constructors, ja que no són instanciables,
- Poden extendre altres classes o implementar interfícies

De forma genèrica, definiriem un objecte de la següent forma:

```
object nomObjecte {
    [nivellAccés] [var | val] propietat1: TipusPropietat1
    ..
    [nivellAccés] [var | val] propietatN: TipusPropietatN

    init{...}

    [nivellAccés] fun metodeX(llista_de_paràmetres): TipusRetorn {
        cos_del_mètode
    }
}
```

Com veiem, és molt semblant a una classe, només reemplacem la paraula `class` per `object`, i eliminem els constructors.

La utilitat de definir objectes la podem trobar, en aquells casos en què només tindríem una única instància d'una classe, com per exemple l'objecte *Aplicació*, la seua configuració (amb definició de constants, com cadenes de connexió a BD, usuaris, etc) o qualsevol objecte que seguisca un patró *Singleton*[1]

El compilador de Kotlin converteix els objectes en *classes `final`* de Java, amb un camp estàtic privat `INSTANCE` que conté una instància única (Singleton) de la classe. Si volem que les funcions o propietats de l'objecte siguen definides en aquesta classe com a `static`, haurem d'utilitzar l'anotació `@JvmStatic` al davant.

```
object ObjecteSingleton {
    @JvmStatic fun funcio(): Unit {
        // cos de la funció
    }
}
```

Podeu ampliar tota aquesta informació sobre objectes al següent tutorial:

*Kotlin Desde Cero: Clases y Objetos

I conéixer més coses interessats sobre classes i objectes, com propietats *late-initialized*, *inline*, d'extensió, etc. en:

- Kotlin From Scratch: Advanced Properties and Classes

1.5 Herència i Polimorfisme

Anem a repassar un parell de mecanismes de reutilització de codi, com són l'herència i el polimorfisme, tant des del punt de vista de Java com de Kotlin.

1.5.1 Herència

- Mecanisme de reutilització de codi que permet definir classes a partir d'altres, *heretant* les seues propietats i mètodes o redefinint-los, i donant la possibilitat de definir-ne de nous.

[1] Un Singleton és un patró de disseny de programari que garanteix que una classe només tinga una instància i que aquesta proporcione un únic punt d'accés global. Quan es sol·licita la classe singleton, sempre s'obté la mateixa instància de la classe.

- Diem subclasse a la classe que hereta propietats d'una altra (descendim en la jerarquia de classes), i superclasse a la classe de la qual s'hereten propietats (ascendim en la jerarquia de classes).

```
class nomSubclasse extends  nomSuperclasse {

    /* Bloc de definició d'atributs propis de la subclasse*/

    [nivellAccés] tipus atribut_1;
    …
    [nivellAccés] tipus atribut_n;

    /* Definició del Constructor    */
    nomSubClasse (llista_de_paràmetres) {
        // invoquem el constructor de la superclasse.
        // Super() ha de ser la 1a ordre.
        // Si no la posem, invocarà per defecte el constructor
        // de la classe pare sense passar-li paràmetres.
        super(paràmetres);
        // Inicialització dels nous atributs, etc.

    }

    /* Definició de Mèdotes propis de la subclasse
       o redefinició de mètodes de la superclasse.
    */

    [nivellAccés] tipus mètode_1 (llista_de_paràmetres) {
        // Cos del mètode
    }
    …
    [nivellAccés] tipus mètode_m (llista_de_paràmetres) {
        // Cos del mètode
    }
}
```

1.5.1.1 Definició de l'herència en Java

1.5.2 Polimorfisme

El polimorfisme comporta **diferent comportament d'un mateix mètode segons el context**.

Gràcies al polimorfisme, podem definir un objecte d'una classe i instanciar-lo amb classes descendents.

El polimorfisme pot ser de mètodes o d'atributs:

- **Polimorfisme de mètodes**: Permet enviar el mateix missatge a objectes distings, que tractaran de forma diferent:
  - ***Sobrecàrrega basada en paràmetres:*** Diversos mètodes amb el mateix nom però diferents paràmetres d'entrada. Un exemple molt clar és el propi constructor. Segons la invocació, es refereix a un o altre mètode.
  - ***Sobrecàrrega basada en l'àmbit***: Diferents classes sense relació poden implementar mètodes amb el mateix nom. Segons la classe sobre la que invoquem el mètode s'executarà un o altre.
  - ***Sobreescriptura***: Les classes descendents "sobreescriuen" els mètodes de les classes ascendents.
- **Polimorfisme d'atributs**: Relaxació del sistema de tipat que permet que una referència a una classe accepte adreces d'obejctes de la mateixa classe i les seues descendents:
  - ***Variables polimòrfiques***: Pot referenciar diferents tipus d'objectes amb relacions d'herència.

Algunes consideracions a tindre en compte sobre el polimorfisme **en Java**:

- Per referir-nos a un mètode o atribut de la classe ascendent fem ús de "super":

```
super.mètodeClassePare(paràmetres);
```

- Quan sobreescrivim un mètode en una classe descendent, convé utilitzar **@override** davant, de manera que indiquem al programador que aquest mètode reemplaça el de la classe ascendent, i ajudem al compilador a que ens avise si hi ha canvis en la classe pare, com que eliminem el mètode o li canviem el nom.

```
@override
[nivellAccés] tipus mètodeSobreescrit (llista_de_paràmetres) {
...
}
```

- Per comprovar el tipus de classe d'un objecte, podem:
  - Fer ús del mètode getClass:

```
if (objecte.getClass().getSimpleName().equals("NomClasseAComparar"))
 ↪  {...}
```

  - Fer ús de l'operador instanceof:

```
if (objecte instanceof NomClasseAComparar){...}
```

1.5.2.1 Herència en Kotlin

El principal aspecte a tindre en compte quan pensem en herència en Kotlin, i a diferència de Java, és que les classes es defineixen com a `final` de manera predeterminada.

Una de les bones pràctiques recomanades per l'enginyeria del programari és que totes les classes es definisquen com a finals, i només es deixen coma *obertes* aquelles que sí que van a tindre herència de forma explícita. Kotlin segueix al peu de la lletra esta recomanació, pel que, per defecte no podrem generar una classe a partir d'una altra si la superclasse no es defineix com a `open`.

Contrastant Kotlin i Java en aquest aspecte, podriem dir que:

- En **Java** totes les classes són **obertes**, es a dir, *es pot heretar d'elles*, sempre i quan no s'indique el contrari especificant-les com a **`final`**.
- En **Kotlin** totes les classes són **finals**, és a dir, ***no*** *es pot heretar d'elles*, sempre i quan no s'indique el contrari, especificant-es com a **open**.

Per altra banda, també cal tindre en compte que els mètodes definits a la superclasse, per defecte també són `final`, pel que, per poder-los sobreescriure, cal indicar-ho també amb `open`. El mètode sobreescrit a la subclasse, ara seguirà sent *open*. Si volem evitar açò, podem definir aquest mètode sobreescrit com a `final`.

```
/* Cal definir prèviament la superclasse com a open */

open class nomSuperclasse {
    // Atributs de la classe
    [ val | var ] NomAtribut: Tipus;

    // Constructor secundari
    constructor(Args:Tipus) {
        this.nomAtribut=Param;
```

```
        ...
    }
    // Marquem els mètodes f1 i f2 com
    // a open, per poder-los sobreescriure
    open fun f1(Paràmetres:Tipus):TipusRetorn{...}

    open fun f2(Paràmetres:Tipus):TipusRetorn{...}

}

class nomSubclasse : nomSuperclasse() {

    /* Bloc de definició d'atributs propis de la subclasse*/
    [ val | var ] NomAtribut: Tipus;

    // Constructor secundari
    constructor(Args:Tipus): super(Args) {..}

    // Sobreescrivim el mètode f1 (serà open per defecte)
    override fun f1(Paràmetres:Tipus):TipusRetorn{...}

    // Sobreescrivim el mètode f2, i el declarem final
    override final fun f2(Paràmetres:Tipus):TipusRetorn{...}


    open fun f1(Paràmetres:Tipus):TipusRetorn{...}

    /* Definició de Mèdotes propis de la subclasse
       o redefinició de mètodes de la superclasse.
    */
    ...
}
```

A l'exemple anterior hem vist com utilitzar un constructor secundari. El mecanisme amb constructors primaris seria exactament el mateix.

Algunes coses més a tindre en compte sobre herència en Kotlin:

- Teniu disponible l'operador `is`, per comprovar si una variable és d'un tipus o classe concret: `variable is Classe`; així com la seua negació `!is`: `variable !is Classe`.
- Totes les classe en Kotlin són descendents de la classe Any, que equivaldria ai tipus `Object` de Java. El tipus Any conté tres mètodes: `equals`, `toString`, i `hasCode`. Recordeu que els

tipus bàsics també es representen com a classes, pel que també hereten aquests mètodes.

- Quan es sobreescriu un mètode que té arguments amb valors per defecte a la superclasse, aquests s'han d'ometre de la signatura de la funció a la subclasse. Els valors per defecte seran sempre els indicats a la superclasse, i no es poden modificar als mètodes sobreescrits.

```
open class A {
    open fun f1(i: Int = 10) { ... }
}

class B : A() {
    override fun f1(i: Int) { ... }
}
```

1.6 Atributs i mètodes estàtics o de classe

Els atributs i mètodes estàtics (`static`) són aquells que s'emmagatzemen a la pròpia classe, sent comuns a tots els objectes que s'instancien d'ella.

1.6.1 Atributs de classe o estàtics en Java

- Prenen valor en la mateixa classe, de manera que aquest valor és comú a tots els objectes de la mateixa classe.
- Usos: Definir constants, variables amb valor comú a tots els objectes, comptadors d'objectes de la classe…
- Declaració:

```
static tipus NomAtribut;
...
// Per accedir a l'atribut fem ús del nom de la classe
NomClasse.NomAtribut=valor;
```

1.6.2 Mètodes estàtics

- Poden utilitzar-se directament des de la classe, sense necessitat de crear cap objecte.
- Porten el modificador *static* davant.
- No treballen amb objectes, només amb paràmetres d'entrada i atributs estàtics.

```
class NomClasse(){
    ...
    static Tipus NomMètode(paràmetres){
        ...
    }
    // Per accedir al mètode estàtic des de
    // dins la classe, no indiquem res davant.
    NomMètode(paràmetres);
}
...
// Per accedir al mètode estàtic des de
// fora de la classe fem ús del nom d'aquesta
NomClasse.NomMètode(paràmetres);
```

1.6.3 Classes estàtiques en Kotlin: Objectes complementaris

Kotlin no suporta mètodes o propietats estàtiques, però ens proporciona una alternativa més potent: els objectes complementaris.

Un **objecte complementari** és un objecte que és membre d'un classe (classe *acompanyant*). Podem dir que és un objecte que *acompanya* la classe. Recordeu que a Kotlin es poden definir objectes directament, sense ser instància d'una classe, per tant, tot el què hem comentat sobre la creació d'objectes ens val per als objectes complemetnaris.

El objectes complementaris, a l'igual que els mètodes estàtics en Java, estaran associats a la classe que acompanyen, i no a les instàncies d'aquesta.

Per tal de crear un objecte complementari seguirem una sintaxi semblant a la del següent exemple:

```
class classeAcompanyant {
    companion object {
        fun f1(){
            println("Este és un mètode en un objecte complementari");
        }
    }
}
```

Per utilitzar aquest *mètode estàtic*, farem referència a la classe acompanyant i al mètode en qüestió, igual que fem amb els mètodes estàtics:

```
classeAcompanyant.f1()
```

Al tutorial Kotlin desde cero: Clases y objetos, teniu a l'apartat *7. Objetos complementarios* una explicació més extensa sobre aquest tipus d'objectes, i com crear amb ells classes de tipus *Factory*.

1.7 Interfícies

- Java no permet l'herència múltiple (una classe té més d'una classe pare).
- Les interfícies són la forma que té Java de fer que classes que no tenen una relació jeràrquica tinguen un comportament comú.

1.7.1 Classes i mètodes abstractes

- **Mètode abstracte**: Aquell que es defineix en una classe, però deixa la seua implementació per a les seues classes descendents (si aquestes no defineixen el mètode també com a abstracte.)
- **Classe abstracta**: Aquella que conté mètodes abstractes.
- Amb açò, el concepte d'interfície s'acostaria a una classe abstracta, amb la diferència que una classe sí que pot implementar diverses interfícies.

1.7.2 Definició d'interfície

- És una mena de plantilla per a la construcció de classes.
- Es compon d'un conjunt de declaracions de capçaleres de mètodes sense implementar (com els mètodes abstractes).
- Si conté atributs, aquests seran `static final`, i estaran inicialitzats, ja que funcionaran com a constants.
- Especifica un protocol de comportament per a una o diverses classes, de manera que si coneixem que un objecte implementa determinada interfície, sabem quin és el seu comportament, sense importar com s'ha realitzat la implementació.
- Una classe pot implementar diverses interfícies, proporcionant la declaració i definició de tots els mètodes de cadascuna de les intefícies (o bé declarar-les com a abstractes).
- També s'usen per declarar constants a utilitzar per altres classes.

1.7.3 Declaració d'una interfície

```
public interface IdentificadorInterficie {
    /* Cos de la interfície */

    // Constants
    public static final tipus Identificador = Valor;
    ...
    public static final tipus Identificador_n = Valor_n;

    // Mètodes abstractes
    TipusRetorn NomMètodeAImplementar_1(paràmetres);
    ...
    TipusRetorn NomMètodeAImplementar_n(paràmetres);

}
```

Consideracions:

- Si declarem la interfície com a `public` ha d'estar en un fitxer amb el mateix nom que la interfície (i extensió .java).
- Els mètodes es declaren implícitament com a `public` i `abstract`.
- Les constants incloses en una interfície es declaren com a `public`, `static` i `final`, i s'han d'inicialitzar en la mateixa declaració.
- Quan declarem una interfície, pot utilitzar-se com a tipus de dada, de manera que aquesta es puga reemplaçar per una objecte d'una classe que implemente la interfície.

1.7.4 Implementació d'una interfície en una classe

- Cal utilitzar la paraula reservada `implements` en la declaració.
- La capçalera de la interfície ha d'apareixer tal qual en la declaració.

```
public class NomClasse implements IdentificadorInterfície{

    // Implementació dels mètodes de la interfície
    TipusRetorn NomMètodeAImplementar_1(paràmetres){
        // Implementació
    }
    ...
    TipusRetorn NomMètodeAImplementar_2(paràmetres){
        // Implementació
```

```
    }
}
```

1.7.5 Interfícies en Kotlin

Per definir una interfície en Kotlin ho farem de forma similar a Java:

```
interface IdentificadorInterficie {
    /* Cos de la interfície */

    // Constants
    val Identificador_1:Tipus_1=Valor_1;
    ...
    val Identificador_n:Tipus_n=Valor_n;

    // Mètodes abstractes
    fun NomMètodeAImplementar_1(paràmetres_1:Tipus_1): TipusRetorn_N;
    ...
    fun NomMètodeAImplementar_M(paràmetres_N:Tipus_N): TipusRetorn_N;

}
```

Per tal de definir una classe que implemente la interfície:

```
class NomClasse : IdentificadorInterfície {

    // Implementació dels mètodes de la interfície
    override fun NomMètodeAImplementar_1(paràmetres_1:Tipus_1):
     ↪  TipusRetorn_N{
        // Implementació
    }
    ...
    override fun NomMètodeAImplementar_M(paràmetres_N:Tipus_N):
     ↪  TipusRetorn_N{
        // Implementació
    }
}
```

Alguns aspectes a tenir en compte:

- No utilitzem cap paraula com `implements`, sinò que fem ús dels `:`, com quan definim una subclasse, però sense invocar el constructor `()`.
- Una classe pot implementar tantes interfícies com es desitge, però només pot extendre d'un classe (com en Java)
- S'utilitza, de forma obligada, el modificador `override` per etiquetar els mètodes i propietats que volem redefinir (en Java l'anotació `@Override` és opcional)
- A més dels mètodes, també podem declarar propietats.
- Un mètode de la interfície pot tindre una implementació predeterminada, i ser sobreescrita en les classes que la implementen si es desitja, indicant-ho amb `override`. Si volem utilitzar el mètode de la interfície, podem referir-nos a ella amb `super`.
- Una interfície pot tindre **propietats**, però aquestes **no mantenen l'estat** (a diferència de les classes abstractes). És a dir, podem definir la propietat, però no donar-li valor inicial (com fem en Java amb els atributs amb `static final`). No obstant això, podem tindre mètodes accessors (`set i get`).
- Les propietats també poden redefinir-se amb `override`.
- Quan una classe sobreescriu un mètode amb una implementació predeterminada a la interfície, pot accedir al mètode d'aquesta amb `super`, com si es tractara d'una classe ascentdent.
- En cas que una classe implemente diverses interfícies, i aquestes tinguen mètodes en comú amb una implementació predeterminada, caldrà especificar, si utilitzem `super` a quina interfície fem referència mitjançant la notació `super<Interficie>.metode()`.

Veiem un exemple autoexplicat amb alguns d'aquests conceptes (`interficies/interfaces.kt`):

```
/* Exemple interficies

    * Definirem dues interfícies, amb mètodes
    comuns, i veurem en una classe que implemente les dos
    com especificar al mètode de quina classe ens referim.
*/


interface Interface1 {

    // Mètode abstracte
    fun metode1()

    // Mètode amb implementació
    // predeterminada
    fun metode2(){
        println("Mètode2 de Interface1");
```

```
    }
}

interface Interface2 {
    fun metode2(){
        // Mètode amb implementació
        // predeterminada.
        println("Mètode3 de Interface2");
    }
}

// Ara definim la classe classe1, que implementa
// les dues interfícies
class classe1 : Interface1, Interface2 {

    // Mètode 1 és abstracte, per tant,
    // necessita sobreescriptura.
    override fun metode1 () {
        println("Metode 1 classe1");
    }

    // El segon mètode, realment, no necessitaria
    // sobreescriptura si no volguérem modificar-ne
    // el funcionament.
    // Anem a sobreescriure'l, per veure com
    // accedir als mètodes de les interfícies.
    override fun metode2(){
        println("Mètode2 de classe1");
        // Amb super podem accedir a la implementació
        // per defecte de la interfície.
        // Si només disposàrem d'un mètode, no hi hauria
        // dubte, i podriem invocar-lo amb super.metode2();
        // Com que hi ha dos classes que l'implementen,
        // indiquem aquesta entre <>:
        super<Interface1>.metode2();
        super<Interface2>.metode2();
    }
}

fun main() {
    var o1=classe1();
    o1.metode1();
```

```
    o1.metode2();
}
```

1.8 Paquets (Packages)

Les aplicacions solen compondre's de diverses classes. En projectes grans o col·laboratius, convé dividir el codi font en diversos fitxers fonts.

1.8.1 Organització de fitxers font en Java

A l'hora d'organitzar els fitxers font en Java, cal tindre en compte alguns aspecte:

- Un ***paquet de Java*** és un conjunt de classes i interfícies interrelacionades.
- Les classes i interfícies de la plataforma Java s'estructuren en paquets organitzats per funcions i tasques (ex: java.lang, java.io…).
- Quan creem una aplicació Java, és convenient agrupar les nostres classes i interfícies en paquets.
- Per tal d'incloure una classe en un paquet, s'ha d'incloure al principi del fitxer la sentència:

  ```
  package identificadorDelPaquet;
  ```

- La sentència `package` afecta tot el fitxer font.
- Si hi ha diverses clsses en un únic font, només es pot declarar una classe como a pública, i aquesta ha de tindre el mateix nom que el fitxer font. A més, només els components públics hi seran accessibles des de fora del paquet.
- Si no fem ús de `package` el que defnim al fitxer correspondrà al paquet per defecte (sense identificador), utilitzat habitualment per a aplicacions menudes.

1.8.2 Identificació d'un paquet i accés

Java permet fer ús del mateix nom per a diverses classes si aquestes pertànyen a paquets diferents L'identificador de la classe anirà precedit per l'identificador del paquet. Aquests identificadors compostos es diuen **identificadors qualificats**.

Les empresses i organitzacions solen usar el nom del seu domini d'Internet invers per identificar els seus paquets. Per exemple, l'empresa amb domini `dam.ieseljust.com`, nomenaría els seus paquest de la forma: `com.ieseljust.dam.*`

Per tal d'accedir als components públics d'un paquet podem fer dues coses:

1. Fer ús de l'identificador qualificat:

```
com.ieseljust.dam.NomPaquet.NomClasse Objecte=new
 ↪  com.ieseljust.dam.NomPaquet.NomClasse();
```

Com veiem es tracta d'un nom bastant llarg, pel que s'usa així quan fem ús d'ell una o molt poques vegades.

2. Importar el component del paquet abans d'usar qualsevol classe o interfície i després de la sentència `package`, si aquesta existeix:

```
package com.ieseljust.dam.NomPaquet2;
import com.ieseljust.dam.NomPaquet1.NomClasse;

NomClasse objecte=new NomClasse();
```

En cas que hajam d'utilitzar els components definits al paquet freqüentment, és convenient fer-ho d'aquesta manera.

Si volem importar tots els components d'un paquet, farem ús de l'asterisc:

```
import com.ieseljust.dam.NomPaquet1.*;
```

Per defecte, Java importa automàticament tres paquets: * El paquet per defecte (sense identificador) * El paquet java.lang, amb les classes més habituals (Object, Math…) * El paquet actual de treball (tot allò que definim al mateix directori).

1.8.3 Recomanacions per a la gestió de fitxers

L'estructura de noms d'un paquet està relacionada per l'estrucutra de directoris en què dividim el codi, de manera que:

- El codi font d'una classe o interfície es guarda en un fitxer de text amb el mateix identificador que la classe o interfície i extensió `.java`.
- Els fitxers els guardem a una carpeta amb el mateix nom que el paquet al que es correspon la classe o interfície.

Exemple

- La implementació de dues classes (Classe1 i Classe2) i una interfície (Interficie1) del paquet anomentat NomPaquet1, tindría una estructura de carpetes com aquesta:

```
com
`-- ieseljust
    `-- dam
        `-- NomPaquet1
            |-- Classe1.java
            +-- Classe2.java
            `-- Interficie1.java
```

- Dins de cada fitxer font .java, inclourem el nom del paquet, i el nom de la classe o interfície:

```
package com.ieseljust.dam.Nompaquet1;
...
class Classe1{
    ...
}
```

1.8.4 Packages en Kotlin

La declaració de paquets en Kotlin es fa de la mateixa manera que amb Java, indicant el nom al principi del fitxer, de manera que afecte a tot el que definim al fitxer font:

```
package com.ieseljust.dam.nomPaquetKotlin
```

Com en Java, fem ús de la paraula `import` per habilitar el compilador per localitzar classes i interfícies, però a més, en Kotlin, també podem importar funcions i objectes.

```
import com.ieseljust.dam.nomPaquet.*
```

També podem fer el que es coneix com *import aliasing*, que no és més que donar-li un *alias* a una classe importada. Açò pot ser d'utilitat quan tenim llibreríes amb noms de classe o funcions en conflicte (amb el mateix nom):

```
import com.ieseljust.dam.nomPaquet.funcio as laMeuaFuncio

fun main(args: Array<String>) {
    laMeuaFuncio()
}
```