

UT9.4. Tipus, Composició de tipus i Optional.

Fonaments de tipus

En un llenguatge de programació fortament tipat, els "**valors**" són les instàncies concretes o les representacions d'un tipus particular. Cada valor pertany a un tipus específic i compleix les regles i restriccions d'aquell tipus.

Ja estem familiaritzats amb valors com 12, "hola", o true. Quan escrivim codi en Java operem dins de l'univers de valors on els utilitzem per a realitzar diversos càlculs

També sabem que podem agrupar estos valors en conjunts anomenats **tipus**. Com ara el tipus int, boolean, string, o altres més complexos com tipus funció o tipus envoltori. Passem a un nivell d'abstracció superior on, en comptes de tractar directament amb valors, ens centrem en els tipus. Això ens ajuda a allunyar-nos dels errors i fer programes robustos delegant la responsabilitat al compilador (intentem assignar "hola" a una variable entera, i el compilador ens avisarà). (Pot Java definir tipus?)

Pero això no acaba ací, vorem un tercer univers que abstrau patrons comuns dels tipus. Una forma d'agrupar tipus. Podem agafar tipus com "Integer", "String", "Boolean" i agrupar-los en un "**kind**" o "**tipus de tipus**" que anomenarem "Type" (o bé "*" en convenció Haskell). Pensarem en els tipus com si foren funcions. Existeixen tipus als quals hem de donar com a entrada un tipus per a que ens construisquen el tipus concret. Per exemple "Optional", si li donem com a entrada el tipus "String" ens dona com a resultat el tipus concret "Optional<String>". Passa el mateix per a ArrayList. Per a estos casos direm que el "kind" de Optional o de ArrayList és "Type → Type" o "*" → "*" ja que necessiten un tipus d'entrada per a generar el tipus concret. I que passa amb HashMap? necessita dos tipus d'entrada per a generar el tipus de mapa concret, així que pertany al kind "Type → Type → Type" (sempre ho expressem de forma currificada).

VALUE UNIVERSE	TYPE UNIVERSE	KIND UNIVERSE
12 "Hola" true 'c'	Integer String Integer → Boolean Optional<String>	Type Type → Type Type → Type → Type (Type → Type) → Type

Així, utilitzant la nomenclatura de tipus de Haskell tenim:

```
1 :: Integer
// 1 pertany al tipus Integer
parseInt :: String → Integer
// parseInt pertany al tipus funció d'String a Integer
-
// map agafa un Optional<A>, i una funció A → B, i genera un Optional<B> (A i B són tipus)

Integer :: *
// Integer pertany al kind *
ArrayList :: * → *
// ArrayList pertany al kind * → *
HashMap :: * → * → *
// HashMap pertany al kind * → * → *
```

Si recordes, vam explicar que una funció podia ser d'ordre superior quan era capaç de prendre una altra funció com a paràmetre d'entrada o eixida. Per exemple la funció “map” o “filter” són d'ordre superior perquè tenen com a paràmetre d'entrada una funció. Això també es pot traslladar als tipus.

Encara que Java no ho permet (s'han de fer ús de llibreries de tercers), molts llenguatges funcionals tenen la possibilitat de crear tipus que pertanyen a un kind com este:

$(* \rightarrow *) \rightarrow *$

Els tipus que pertanyen a kinds amb signatures tipus que tenen parèntesis en algun lloc del costat esquerre s'anomenen “tipus de tipus superiors” o bé **“higher-kinded types”** anàlogament al que vam veure amb les funcions d'ordre superior. En la pràctica suposa que podem utilitzar genèrics de tipus genèrics.

En definitiva un higher-kinded type és un tipus que s'abstreu sobre algun tipus que, alhora, s'abstreu sobre un altre tipus.

NO ÉS UN HKT	SÍ ÉS UN HKT
<code>ArrayList<A></code> <code>Type → Type</code>	<code>F<Integer></code> <code>(Type → Type) → Type</code>
Permet construir: <code>ArrayList<Integer></code> <code>ArrayList<Boolean></code> <code>ArrayList<String></code> ...	Permet construir: <code>ArrayList<Integer></code> <code>List<Integer></code> <code>Optional<Integer></code> ...

D'esta manera és possible fer abstraccions, com podem veure en este exemple en TypeScript:

```

type map_option = <A,B>(f: A=>B) => Option<A> => Option<B>
type map_list   = <A,B>(f: A=>B) => List<A> => List<B>

```

Utilitzant HKT:

```

Functor<F>{
  map: <A,B>(f: A=>B) => F<A> => F<B>
}

```

Finalment, en els llenguatges funcionals disposem de classes tipificadoras (**Type Class**). Això ens proporciona polimorfisme ad-hoc, permetent que diferents tipus es comporten de manera uniforme malgrat no compartir una jerarquia d'herència.

Nosaltres ja coneixem el polimorfisme paramètric, és a dir, els genèrics:

```

public static void main(String[] args) {
    System.out.println(aString("Hola"));
    System.out.println(aString(true));
    System.out.println(aString(23));
}

public static <A> String aString(A a){
    return a.toString();
}

```

Quan parlem de polimorfisme ad-hoc, no tenim cap implementació per al nostre mètode sino que només definim i sabem que diversos tipus compartiran eixe mètode, però cadascun tindrà la seua pròpia implementació.

```

public interface Mostrable<A> {
    String aString(A a);
}

public class NewMain {

    public static void main(String[] args) {
        final Mostrable<Integer> enter = (Integer i) -> i.toString();
        final Mostrable<String> string = (String s) -> s;
        final Mostrable<Boolean> boolea = (Boolean b) -> b ? "Sí" : "No";

        System.out.println(boolea.aString(true));
    }
}

```

Encara que Java no s'adapta perfectament al concepte de Type Class, el que volem aconseguir és agrupar uns tipus per compartir uns mètodes (encara que cada implementació siga distinta). Així, en llenguatges com Haskell podem fer:

```
class Show a where
  show :: a -> String

-- Instància de la classe Show per a enters
instance Show Int where
  show x = "Enter: " ++ x

-- Instància de la classe Show per a booleans
instance Show Bool where
  show True  = "Sí"
  show False = "No"

-- Exemples d'ús

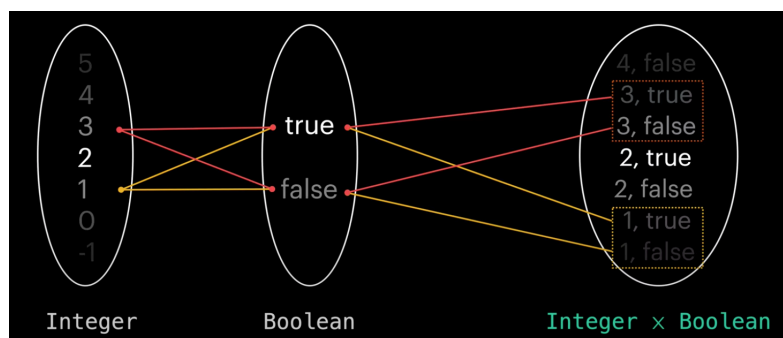
show (5 :: Int)      -- Mostra: Enter: 5
show True            -- Mostra: Sí
```

Composició de tipus

La "composició de tipus" en programació funcional és una idea fonamental que es refereix a la capacitat de crear tipus de dades més complexes combinant tipus de dades més simples. Aquest concepte permet construir estructures de dades modulars i flexibles que són essencials per a la creació de programes funcionals robustos i expressius.

Product Type

Imagina que volem crear un nou tipus de dada a partir de la composició d'Integer i Boolean:



El tipus resultant és realment el **producte cartesià** entre els dos conjunts. Si parlem de tipus, es coneix com a “**Product Type**” (en este cas Integer x Boolean) ja que podrà

contindre qualsevol parella i per tant tindrà tants valors com tinga el conjunt Integer multiplicat per la quantitat de valors que conté Boolean. *[SELECT * FROM Integer, Boolean;]*

En Java podem modelar esta composició de tipus mitjançant una classe o un record:

```
public class IntegerXBoolean {
    private final Integer enter;
    private final Boolean boolea;

    public IntegerXBoolean(Integer enter, Boolean boolea) {
        this.enter = enter;
        this.boolea = boolea;
    }

    public Integer getEnter() {
        return enter;
    }

    public Boolean getBoolea() {
        return boolea;
    }
}
```

Un exemple típic de tipus producte:

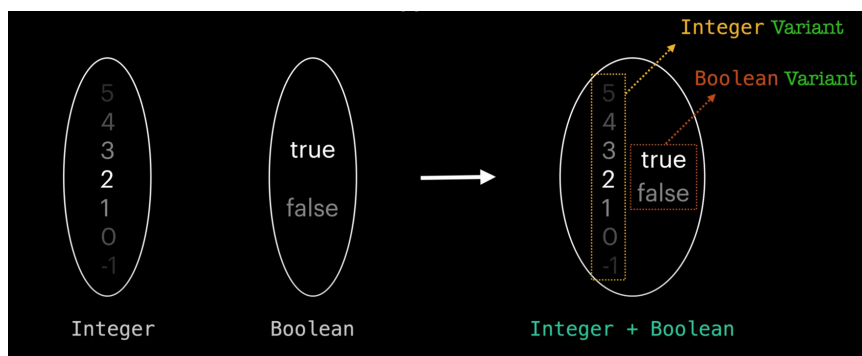
```
// Un punto en un plano bidimensional es un Product Type porque combina dos valores: x e y.
record Punto(int x, int y) {}
```

```
Punto p = new Punto(10, 20);
System.out.println("X: " + p.x() + ", Y: " + p.y());
```

Veure “Exemple Product”

Sum Type

Ara volem crear un nou tipus de dada a partir de la composició d’Integer i Boolean de la següent manera:



Ara el tipus resultant és una suma categòrica o **coproducte**. Si parlem de tipus, “Sum Type” (en este cas Integer + Boolean) només podrà contindre un valor, que podrà ser qualsevol enter o qualsevol booleà, per eixe motiu tindrà tants possibles valors com la quantitat de valors dels enters més la quantitat de valors dels booleans.

*[SELECT * FROM Integer UNION ALL SELECT * FROM Boolean;]*

En java podem modelar esta composició de tipus gràcies al polimorfisme. Generalment es crea una interfície o bé classe pare abstracta (farà la funció de tipus compost) i classes derivades concretes que fan la funció dels tipus primitius.

```
public interface IntegerOrBoolean {
}

class Enter implements IntegerOrBoolean{
    private final Integer enter;

    public Enter(Integer enter) {
        this.enter = enter;
    }
    // Ací la resta de mètodes get/set, equals, toString...
}

class Boolea implements IntegerOrBoolean{
    private final Boolean boolea;

    public Boolea(Boolean boolea) {
        this.boolea = boolea;
    }
    // Ací la resta de mètodes get/set, equals, toString...
}

public static void main(String[] args) {
    IntegerOrBoolean eob = new Enter(3);
    IntegerOrBoolean eob2 = new Boolea(false);
}
```

```
sealed interface Semaforo permits Rojo, Verde, Amarillo {}

final class Rojo implements Semaforo {}
final class Verde implements Semaforo {}
final class Amarillo implements Semaforo {}

Semaforo estado = new Rojo(); // Puede ser uno de los tres tipos concretos
```

L'exemple més típic de coproducte o "sum type" en Java es el tipus **enumerat**, ja que pot contenir només un dels valors enumerats.

ADT (Algebraic Data Type)

Un ADT o “Tipus de Dada Algebraic” és bàsicament un tipus compost que utilitza operacions de suma i de producte com hem vist abans. En la següent part del tema veurem alguns d'aquests ADT com pot ser Maybe, Pair, Either, Try, Result ...

```
public class Pair<A, B>
{
    private final A a;
    private final B b;

    public Pair(A a, B b)
    {
        this.a = a;
        this.b = b;
    }

    public static <A, B> Pair<A, B> of(A a, B b) { return new Pair<>(a, b); }

    public A fst() { return a; }

    public B snd() { return b; }
```

```
public sealed interface Maybe<T> {
    record Just<T>(T value) implements Maybe<T>{}
    record Nothing<T>() implements Maybe<T>{}

    static <T> Maybe<T> of(T value){
        if(value == null) return new Nothing<>();
        else return new Just<>(value);
    }
}
```

ALGEBRAIC DATA TYPES	
SUM TYPES	PRODUCT TYPES
<pre>data State = Dead Alive sealed trait State case object Dead extends State case object Alive extends State</pre> <p>ONE OF TWO VALUES: EITHER DEAD OR ALIVE</p>	<pre>data RumCola = RumCola Rum Cola type RumCola = { rum :: Rum, cola :: Cola } case class RumCola(rum :: Rum, cola :: Cola)</pre> <p>RUM AND COLA TOGETHER</p>
<p>CANONICAL SUM TYPE EXAMPLE</p> <pre>data Either a b = Left a Right b</pre> <p>IF A IS INHABITED BY N VALUES AND B TYPE — BY M VALUES, THEN EITHER A B IS INHABITED BY M + N</p>	<p>CANONICAL PRODUCT TYPE EXAMPLE</p> <pre>(a, b) -- Tuple a b</pre> <p>IF A IS INHABITED BY N VALUES AND B TYPE — BY M VALUES, THEN [A, B] IS INHABITED BY M * N</p>

Pattern matching

El pattern matching o “concordança de patrons” ens permet determinar quin dels tipus primitius és el que estem utilitzant en un “sum type”.

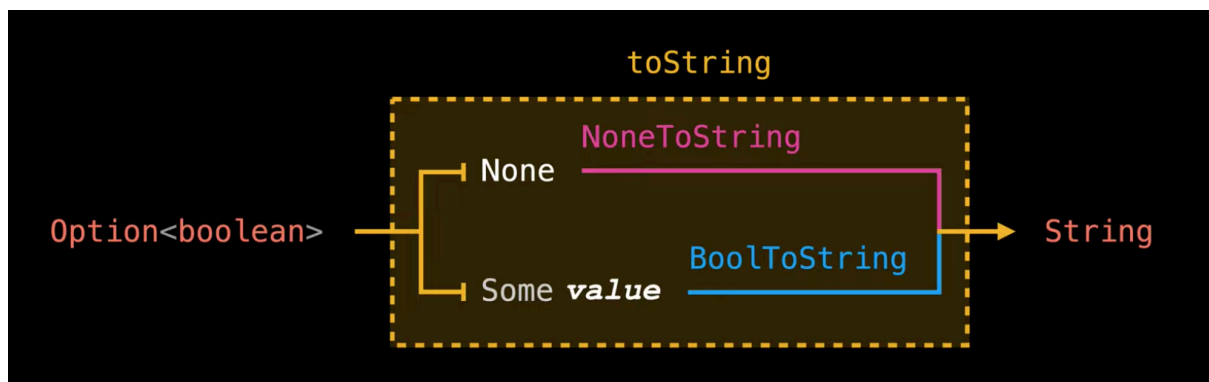
Imagina que tenim el tipus de dada “IntegerOrBoolean” i volem fer-li una operació per tal que si conté un enter es multiplique per dos, i si conté un booleà es negue. En java podem utilitzar “instanceof”:

```
public static void main(String[] args) {  
    IntegerOrBoolean eob = new Enter(3);  
    IntegerOrBoolean eob2 = new Boolea(false);  
  
    if (eob instanceof Enter e) {  
        eob = new Enter(e.getEnter() * 2);  
    }  
    else if (eob instanceof Boolea b){  
        eob = new Boolea(!b.getBoolea());  
    }  
  
    System.out.println(eob);  
}
```

Java ha anat incorporant en les últimes versions noves formes de poder aplicar “pattern matching”. Abans de finalitzar el curs explicarem algunes d’aquestes novetats. (És important remarcar que els switch han de ser exhaustius).

```
public static void main(String[] args) {  
    IntegerOrBoolean eob = new Enter(3);  
    IntegerOrBoolean eob2 = new Boolea(false);  
  
    switch (eob) {  
        case Enter e -> eob = new Enter(e.getEnter() * 2);  
        case Boolea b -> eob = new Boolea(!b.getBoolea());  
        case default -> System.out.println("Tipus invàlid");  
    }  
  
    System.out.println(eob);  
}
```

Per al cas de Maybe imagina que vols implementar un mètode “toString” que convertisca en String el valor que conté. Però si no conté valor (Nothing) hem de retornar una cadena buida:




```

public sealed interface Maybe<T> {
    record Just<T>(T value) implements Maybe<T>{}
    record Nothing<T>() implements Maybe<T>{}

    static <T> Maybe<T> of(T value){
        if(value == null) return new Nothing<>();
        else return new Just<>(value);
    }

    default String toString(){
        if(this instanceof Just j){
            return j.value.toString();
        }
        else return "";
    }
}

```

```

public static void main(String[] args) {
    Maybe<Integer> intMaybe = Maybe.of(7);
    Maybe<Integer> intMaybe2 = Maybe.of(null);

    System.out.println(intMaybe);
    System.out.println(intMaybe2);

    System.out.println(intMaybe.toString() + ".");
    System.out.println(intMaybe2.toString() + ".");
}

```

Borrarme (run) × Borrarme (run) #2 × Borrarme (run) #3 × Borrarme (run) #4

```

run:
Just[value=7]
Nothing[]
7.
.
BUILD SUCCESSFUL (total time: 0 seconds)

```

Videos recomanats:

▶ Records de Java: qué son y cómo usarlos

▶ Pattern matching con records en Java

Maybe en Java: Optional

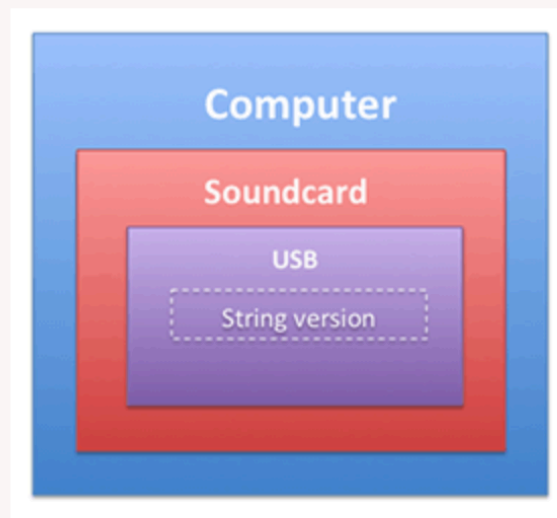


Figure 1: A nested structure for representing a Computer

What's possibly problematic with the following code?

```
String version = computer.getSoundcard().getUSB().getVersion();
```

Creació

```
// Des d'un objecte no null
//Optional<Producte> optionalProducte1 = Optional.of(producte);
Optional<Double> optionalDouble = Optional.of(3.5);

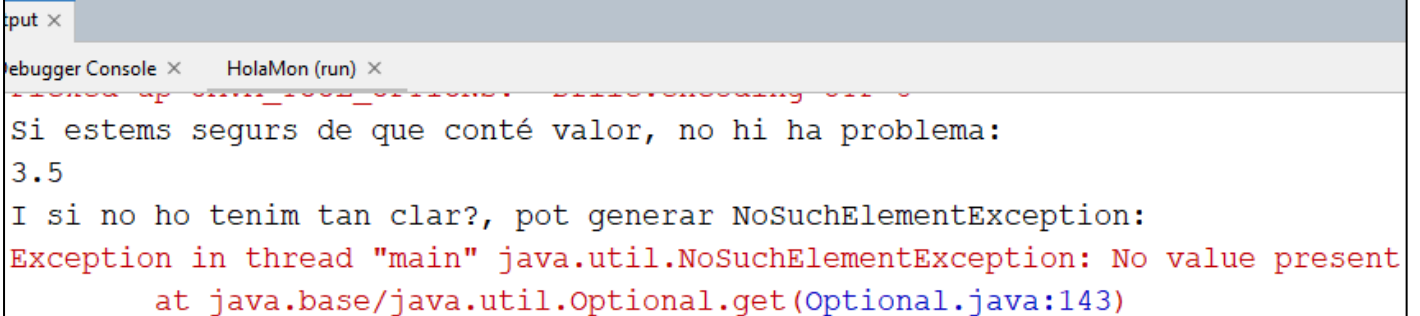
// Des d'un objecte que pot contindre un valor null
//Optional<Producte> optionalProducte2 = Optional.ofNullable(ProducteDAO.getProductePerId(23));
Optional<Double> optionalDouble2 = Optional.ofNullable((Math.random() < 0.5 ? 3.5 : null));

// Un Optional directament buit
// Optional<Producte> optionalProducte3 = Optional.empty();
Optional<Double> optionalDouble3 = Optional.empty();

// Com a resultat d'altres mètodes (per exemple d'Stream)
Optional<String> optionalString = Stream.of("Dilluns", "Dimarts", "Dimecres", "Dijous")
    .filter(dia -> dia.startsWith("X"))
    .findFirst();
```

Obtenint el valor amb get()

```
System.out.println("Si estem segurs de que conté valor, no hi ha problema: ");
System.out.println(optionalDouble.get());
System.out.println("I si no ho tenim tan clar?, pot generar NoSuchElementException: ");
System.out.println(optionalDouble2.get());
```



Si estem segurs de que conté valor, no hi ha problema:
3.5
I si no ho tenim tan clar?, pot generar NoSuchElementException:
Exception in thread "main" java.util.NoSuchElementException: No value present
at java.base/java.util.Optional.get(Optional.java:143)

No és aconsellable l'ús del mètode get() ja que un Optional precisament pot contindre valor o no contindre valor.

Comprovant si hi ha valor: isPresent / isEmpty

Gracies a estos mètodes es pot fer més segur l'ús de "get", però tampoc és aconsellable fer-ho d'aquesta manera perquè realment no és millor que treballar amb el valor "null" i fer condicionals de tipus if(variable == null)

```
if(optionalDouble2.isPresent()){ // Per a fer això podríem haver fet sense optionals if(variableDouble != null)...
    System.out.println(optionalDouble2.get());
}
else{
    System.out.println("No hi ha valor");
}
```

Valor per defecte amb orElse i orElseGet

```
// Si conté valor guardarem el valor. Si no conté valor guardarem 0.0
double resultat = optionalDouble2.orElse(0.0);
System.out.println(resultat);
```

Si volem treballar de forma mandrosa (lazy) també podem proveir un valor amb Supplier.

Diferència entre `orElse` (avaluació ansiosa) i `orElseGet` (avaluació mandrosa):

<pre>double resultat = optionalDouble2.orElse(calculs(0.0)); System.out.println(resultat); }</pre> <pre>private static double calculs(double valor){ System.out.println("Entra al mètode 'càlculs'..."); //Fem els càlculs que toque. return valor; }</pre>	<pre>double resultat = optionalDouble2.orElse(calculs(0.0)); System.out.println(resultat); }</pre> <pre>private static double calculs(double valor){ System.out.println("Entra al mètode 'càlculs'..."); //Fem els càlculs que toque. return valor; }</pre>
Output × Debugger Console × HolaMon (run) × run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 Entra al mètode 'càlculs'... 3.5	Output × Debugger Console × HolaMon (run) × run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 Entra al mètode 'càlculs'... 0.0

<pre>double resultat = optionalDouble2.orElseGet(() -> calculs(0.0)); System.out.println(resultat); }</pre> <pre>private static double calculs(double valor){ System.out.println("Entra al mètode 'càlculs'..."); //Fem els càlculs que toque. return valor; }</pre>	<pre>double resultat = optionalDouble2.orElseGet(() -> calculs(0.0)); System.out.println(resultat); }</pre> <pre>private static double calculs(double valor){ System.out.println("Entra al mètode 'càlculs'..."); //Fem els càlculs que toque. return valor; }</pre>
Output × Debugger Console × HolaMon (run) × run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 3.5	Output × Debugger Console × HolaMon (run) × run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 Entra al mètode 'càlculs'... 0.0

Obtenint un altre opcional amb `or()`

De vegades ens interessa que si l'opcional està buit, ens retorne un opcional nou

```
Optional<Double> resultat = optionalDouble2.or(() -> Optional.of(0.0));
System.out.println(resultat);
}
```

Output ×
Debugger Console × HolaMon (run) ×
run:
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Optional[0.0]

Excepcions amb orElseThrows

En altres ocasions voldrem llançar una excepció en cas que l'opcional estiga buit.

```
double resultat = optionalDouble2.orElseThrow(() -> new RuntimeException("El valor no pot estar absent"));
```

Debugger Console × HolaMon (run) ×

```
run:
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Exception in thread "main" java.lang.RuntimeException: El valor no pot estar absent
```

Acció condicional amb ifPresent

Com a paràmetre necessita un Consumer. Aplicarà el consumidor si conté valor, i no l'aplicarà si no conté valor.

<pre>optionalDouble2.ifPresent(System.out::println);</pre> <p>Debugger Console × HolaMon (run) ×</p> <pre>run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 3.5 BUILD SUCCESSFUL (total time: 0 seconds)</pre>	<pre>optionalDouble2.ifPresent(System.out::println);</pre> <p>Debugger Console × HolaMon (run) ×</p> <pre>run: Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8 BUILD SUCCESSFUL (total time: 0 seconds)</pre>
--	--

Acció condicional i alternativa amb ifPresentOrElse

El mètode ifPresentOrElse accepta dos paràmetres: el primer és un Consumer que s'executarà si l'Optional conté un valor, i el segon és un Runnable que s'executarà si l'Optional està buit. Aquesta estructura permet una gestió més clara i concisa dels casos en què un valor pot o no estar present, evitant l'ús excessiu de comprovacions if-else per determinar si un Optional conté un valor.

```
optionalDouble2.ifPresentOrElse(
    System.out::println,
    () -> System.out.println("NO HI HA VALOR"));
```

Debugger Console × HolaMon (run) ×

```
run:
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
NO HI HA VALOR
BUILD SUCCESSFUL (total time: 0 seconds)
```

Retorn de contingut opcionalment amb filter

El mètode `filter` accepta un predicat (`Predicate<T>`) com a paràmetre. Si l'`Optional` està buit, o si el predicat retorna `false`, `filter` retorna un `Optional` buit. Si l'`Optional` conté un valor i el predicat retorna `true`, es retorna un `Optional` que conté el mateix valor. Això permet realitzar comprovacions condicionals sobre el valor de l'`Optional` sense necessitat d'extraure el valor explícitament o comprovar si està present.

```
Scanner teclat = new Scanner(System.in);
System.out.print("Introdueix un nom: ");
Optional<String> nomA = Optional.ofNullable(teclat.nextLine());

// Mantenim el valor si comença per A
nomA = nomA.filter(nom -> nom.startsWith("A"));

System.out.println(nomA.orElse("NO COMENÇA PER A!"));
```

```
Introdueix un nom: David
NO COMENÇA PER A!
```

```
Introdueix un nom: Arnau
Arnau
```

Quin problema trobes?

```
public static void main(String[] args) {
    String nom = generaNom();
    if(nom.length() > 3) System.out.println("Nom de més de 3 lletres.");
}

public static String generaNom(){
    String[] noms = {"Pepe", "Ana", "Joan", null, "David", "Laura", "Maria", null};
    return noms[new Random().nextInt(8)];
}
```

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "nom" is null

Amb "optional" ja no tindrem `NullPointerException`. `Filter` només aplica si l'opcional conté valor!

```
public static void main(String[] args) {
    generaNom()
        .filter(nom -> nom.length() > 3)
        .ifPresent(nom -> System.out.println("Nom de més de 3 lletres"));
}

public static Optional<String> generaNom(){
    String[] noms = {"Pepe", "Ana", "Joan", null, "David", "Laura", "Maria", null};
    return Optional.ofNullable(noms[new Random().nextInt(8)]);
}
```

Transformant valors amb map

Este mètode és utilitzat per a aplicar una funció de transformació al valor contingut dins de l'Optional, si el valor està present. La funció passada a map pren el valor contingut dins de l'Optional i el transforma en un altre valor, que després es torna a embolicar en un nou Optional. Si el valor no està present (empty) retorna Optional.empty. Això ajuda a evitar errors com NullPointerException i fa que el codi siga més net i fàcil de llegir.

```
public static void main(String[] args) {
    String nom = generaNom();
    System.out.println(generaNom().toUpperCase());
}

public static String generaNom(){
    String[] noms = {"Pepe", "Ana", "Joan", null, "David", "Laura", "Maria", null};
    return noms[new Random().nextInt(8)];
}
```

```
java.lang.NullPointerException: Cannot invoke "String.toUpperCase()"
```

De la mateixa manera, map només aplica la funció de transformació si hi ha contingut. En cas contrari no s'aplica cap funció.

```
public static void main(String[] args) {
    generaNom()
        .map(String::toUpperCase)
        .ifPresent(System.out::println);
}

public static Optional<String> generaNom(){
    String[] noms = {"Pepe", "Ana", "Joan", null, "David", "Laura", "Maria", null};
    return Optional.ofNullable(noms[new Random().nextInt(8)]);
}
```

Sense fer ús d'Optional, com implementaries este codi? Lluitant contra NullPointerException

```
// Imprimir la ciutat del client amb codi 23
```

```
System.out.println(CustomerDAO.getCustomerById(23)
    .map(Customer::getAddress)
    .map(Address::getCity)
    .orElse("NO TROBAT!"));
```

Transformant i aplanant amb flatMap

Mentre que map es pot utilitzar per aplicar una funció de transformació al valor contingut dins d'un Optional, produint com a resultat un nou Optional que conté el resultat de la funció, flatMap es fa servir quan la funció de transformació ja retorna un Optional, i volem evitar acabar amb un Optional<Optional<T>>.

El mètode flatMap permet, per tant, "aplanar" el resultat d'aquestes transformacions, de manera que en lloc de tenir un Optional dins d'un altre Optional, simplement tenim un únic Optional que conté el resultat directament. Això és particularment útil quan treballem amb cadenes de crides de mètodes que poden retornar Optional, permetent-nos encadenar diverses operacions de manera més neta i sense acumular nivells innecessaris d'Optional.

```
class Ordinador {
    Optional<TargetaDeSo> targetaDeSo;
    // Constructors, getters i setters
}

class TargetaDeSo {
    Optional<USB> usb;
    // Constructors, getters i setters
}

class USB {
    String versio;
    // Constructors, getters i setters
}

public class Test {
    public static void main(String[] args) {
        Ordinador ordinador = // inicialització de l'ordinador

        String versioUSB = ordinador.getTargetaDeSo()
            .flatMap(TargetaDeSo::getUsb)
            .map(USB::getVersio)
            .orElse("Versió no disponible");

        System.out.println(versioUSB);
    }
}
```


Convertint a Stream amb stream()

El mètode `stream()` permet transformar un `Optional<T>` en un `Stream<T>`. Si l'`Optional` conté un valor, el `Stream` resultant tindrà un element que és aquest valor. Si l'`Optional` està buit, es crearà un `Stream` buit. Aquesta conversió és útil quan volem utilitzar les operacions de flux (stream operations) com `filter`, `map`, `reduce`, entre d'altres, amb valors que poden ser opcionals.

Exemple:

```
Optional<String> stringOpcional = //...
// Volem una llista que continga 1 element si l'opcional conté valor, i
0 elements si no conté valor.
List<String> llista = stringOpcional.stream()
                                .toList();
```

Quan utilitzar i quan no utilitzar Optional

Quan utilitzar Optional:

- Com a Tipus de Retorn: `Optional` és molt útil com a tipus de retorn en mètodes on hi pot haver un resultat absent. Això fa que el codi siga més segur i expressiu, ja que obliga al consumidor del mètode a tractar explícitament la possibilitat que no hi haja un valor present.
- Per a Encadenar Operacions: `Optional` permet encadenar operacions de manera segura mitjançant mètodes com `map`, `flatMap`, i `filter`. Això facilita la realització de transformacions i comprovacions sense haver de fer múltiples comprovacions de nul·litat.
- Per a Evitar `NullPointerException`: L'ús d'`Optional` ajuda a evitar `NullPointerException`, ja que proporciona mètodes per a tractar valors absents de manera explícita, com `orElse`, `ifPresent`, `ifPresentOrElse`, `orElseGet`, i `orElseThrow`.

Quan no utilitzar Optional:

- Com a atributs de classe: No s'aconsella utilitzar `Optional` com a atributs dins de classes. Això pot augmentar l'ús de memòria i complicar el disseny de la classe. En lloc d'això, es recomana utilitzar valors nuls i gestionar-los adequadament.
- En paràmetres formals de mètodes: Utilitzar `Optional` com a tipus de paràmetre d'entrada en mètodes pot fer que l'API siga menys clara i més difícil d'utilitzar. És millor utilitzar tipus concrets i gestionar la nul·litat dins del mètode si és necessari.
- Per a col·leccions buides: En lloc d'utilitzar `Optional` per a col·leccions que poden estar buides, és millor retornar directament una col·lecció buida. Les col·leccions

com List, Set, i Map tenen mètodes per a comprovar si estan buides, fent innecessari l'ús d'Optional.

- Abús d'Optional: No utilitzes Optional per a tot. Hi ha casos on els valors null són suficients i no justifiquen l'ús d'Optional.

En resum, Optional és una eina poderosa per a millorar la seguretat i llegibilitat del codi quan s'utilitza adequadament. No obstant això, és important no abusar d'aquesta característica i utilitzar-la només quan aporta un valor clar en termes de gestió de valors absents i millora de la claredat del codi.

```
public BigDecimal calculateDiscount(Order order, Customer customer) {  
    /* BigDecimal discount = getDiscountPercentage(customer.getRewardPoints());  
    if (discount != null) {  
        return order.getTotal().multiply(discount);  
    } else {  
        return BigDecimal.ZERO;  
    }  
    */  
    return getDiscountPercentage(customer.getRewardPoints())  
        .map(discount -> order.getTotal().multiply(discount))  
        .orElse(BigDecimal.ZERO);  
}
```

```
// BAD  
String process(String s) {  
    return Optional.ofNullable(s).orElseGet(this::getDefault);  
}  
  
// GOOD  
String process(String s) {  
    return (s != null) ? s : getDefault();  
}
```

Òptiques

Les òptiques són un concepte fonamental en programació funcional per a treballar amb estructures de dades immutables. **Permeten accedir, modificar i transformar valors profundament anidats sense violar la immutabilitat.** En lloc de modificar directament una estructura, les òptiques creen una nova estructura amb els canvis aplicats. En definitiva, ens permeten enfocar-nos en parts específiques d'una estructura complexa, com si tinguérem una "lent" que se centra en una part concreta de les dades.

- **Lents (Lens):** Accés i modificació d'una part específica d'una estructura (una "vista").
- **Prismes (Prism):** Treballen amb tipus suma, permetent accedir i transformar un cas concret.
- **Travessals (Traversal):** Permeten accedir i modificar múltiples valors dins d'una estructura.
- **Òptiques completes (Fold):** Llegeixen diverses parts d'una estructura de manera agregada, però sense modificar-la.
- **Iso (Isomorfismes):** Representen una transformació bidireccional entre dos tipus.

Exemples en Scala:

```
1  import monocle.Lens
2
3  case class Cotxe(model: String, motor: Motor)
4  case class Motor(potencia: Int, tipus: String)
5
6  // Hem de proporcionar el mètode "get" (primer parèntesi) i el mètode "set" (segon parèntesi)
7  val motorLens: Lens[Cotxe, Motor] = Lens[Cotxe, Motor](_.motor)(m => c => c.copy(motor = m))
8  val potenciaLens: Lens[Motor, Int] = Lens[Motor, Int](_.potencia)(p => m => m.copy(potencia = p))
9
10 val cotxe = Cotxe("Toyota", Motor(150, "Híbrid"))
11
12 // Accedir a la potència del motor
13 val potencia = motorLens.andThen(potenciaLens).get(cotxe) // 150
14
15 // Modificar la potència
16 val cotxeModificat = motorLens.andThen(potenciaLens).set(200)(cotxe)
17 println(cotxeModificat) // Cotxe("Toyota", Motor(200, "Híbrid"))
```

```

import monocle.Prism

sealed trait Forma
case class Cercle(radi: Double) extends Forma
case class Quadrat(costat: Double) extends Forma

val cerclePrism: Prism[Forma, Cercle] = Prism[Forma, Cercle] {
  case c: Cercle => Some(c)
  case _         => None
}(c => c)

// Exemple d'ús
val forma: Forma = Cercle(10)
val radi = cerclePrism.getOption(forma).map(_.radi) // Some(10)

val forma2: Forma = Quadrat(5)
val radi2 = cerclePrism.getOption(forma2).map(_.radi) // None

```

```

1  import monocle.Traversal
2
3  val travessal: Traversal[List[Int], Int] = Traversal.fromTraverse[List, Int]
4  val llista = List(1, 2, 3, 4, 5)
5
6  // Multiplicar cada element per 3
7  val novallista = travessal.modify(_ * 3)(llista) // List(3, 6, 9, 12, 15)

```

```

1  import monocle.Fold
2
3  val fold: Fold[List[Int], Int] = Fold.fromFoldable[List, Int]
4  val llista = List(1, 2, 3, 4, 5)
5
6  // Sumar tots els elements
7  val suma = fold.fold(llista)(_ + _) // 15

```

```

1  import monocle.Iso
2
3  // Definim un isomorfisme entre List[T] i Array[T]
4  // Hem d'implementar "get" (primer parèntesi) i "reverseGet" (segon parèntesi)
5  def listArrayIso[T]: Iso[List[T], Array[T]] =
6    Iso[List[T], Array[T]](_.toArray)(_.toList)
7
8  // Exemple d'ús
9  val llista: List[Int] = List(1, 2, 3, 4, 5)
10
11  // Convertim de List a Array
12  val array: Array[Int] = listArrayIso.get(llista)
13  println(array.mkString(", ")) // "1, 2, 3, 4, 5"
14
15  // Convertim de nou d'Array a List
16  val novallista: List[Int] = listArrayIso.reverseGet(array)
17  println(novallista) // List(1, 2, 3, 4, 5)

```

Altres camps de la programació funcional

1. Teoria de Tipus

La teoria de tipus és fonamental en la programació funcional, ja que molts llenguatges funcionals com Haskell, OCaml i Agda utilitzen sistemes de tipus avançats per garantir la correcció del codi.

- Inferència de tipus: Deducció automàtica dels tipus en el codi.
- Tipus dependents: Tipus que depenen de valors, usats en llenguatges com Agda i Idris.
- Polimorfisme: Capacitat d'escriure funcions genèriques que operen sobre múltiples tipus.
- Teoria Curry-Howard: Relació entre la lògica matemàtica i els sistemes de tipus.

2. Càlcul Lambda

El càlcul lambda és la base teòrica de la programació funcional. És un sistema formal per definir i aplicar funcions.

- Currying: Transformació de funcions amb múltiples arguments en una cadena de funcions unàries.
- Reducció alfa i beta: Regles per simplificar expressions lambda.
- Extensionalitat: Principi que defineix quan dues funcions són equivalents.

3. Efectes Algebraics

Aquest camp estudia com gestionar efectes secundaris (com entrada/eixida, estat mutable o excepcions) en un marc purament funcional.

- Mònades: Abstracció per gestionar efectes en llenguatges funcionals.
- Efectes algebraics: Alternativa a les mònades, més flexible i composable.
- Transformadors monàdics: Combinació modular de diferents efectes.

4. Avaluació Pura i Lazy

L'avaluació pura i lazy (avaluació diferida) és una característica clau en llenguatges com Haskell.

- Avaluació mandrosa (lazy): Les expressions no s'avaluen fins que són necessàries.
- Transparència referencial: Garantia que una funció sempre retorna el mateix resultat per als mateixos arguments.
- Estratègies d'avaluació: Comparació entre avaluació estricta (*eager*) i mandrosa (*lazy*).

5. Estructures Recursives

Les estructures recursives són essencials en programació funcional per treballar amb dades immutables.

- Recursió estructural: Processament basat en l'estructura de les dades (llistes, arbres).
- Corecursió: Construcció infinita o generació mandrosa de dades.
- Algorismes recursius: Ús extensiu en llistes, arbres i grafs.

6. Transformació i Optimització del Programa

La programació funcional utilitza tècniques matemàtiques per optimitzar programes sense canviar el seu comportament.

- Fusió de funcions (fusion): Combinar múltiples passos en un sol per millorar el rendiment.
- Transformacions algebraiques: Simplificació basada en propietats matemàtiques.
- Optimització basada en grafs: Representar programes com a grafs per optimitzar-ne l'execució.

7. Lògica Matemàtica

La lògica matemàtica té una connexió directa amb la programació funcional a través de l'isomorfisme Curry-Howard.

- Lògica intuicionista: Base per a sistemes de prova constructiva.
- Proves formals: Verificació matemàtica de propietats del programa.
- Assistents de prova: Eines com Coq o Agda que combinen programació funcional amb proves formals.

8. Programació Reactiva

Aquest camp se centra a gestionar fluxos de dades asíncrones i esdeveniments.

- Programació reactiva funcional (FRP): Model basat en fluxos continus i esdeveniments discrets.
- Exemple: Llibreries com RxJava o ReactiveX.

9. Computació Quàntica

La programació funcional s'utilitza en computació quàntica gràcies a la seua naturalesa declarativa i el seu enfocament matemàtic.

- Exemple: Llenguatges com Quipper o Silq estan dissenyats per modelar algorismes quàntics utilitzant paradigmes funcionals.

10. Programació Distribuïda

Llenguatges funcionals com Erlang són ideals per a sistemes distribuïts gràcies al seu enfocament en concurrència i tolerància a fallades.

- Model Actor: Enfocament basat en actors concurrents que es comuniquen mitjançant missatges.
- Usat àmpliament en telecomunicacions i sistemes distribuïts robustos.

11. Desenvolupament de Llenguatges Específics del Domini (DSLs)

La programació funcional facilita la creació de DSLs gràcies a la seua capacitat per definir abstraccions clares i concises.

Exemple:

- Llenguatges específics per al processament financer o gràfics computacionals.

12. Ciència de Dades i Aprenentatge Automàtic

Els principis funcionals s'apliquen àmpliament en ciència de dades i aprenentatge automàtic:

- Manipulació immutable de dades amb llibreries com pandas (Python).
- Modelatge matemàtic pur utilitzant frameworks com TensorFlow o PyTorch.