UT9.2. Interfícies funcionals i expressions lambda

Interfícies

Recordem que una interfície és un "**contracte**" que compromet les classes a implementar certs mètodes. Estes interfícies se solen utilitzar com a referències (variables) a l'hora de crear objectes, però recordem també que **NO es pot instanciar una interfície**.

```
List<String> llista = new ArrayList<>();
```

Des de Java SE 8 les interfícies poden incloure mètodes "static" i "default".

```
public interface Interficie{
    public void metode(); // Abstracte
    default public void metodePerDefecte(){
        System.out.println("Mètode per defecte");
    }
    public static void metodeEstatic(){
        System.out.println("Mètode estàtic");
    }
}
```

Interfície funcional

- Interfície que **només té un mètode abstracte** (sense comptar amb aquells mètodes abstractes que sobreescriuen mètodes de classes superiors).
- Pot tenir zero, un o diversos mètodes per defecte i/o estàtics
- Usualment, els implementem amb una classe anònima.
- Moltes interfícies conegudes són funcionals.

Un exemple d'interfície funcional és la interfície "Comparator<T>" que hem utilitzat alguna volta. Com veiem l'ÚNIC mètode d'esta interfície que no sobreescriu cap altre és el mètode **int compare(T o1, T o2)**

veure exemple “comunicable”

Interfície funcional. Exemple d'ús

La classe Collections ofereix el mètode estàtic “sort” per ordenar llistes.

public static <T> void sort((List<T> llista, Comparator<T> comparador)

El mètode sort és un mètode genèric que accepta com a primer paràmetre una llista (interfície java.util.List<T>) i com a segon paràmetre un comparador (interfície java.util.Comparator<T>).

Tenim una llista d'objectes de la classe Usuari:

```
public class Usuari {
    private String nom;
    private int edat;
    public Usuari(String nom, int edat){
        this.nom = nom;
        this.edat = edat;
    }
    public String getNom(){
        return nom;
    }
    public int getEdat(){
        return edat;
    }
    //...
}
```

La classe ComparadorUsuaris implementa un criteri d'ordenació d'usuaris per edat (de menor a major):

```java
import java.util.*;

public class ComparadorUsuaris implements Comparator<Usuari>{

    @Override
    public int compare(Usuari o1, Usuari o2) {
        return o1.getEdat() - o2.getEdat();
    }

}
```

Patró estratègia: esta classe està implementant una estratègia de comparació. Els objectes de la classe seran utilitzats pel mètode sort per a comparar objectes.

Ordenarem una llista d'usuaris (ArrayList<Usuari>) utilitzant el criteri anterior:

```java
ArrayList<Usuari> llista = new ArrayList<>();
// llista.add(usuari1); llista.add(usuari2) ...
Collections.sort(llista, new ComparadorUsuaris());
```

El mètode d'ordenació s'aplicarà correctament ja que l'objecte de la classe ComparadorUsuaris és compatible amb la interfície Comparator<Usuari> que necessita el mètode "sort" com a segon paràmetre.

Problema: la necessitat de nous criteris d'ordenació comporta la proliferació de classes que implementen comparadors.

Per a evitar tindre que declarar una classe que només serà utilitzada en un punt del codi, Java permet crear **classes anònimes**. De manera que podríem haver-nos estalviat la implementació de la classe ComparadorUsuaris i directament canviar l'última línia del codi anterior per:

```java
Collections.sort(llista,
    new Comparator<Usuari>(){
        public int compare(Usuari o1, Usuari o2) {
            return o1.getEdat() - o2.getEdat();
        }
    }
);
```

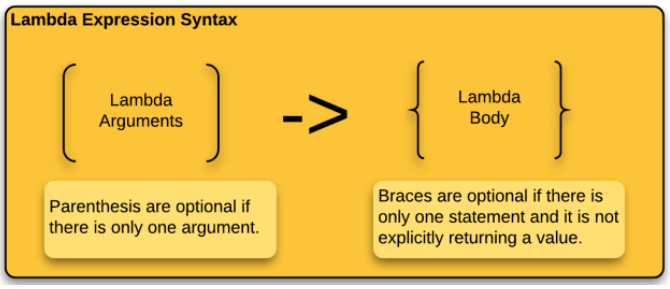
Si el criteri d'ordenació haguera sigut un altre, simplement hauríem d'haver canviat la línia del return.

Però Java pot fer molt més per nosaltres. Si una interfície funcional només tindrà un mètode abstracte que hem d'implementar, no té massa sentit haver d'escriure cada volta la capçalera del mètode (public int compare…) ni tan sols haver d'escriure el tipus de dades dels paràmetres d'entrada (Usuari o1…) ja que pot deduir-ho quan fem referència a una Interfície Funcional concreta. A continuació, vorem com arribar del codi anterior a:

```
Collections.sort(llista,
    (o1, o2) -> o1.getEdat() - o2.getEdat()
);
```

Expressions lambda

Una [expressió lambda] és un bloc de codi que representa una funció anònima.




```
// Sense paràmetres d'entrada ni eixida
() -> System.out.println("Hola")
() -> {System.out.println("Hola");}

// Sense entrada i amb un d'eixida
() -> "Hola"
() -> {return "Hola";}

// Amb un paràmetre d'entrada
(String x) -> System.out.println(x)
(x) -> System.out.println(x)
x -> System.out.println(x)
System.out::println
x -> x.length()
x -> {System.out.println(x); return x.length();}

// Amb múltiples paràmetres d'entrada

(x,y) -> {
    System.out.println(x);
    System.out.println(y);
    return x + y;
}

(Integer x, Integer y) -> x + y

(x,y) -> x + y

Integer::sum
```

Les expressions lambda tenen accés a les variables i els atributs del context del codi:

- Variables locals i paràmetres, sempre que aquests no siguen modificats després de la declaració de l'expressió lambda.
- Accés a atributs, que siguen visibles, sense restricció.

Expressions lambda. Exemple d'ús.

```
public interface OperacionDouble{
    double operar(double op1, double op2);
}
```

```
double n1 = 3.5;
double n2 = 2.0;
/*
OperacionDouble sumar = new OperacionDouble() {

    @Override
    public double operar(double op1, double op2) {
        return op1 + op2;
    }

};
*/
OperacionDouble sumar = (a,b) -> a+b;
System.out.println(sumar.operar(n1, n2));

sumar = (a,b) -> a * b;
System.out.println(sumar.operar(n1, n2));
```

Expressions lambda. Exemple d'ús 2.

```java
List<Integer> numeros = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

// Iterador extern 1
for(int i=0; i<numeros.size(); i++){
    System.out.println(numeros.get(i));
}
// Iterador extern 2
for(int numero: numeros){
    System.out.println(numero);
}

// Iterador intern. Consulta el mètode "forEach"
numeros.forEach(new Consumer<Integer>(){
    public void accept(Integer valor){
        System.out.println(valor);
    }
});

numeros.forEach((Integer v) -> System.out.println(v));

numeros.forEach(v -> System.out.println(v));

numeros.forEach(System.out::println);
```

Fer "Exercici operació binaria"

Interfícies funcionals de l'API de Java

Recorda que una interfície funcional és aquella que conté un sol mètode abstracte. Per tant, les expressions lambda són compatibles amb interfícies funcionals. (**Assignar una expressió lambda a una interfície funcional equival a implementar el seu únic mètode abstracte**).

L'API de Java ens proporciona un gran nombre d'Interfícies Funcionals que ens poden ser útils. Estes es troben al paquet java.util.function:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html

Anem a analitzar-ne algunes.

Interfícies funcionals. Predicate

- Interfície **Predicate<T>**
- Mètode **boolean test(T t)**.
- Comprova si es compleix una determinada condició o no.
- Molt utilitzat amb expressions lambda per a filtrar

```
Predicate<Integer> p = v -> v % 5 == 0;
System.out.println(p.test(25));
System.out.println(p.test(24));

Predicate<String> p2 = v -> v.length() % 2 == 0;
System.out.println(p2.test("Hola"));
System.out.println(p2.test("mon"));

Predicate<String> p3 = v -> v.contains("o");
System.out.println(p3.test("Adeu"));
System.out.println(p3.test("Hola"));
```

```
public void inici() {
    List<Integer> llista = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
    imprimirFiltrant(llista, v -> true);
    // imprimirFiltrant(llista, v -> v%2==0);
    // imprimirFiltrant(llista, v -> v>3 && v<6);
    // imprimirFiltrant(llista, Predicats::esPrimer);
}

public void imprimirFiltrant(List<Integer> llista, Predicate<Integer> predicat) {
    for (int element : llista) {
        if (predicat.test(element)) System.out.println(element);
    }
}
```

```java
public void inici() {
    Predicate<Integer> p1 = divisible(2);
    System.out.println(p1.test(12));
    System.out.println(p1.test(31));

    Predicate<Integer> p2 = divisible(5);
    System.out.println(p2.test(12));
    System.out.println(p1.test(15));
    // imprimirFiltrant(llista, divisible(3));

}

Predicate<Integer> divisible(int valor){
    return n -> n%valor == 0;
}
```

```java
Persona alumne = new Persona("Jose",23);
Predicate<Persona> majorEdat = p -> p.getEdat() >= 18;
Predicate<Persona> nomAmbA = p -> p.getNom().contains("a");

Predicate<Persona> majorEdatOambA = majorEdat.or(nomAmbA);
Predicate<Persona> majorEdatIambA = majorEdat.and(nomAmbA);
Predicate<Persona> menorEdat = majorEdat.negate();

System.out.println(majorEdatOambA.test(alumne));
System.out.println(majorEdatIambA.test(alumne));
System.out.println(menorEdat.test(alumne));
```

¿ `Predicate<Persona> complex = majorEdat.and(nomAmbA).and(p->p.getNom().startsWith("D"));` ?

```java
BiPredicate<Integer,Integer> bp = (n1,n2) -> n1 % n2 == 0;
System.out.println(bp.test(8, 2));
bp = (n1,n2) -> n1 < n2;
System.out.println(bp.test(8, 2));
BiPredicate<String,Integer> bp2 = (s,n) -> s.length() == n;
System.out.println(bp2.test("Hola", 3));

DoublePredicate dp = v -> v - (int)v >= .5;
IntPredicate ip = v -> v % 2 == 0;
```

Revisa el mètode "removeIf" dels ArrayList i el mètode "filter" de la classe Optional

Interfícies funcionals. Consumer

- Interface **Consumer<T>**
- Mètode void **accept(T t)**.
- Representa procediments que fan una acció amb un objecte, sense retornar res. Exemple: mostrar-ho per la consola, emmagatzemar-ho al disc, etc.
- Molt utilitzat amb expressions lambda per imprimir.

```
Consumer<Integer> ci = v -> System.out.println("Valor: " + v);
ci.accept(3);
ci.accept(-5);
Consumer<String> cs = s -> System.out.println(new StringBuilder(s).reverse().toString());
cs.accept("Hola");
cs.accept("Mon");
Consumer<List<Integer>> cli = l -> Collections.sort(l);
List<Integer> llista = new ArrayList<>(Arrays.asList(3,1,6,5,4));
cli.accept(llista);
System.out.println(llista.toString());
```

Addicionalment, té el mètode andThen, que permet compondre consumidors, i encadenar així una seqüència d'operacions

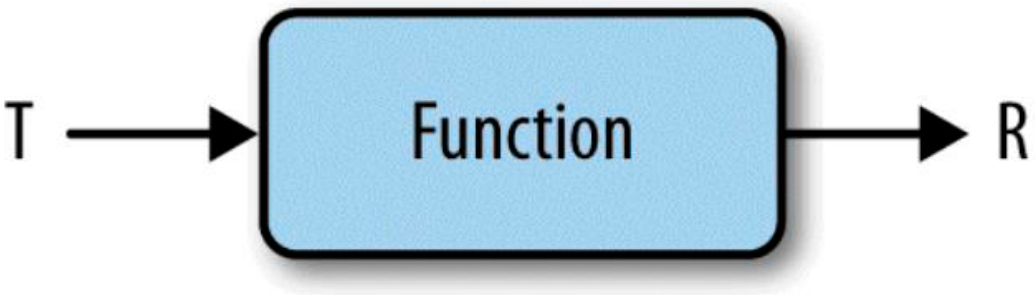
```
List<Integer> llista = new ArrayList<>(Arrays.asList(3,1,6,5,4));
Consumer<Integer> ci = v -> System.out.println("Valor: " + v);
llista.forEach(ci);
System.out.println();
Consumer<Integer> ci2 = v -> System.out.println("Su doble es " + v*2);
llista.forEach(ci.andThen(ci2));
System.out.println();
ci.andThen(ci2).andThen(v->System.out.println("Y su triple " + v*3)).accept(5);
```

Què farà el codi següent?

```
DoubleConsumer dc = v -> System.out.println(v*2);
dc.accept(5);
BiConsumer<String, Integer> bc = (s,i) -> System.out.println(s + ": " + i);
bc.accept("Edat", 33);
```

Revisa el mètode "forEach" dels ArrayList i el mètode "ifPresent" de la classe Optional.

Interfícies funcionals. Function



- Interface **Function<T, R>**
- Mètode **R appy(T t)**.
- Representa les funcions que fan correspondre un objecte de tipus T (entrada) amb un altre de tipus R (sortida)
- Serveix per aplicar una transformació sobre un objecte.
- L'exemple més clar és el mapatge d'un objecte a l'altre

```
Function<Integer,Double> f1 = i -> i/2.0;
System.out.println(f1.apply(35));
System.out.println(f1.apply(32));

Function<Integer,String> f2 = i-> String.valueOf(i);
System.out.println(f2.apply(35));
Function<String,String> f3 = s -> "'" + s + "'";
System.out.println(f3.apply("Hola"));
Function<Integer,String> f4 = f3.compose(f2);
System.out.println(f4.apply(35));

System.out.println(f2.andThen(f3).apply(38));
```

```
BiFunction<Integer,Integer,Boolean> esDivisible = (i1,i2) -> i1%i2==0;
System.out.println(esDivisible.apply(6,2));

IntFunction<Double> tresCuartos = v -> v*3/4.0;
System.out.println(tresCuartos.apply(120));

ToIntFunction<String> longitud = s -> s.length();
System.out.println(longitud.applyAsInt("mon"));
```

*¿Què fan realment els mètodes "**andThen**" i "**compose**"?*

Revisa el mètode "setAll" de la classe Arrays així com "computeIfPresent" de HashMap

Per a pensar més…

🤯

```
BiFunction<String,String,Function<String,String>> bf;
bf = (a,b) -> (c) -> a+b+c;
System.out.println(bf.apply("a","b").apply("c"));

BiFunction<String,String,String> bf2;
bf2 = (a,b) -> a+b;
System.out.println(bf2.apply(bf2.apply("a","b"),"c"));

BiFunction<Function<String,String>,String,String> bf3;
bf3 = (a,b) -> a.apply("a")+b;
System.out.println(bf3.apply((a) -> (a+"b"), "c"));
```

Interfícies funcionals. Supplier

- Interface **Supplier<T>**
- Mètode **T get()**.
- No rep cap paràmetre
- Serveix per obtenir objectes.
- El seu ús és menys estès que les anteriors.

```
IntSupplier s = () -> new Random().nextInt(10);
int n1 = s.getAsInt();
int n2 = s.getAsInt();
System.out.println(n1 + " " + n2);

Supplier<Persona> sp = () -> new Persona(); // Persona::new
Persona p1 = sp.get();
Persona p2 = sp.get();
System.out.println(p1);
System.out.println(p2);
```

Revisa el mètode "orElseGet" de Optional

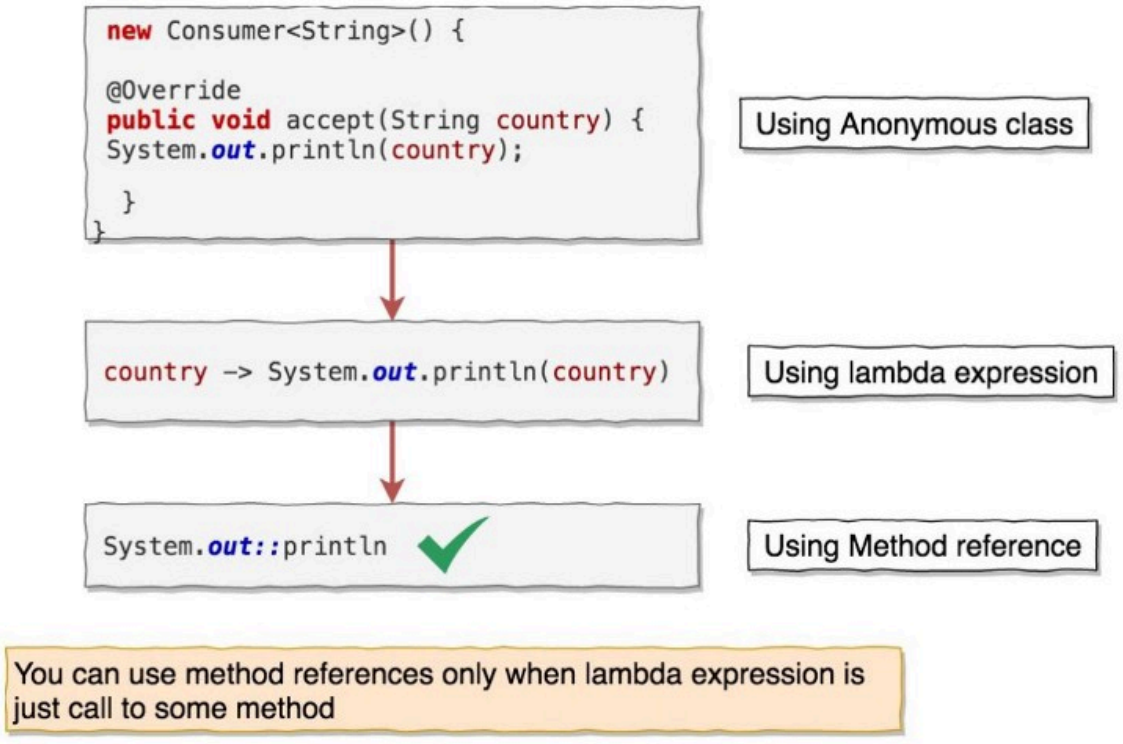
Interfícies funcionals. Altres.

```
Supplier        ()     -> x
Consumer        x      -> ()
BiConsumer      x, y   -> ()
Callable        ()     -> x throws ex
Runnable        ()     -> ()
Function        x      -> y
BiFunction      x,y    -> z
Predicate       x      -> boolean
UnaryOperator   x1     -> x2
BinaryOperator  x1,x2  -> x3
```

Revisa el mètode "updateAndGet" de la classe "AtomicInteger"

Referències a mètodes.

Un tipus de dades que corresponga amb una interfície funcional pot acceptar com a valor tant una expressió lambda com una referència a un mètode ja existent. A partir de la versió 8 de Java, és possible treballar amb referències a mètodes ja definits a alguna classe. Per exemple, la funció lambda ***x -> Math.sqrt(x)*** es pot substituir directament per **Math::sqrt** només cal que els paràmetres d'entrada i els tipus d'entrada i eixida siguen compatibles.



Referències a mètodes. Mètodes Static.

Function<String, Integer> funcio = Integer::parseInt;
int valor1 = funcion1.apply("20");

Utilitzem :: per fer referència al nom del mètode.

No s'especifiquen els tipus dels paràmetres: és implícit en ser assignat a una interfície funcional.

```
public static void main(String[] args) {
    Double[] valors = {3.5,5.2,2.0,10.7};
    aplicar(valors, x -> x * 2);
    System.out.println(Arrays.toString(valors));

    aplicar(valors, Math::sqrt);
    System.out.println(Arrays.toString(valors));
}

public static void aplicar(Double[] llista, Function<Double, Double> funcio) {
    for (int i = 0; i < llista.length; i++) {
        llista[i] = funcio.apply(llista[i]);
    }
}
```

Referències a mètodes. Mètodes d'instància.

Es fa referència a un mètode d'instància com si fora un mètode static.

No obstant això, el compilador tradueix la referència al mètode d'instància en una expressió lambda.

Si ens adonem, el paràmetre d'entrada és el mateix objecte, per la qual cosa el mètode realment disposa només del tipus d'eixida que ha de coincidir amb el de la funció (String getNom()).

```
Function<Usuari,String> funcioNom = Usuari::getNom;
Usuari u1 = new Usuari("Antonio", 33);
System.out.println(funcioNom.apply(u1));
```

Referències a mètodes. Missatges.

Un missatge és l'aplicació d'un mètode sobre un objecte concret i sempre sobre eixe objecte.

```java
LinkedList<Integer> llista = new LinkedList<>(Arrays.asList(2,6,9,3));
Supplier<Integer> eliminarUltim = llista::removeLast;

System.out.println(eliminarUltim.get());
System.out.println(llista.toString());
```

Referències a constructors

Els constructors són considerats com a funcions que retornen un objecte del tipus de la classe.

Per tant, es poden utilitzar com a referències a mètodes fent servir l'identificador new.

```java
Supplier<Persona> generarPersona = Persona::new;
Persona p1 = generarPersona.get();
Persona p2 = generarPersona.get();
```

```java
Function<Integer,ArrayList<Persona>> generarPersones = n -> {
    ArrayList<Persona> persones = new ArrayList<>();
    for(int i=0; i<n; i++){
        persones.add(generarPersona.get());
    }
    return persones;
};

ArrayList<Persona> equip = generarPersones.apply(8);
System.out.println(equip.toString());
```

En establir la referència a un mètode o constructor, no s'identifiquen els paràmetres.

No obstant això, Java permet sobrecàrrega de mètodes. **En cas que una operació estiga sobrecarregada, s'escollirà la definició que s'ajuste a la interfície funcional del tipus de variable**.

```java
public class Test {

    public static void main(String[] args) {
        Supplier<Persona> generarPersona = Persona::new;
        Function<String, Persona> generarPersonaNom = Persona::new;
        BiFunction<String, Integer, Persona> generarPersonaCompleta = Persona::new;
        System.out.println(generarPersona.get());
        System.out.println(generarPersonaNom.apply("Pepe"));
        System.out.println(generarPersonaCompleta.apply("Maria", 45));
        System.out.println();

        Function<Integer, ArrayList<Persona>> generador = Test::generarPersones;
        System.out.println(generador.apply(4));
        System.out.println();
        System.out.println(generador.apply(3));
    }

    public static ArrayList<Persona> generarPersones(int n){
        ArrayList<Persona> persones = new ArrayList<>();
        for (int i = 0; i < n; i++) persones.add(new Persona("Persona"+i));
        return persones;
    }
```

```java
public class Persona {
    private final String nom;
    private final int edat;

    public Persona(String nom, int edat) {
        this.nom = nom;
        this.edat = edat;
    }

    public Persona(String nom){
        this(nom, new Random().nextInt(18, 65));
    }

    public Persona(){
        this("Anònim");

    }

    public String getNom() {
        return nom;
    }

    public int getEdat() {
```

Expressions switch

No hem de confondre la sentència o instrucció switch amb una expressió switch. La primera ja la coneixem, i no deixa de ser una sentència com si es tractara d'un bloc d'if-else. Amb les expressions switch a partir de Java 12 tenim la possibilitat de **retornar diferents valors depenent de cada cas**. El que retorna una expressió switch ho podem assignar a una variable o ens pot servir com a paràmetre d'entrada d'una funció (molt útil en programació funcional)

```
String dia = "dilluns";
String queFer = switch (dia) {
    case "dilluns", "dimarts" -> "escola";
    case "dimecres" -> "treball";
    case "dijous", "divendres" -> "més escola";
    case "dissabte", "diumenge" -> "descans";
    default -> "No existeix!";

};

System.out.println(queFer);
```

També ens pot interessar que l'expressió lambda tinga un cos més llarg per fer alguna operació més (cos entre claus). Això pot suposar un problema:

```
public static String notaQualitativa(int nota) {
    String resultat = switch (nota) {
        case 1,2,3,4 -> {
            System.out.println("Molt malament...");
            return "Suspés";
        }
        case 5,6 ->{
            System.out.println("Enara pots millorar...");
            return "Aprovat";
        }

        case 7,8,9 ->
            {
            System.out.println("Genial!!");
            return "Notable";
        }
        case 10 ->
            {
            System.out.println("Molt malament...");
            return "Excel·lent";
        }

        default ->"valor incorrecte";
    };

    return "La teua nota és un " + resultat;

}
```

Java no sap si el return que hi ha dins del switch és el valor que ha de retornar la funció (que podria interessar-nos, que si es dóna un cas retorne ja un valor la funció) o bé el valor que ha de retornar l'expressió switch. Per solucionar això apareix la paraula reservada "**yield**" que indica a java que és el valor que volem retornar a l'expressió switch i no el valor de retorn de la funció:

```
public static String notaQualitativa(int nota) {
    String resultat = switch (nota) {
        case 1,2,3,4 -> {
            System.out.println("Molt malament...");
            yield "Suspés";
        }
        case 5,6 -> {
            System.out.println("Encara pots millorar...");
            yield "Aprovat";
        }

        case 7,8,9 -> {
            System.out.println("Genial!!...");
            yield "Notable";
        }
        case 10 -> {
            System.out.println("Enhorabona...");
            yield "Excel·lent";
        }

        default -> "valor incorrecte";
    };

    return "La teua nota és un " + resultat;

}
```