UT9.1. INTRODUCCIÓ

Vegem un exemple abans de començar:

```
ArrayList<String> ciutats = new ArrayList<>(
        Arrays.asList("Alacant",
                "Villena",
                "Petrer",
                "Novelda",
                "Elda"));
// Versió imperativa:
boolean trobat = false;
for (String ciutat : ciutats) {
    if (ciutat.equals("Petrer")) {
        trobat = true;
        break;
    }
}
System.out.println("Trobat Petrer?: " + trobat);
```

```
ArrayList<String> ciutats = new ArrayList<>(
        Arrays.asList("Alacant",
                "Villena",
                "Petrer",
                "Novelda",
                "Elda"));
// Versió funcional:
System.out.println("Trobat Petrer?: " + ciutats.contains("Petrer"));
```

Al segle passat, un programador ja jubilat podia escriure este codi en C:

```c
typedef int (*opBinaria) (int, int);

int sumar(int a, int b){ return a + b; }

int multiplicar(int a, int b){ return a * b; }

void executar(opBinaria f, int op1, int op2){
    int resultat = f(op1, op2);
    printf("%d\n", resultat);
}

int main(){
    executar(sumar,1,2);
    executar(multiplicar,1,2);
    return 0;
}
```

En Java les variables només poden contindre dades, no funcions. Així doncs, no és possible declarar un mètode que accepte com a paràmetre una funció. A partir de Java 8 es resol eixa limitació apropant la POO a la Programació Funcional, com veurem al llarg d'esta unitat.

Què és la programació funcional?

La programació funcional és una forma de pensar i dissenyar la solució a un problema en codi de programació (paradigma). Segueix les següents idees principals:

- Les **funcions** són el centre del teu disseny.
- Els problemes es resolen mitjançant la **composició** de funcions (combinació)
- Les funcions son tractades com a **ciutadanes de primera classe**.
- Utilitzem programació **declarativa** (en comptes d'imperativa).
- A partir d'una entrada farem un **encadenament de transformacions** fins a l'eixida.

Programació funcional

En informàtica , la **programació funcional** és un paradigma de programació declarativa basat en l'ús de veritables funcions matemàtiques . En aquest estil de programació, les funcions són ciutadanes de primera classe, perquè les seves expressions poden ser assignades a variables com es faria amb qualsevol altre valor; a més que es poden crear funcions d'ordre superior . [1]

La programació funcional té les seves arrels al càlcul lambda , un sistema formal desenvolupat en els anys 1930 per investigar la naturalesa de les funcions, la naturalesa de la computabilitat i la seva relació amb la recursió . Els llenguatges funcionals prioritzen l'ús de recursivitat i l'aplicació de funcions d'ordre superior per resoldre problemes que en altres llenguatges es resoldrien mitjançant estructures de control (per exemple, cicles). Alguns llenguatges funcionals també busquen eliminar la mutabilitat o efectes secundaris; en contrast amb la programació imperativa, que es basa en els canvis d''estat mitjançant la mutació de variables i objectes. Això significa que, en programació funcional pura, dues o més expressions sintàctiques idèntiques (per exemple, dues trucades a rutines o dues avaluacions) sempre tornaran el mateix resultat. És a dir, es té transparència referencial . Això també pot ser aprofitat per dissenyar estratègies davaluació que eviten repetir el còmput dexpressions abans vistes, estalviant temps dexecució.

Al llarg d'esta unitat anirem veient el paradigma de la programació funcional. Quins requisits cal seguir per programar així?

- Només funcions i expressions.
- Sense efectes col·laterals
- Flux de control completament fix.

Només funcions i expressions.

```
public double sumarDuplicar(double x, double y){
    double suma = 0, doble = 0;
    suma = x + y;
    doble = suma * 2;

    return doble;
}
```

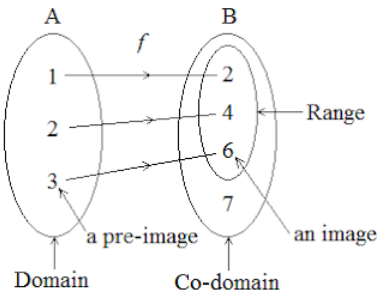
```
public double sumarDuplicar(double x, double y){
    return (x + y) * 2;
}
```

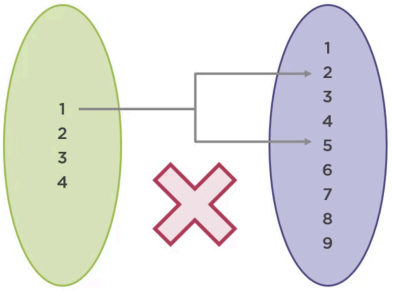
En el primer cas tenim dos variables, depenem de dos espais de memòria i per això al final tenim efectes col·laterals si canviem l'ordre de les assignacions o en un programa molt llarg un altre desenvolupador canvia alguna de les operacions estarà afectant el resultat final sense que siga senzill vore l'error (de fet no ens dóna cap error).

En la segona opció no usem variables, sinó que retornem una expressió. Queda més senzill perquè el lligen futurs companys. Tenim una major integritat dels càlculs en fer-los en la mateixa instrucció i evitem efectes col·laterals (que encara serien més grans si treballem amb variables globals evidentment).

Funcions

En matemàtiques, funció matemàtica és una relació que s'establix entre dos conjunts, a través de la qual a cada element del primer conjunt se li assigna un únic element del segon conjunt. Al conjunt inicial o conjunt de partida també se'n diu **domini**; al conjunt final o conjunt d'arribada, **codomini**.





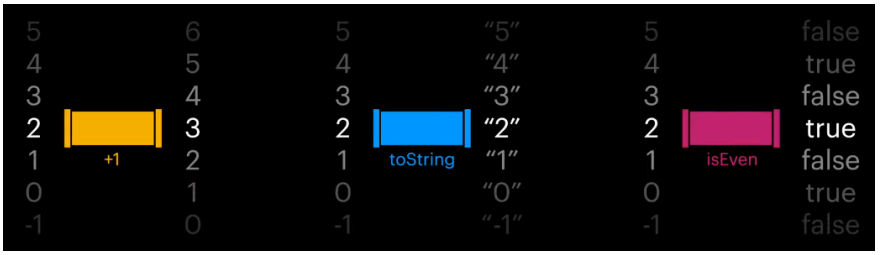
Tots sabem què és una funció matemàtica:

F(x) = x + 5

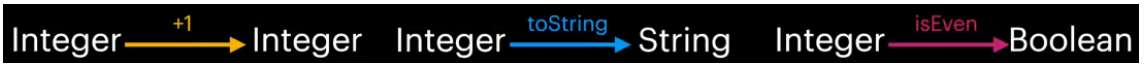
La funció pren el paràmetre X i retorna eixe valor més cinc:

F(10) = 15
F(15) = 20

Un valor entra a una funció i ix transformat:



Si fem una abstracció de les funcions anteriors podem observar el següent:



És habitual expressar-ho amb la notació de tipus de Haskell:

```
increment :: Integer -> Integer    toString :: Integer -> String      isEven :: Integer -> Boolean
```

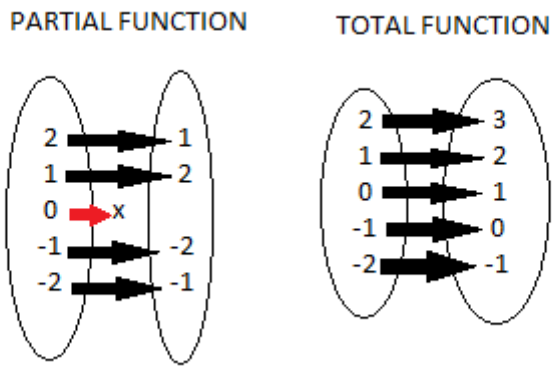
Funció determinista

Una funció determinista és una funció que **per al mateix argument sempre tornarà el mateix resultat**. Els exemples que hem vist anteriorment són funcions deterministes ja que donat un valor d'entrada sempre tindrà com a eixida el mateix valor d'eixida. El resultat d'una funció determinista és predictible.

Pel contrari una funció no-determinista no és predictible, i per a la mateixa funció en ocasions tindrà un resultat i en altres ocasions el resultat serà distint. Un exemple pot ser una funció que multiplica el paràmetre d'entrada per un valor aleatori entre 1 i 10.

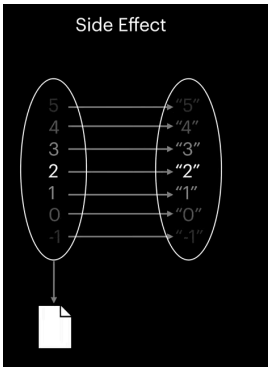
Funció total

Una funció es diu que és total si està definida per a tot el conjunt de partida. Es a dir, que **existeix una eixida per a qualsevol valor del conjunt d'entrada**. Per exemple, la funció "f(x) = x + 1" és una funció total mentre que la funció "f(x) = 2/x" no és una funció total (per al valor d'entrada 0 no es correspon cap valor d'eixida). Una funció que no és total, és a dir, que està indefinida per a algun o alguns elements, es coneix com a **funció parcial**.



Alguns conceptes bàsics https://www.fisicalab.com/apartado/f-inyectiva-sobreyectiva-biyectiva

Efectes secundaris (side-effect)



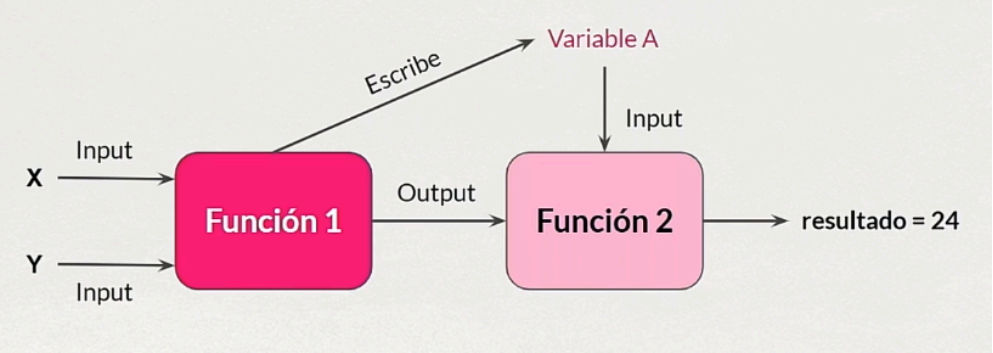
Un efecte secundari o colateral és tot **canvi observable des de fora del sistema**. Els efectes secundaris són inevitables (perquè acaben sent necessaris), alguns exemples són:

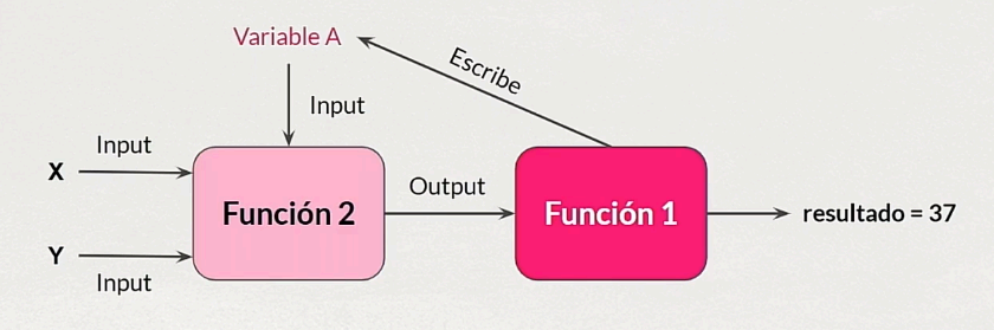
- CRUD sobre fitxers
- CRUD sobre una base de dades
- Enviar/Rebre una trucada de xarxa
- Alterar un objecte/variable usada per altres funcions.

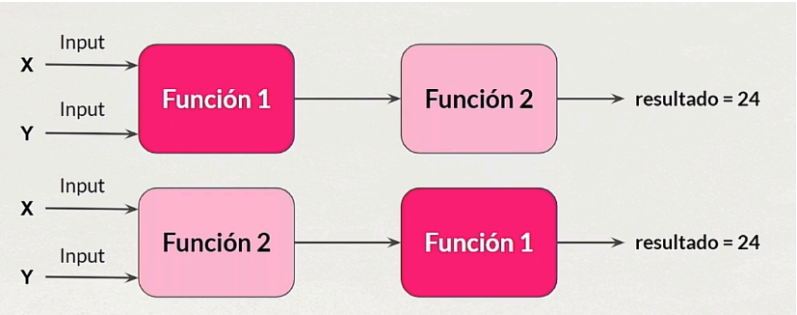
No obstant això, **cal reduir els efectes secundaris (no eliminar-los per complet)**, perquè ajuda a tindre una millor estructura del codi (afavorint la generació de funcions pures, la modularitat i el testing). Tindrem un major control del que passa al nostre programa.

Funció pura

Una funció pura és una funció **determinista**, **total** i **sense efectes secundaris**.







Les funcions de l'última imatge són anomenades "**funcions pures**" ja que no tenen cap efecte secundari i el seu resultat NOMÉS depèn dels paràmetres d'entrada.

Avantatges de les funcions pures:

- Podem eliminar una funció pura que ja no necessitem del nostre programa sense que afecte a la resta del programa (sabem que no modifica altres variables ni objectes)
- Sabem amb tota certesa que en invocar una funció amb els mateixos paràmetres d'entrada sempre tornarà el mateix resultat.
- És possible invertir l'ordre de les invocacions, com s'ha vist en l'última imatge.
- Els compiladors són capaços de detectar les funcions pures i optimitzar-les.

Exemple javascript:

```
const dobleNumero = (n) => {
    return n * 2;
}

const meitatNumero = n => n / 2;

console.log(dobleNumero(meitatNumero(4)));
console.log(meitatNumero(dobleNumero(4)));

// Les dos instruccions anteriors retornen 4
```

- **Funció pura:** determinista (resultat predictible). Fàcil de provar. El resultat serà sempre el mateix en rebre sempre els mateixos paràmetres. No depenen del context, sempre generarà el mateix resultat i no generarà efectes secundaris, és a dir, no afectarà dades d'entrada ni altres dades relatives a altres fluxos de dades. No depenen de l'estat del sistema.

- **Funció impura:** no determinista. Depenen de l'estat del sistema. Depenent del context. Poden generar efectes secundaris, és a dir, poden afectar altres fluxos de dades o veure's afectades per altres fluxos de dades subjacents. No són predictibles.

Una funció pura pot invocar una funció pura, però no a una impura. Si una funció pura invoca una impura es transformarà llavors en una funció impura ja que la naturalesa de la impura farà impredictible el resultat de la funció pura, ja siga per el resultat o pels efectes secundaris i context que impliquen.

"En general, les funcions pures es poden memoitzar."

Aritat

És la quantitat d'arguments que pren una funció, operació o relació.

Així, tenim:

0 arguments: Funció d'aritat 0, nul·lària o nilàdica.

1 arguments: Funció d'aritat 1, unària o monàdica.

2 arguments: Funció d'aritat 2, binària o diàdica.

3 arguments: Funció d'aritat 3, ternària o triàdica.

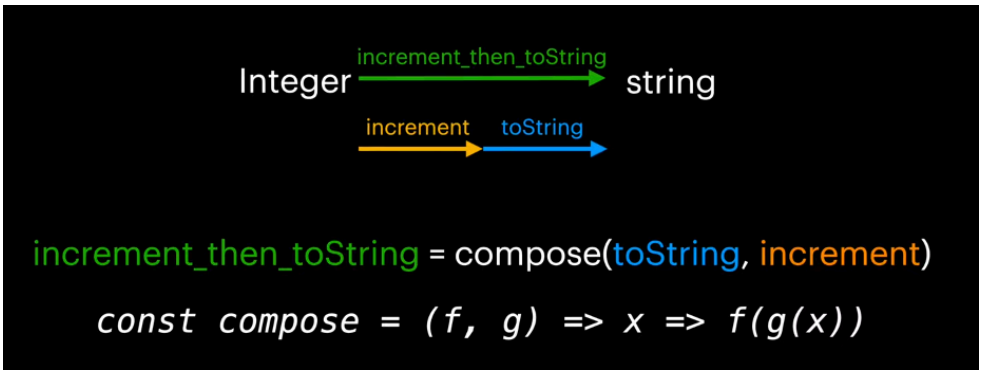
n-arguments: Funció d'aritat N, n-ària o poliàdica.

arguments variables: Funció d'aritat variable o variàdica.

Composició de funcions

Abans hem vist estes dos funcions:

Si ens adonem, el tipus d'eixida de la primera funció és compatible amb el tipus d'entrada de la segona. Per tant és lògic pensar que podem composar estes funcions el que generarà una nova funció:



Quines coses cal evitar en la programació funcional.

- L'ús de bucles d'iteració és una pràctica desaconsellada en programació funcional a favor de la recursió.
- Les declaracions de variables també estan desaconsellades, de manera que sempre que puguem utilitzarem constants (final)
- Igualment les mutacions d'objectes i els mètodes que produeixen mutacions o efectes secundaris com "pop" o "push" no es consideren bona pràctica si s'apliquen sobre la col·lecció original.

Transparència referencial

La transparència referencial és un concepte fonamental en la programació funcional. Es diu que una funció és referencialment transparent si es pot reemplaçar amb el valor al que equival sense canviar el comportament del programa. Com a resultat, l'avaluació d'una funció referencialment transparent dóna el mateix valor per als mateixos arguments. Aquestes funcions hem vist que s''anomenen funcions pures. Una expressió que no és referencialment transparent s'anomena referencialment opaca.

Buscant la immutabilitat.

Una mutació és un **canvi en l'estructura d'un objecte**, en comptes d'una substitució de la instància

Les mutacions són generalment perilloses, ja que un ús habitual augmenta el nombre d'efectes secundaris. Cal **utilitzar mètodes que copien el contingut de l'objecte i no la referència**, com pot ser Arrays.copyOf (per a arrays) o addAll (per a llistes) en Java. (Arrays.asList vs List.of)

```
List<Double> temperatures = new ArrayList<>(Arrays.asList(2.5,4.7,-3.2));
List<Double> temperatures2 = temperatures;
temperatures2.add(9.9);
System.out.println(temperatures);
```

```
List<Double> temperatures = Collections.unmodifiableList(
        new ArrayList<> (Arrays.asList(2.5,4.7,-3.2))
);
List<Double> temperatures2 = new ArrayList<Double>(temperatures);
temperatures2.add(9.9);
temperatures2 = Collections.unmodifiableList(temperatures2);
```

Ja coneixes els getters i setters. Investiga què són els "withers"

Funcions com a ciutadanes de primera classe.

Direm que un llenguatge té funcions de primera classe o de primer ordre si poden ser tractades com qualsevol variable.

Això significa que una funció pot ser passada com a argument, retornada per una altra funció o assignada a una altra variable.

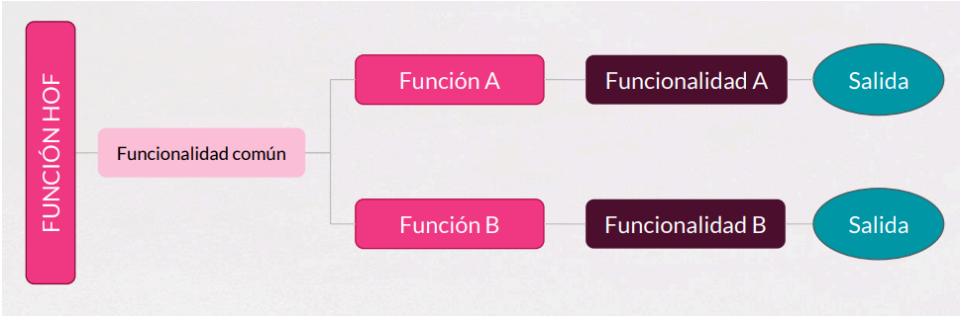
En llenguatges com Javascript o Kotlin les funcions sí que són de primera classe i és senzill aplicar tots estos conceptes. En altres llenguatges com Java les funcions no són ciutadanes de primera classe, per la qual cosa haurem de trobar mecanismes alternatius per a aplicar eixos conceptes.

```
const doblar = function(n){
    return n * 2;
}

doblar(22); // Retorna 44

const aplicarFuncio = function(unaFuncio, x){
    return unaFuncio(x);
}

aplicarFuncio(doblar,5); //Retorna 10
```

```
final Function<Double,Double> doblar = (n) -> n * 2;
doblar.apply(22.0); // Retorna 44


final BiFunction<Function<Double,Double>,Double,Double> aplicarFuncio = (f,x) -> f.apply(x);
aplicarFuncio.apply(doblar, 5.0); // Retorna 10
```

Funció d'ordre superior

Són funcions que treballen amb funcions, és a dir, aquelles funcions que tenen una altra funció com a paràmetre d'entrada o com a paràmetre d'eixida.



Les funcions d'ordre superior són una abstracció de tasques comunes per a funcions

```
const sumar = (x,y) => x+y;
const restar = (x,y) => x-y;
const multiplicar = (x,y) => x*y;

const operar = (x,y,f) => f(x,y);

operar(6,2,restar);  //Retorna 8
operar(6,2,sumar);   //Retorna 4
```

L'ús de funcions d'ordre superior ens permet utilitzar la composició de funcions per a obtindre els nostres resultats.

Si tenim una funció "f(x) = 2x" i un altra funció "g(x) = 3x" ¿quina diferència hi ha entre aplicar $g(f(5))$ *i aplicar* $(g \circ f)(5)$*?*

```
const majorQue = n => {
    return m => m > n;
}

const majorQue5 = majorQue(5);
const majorQue10 = majorQue(10);

majorQue5(8);  //Retorna true
majorQue10(8); //Retorna false
```

Imagina el cas que tenim una funció que rep un String, el converteix a majúscules i l'imprimeix. Més endavant necessitem la funció que rep un String, el converteix a minúscules i l'imprimeix etc. Podem estalviar la repetició de codi fent una funció més general, que rep un String i una funció, aplica esta funció a l'String i l'imprimeix (Exemple en Kotlin):

```
fun logFuncio(str: String, fn: (String) -> String) {
    println(fn(str))
}


fun main() {
    logFuncio("hola", String::uppercase)    // HOLA
    logFuncio("HOLA", String::lowercase)    // hola
    logFuncio("hola", String::capitalize)   // Hola
}
```

Funcions aplicades parcialment

Com que tenim funcions com a ciutadanes de primera classe i per tant podem construir funcions d'ordre superior, apareix el concepte de funcions parcialment aplicades.

Per aplicació parcial s'entén a l'aplicació d'una funció, però subministrant menys paràmetres que els que requereix. El resultat d'aplicar parcialment una funció és una altra funció que espera menys paràmetres que l'original, ja que pot fer reemplaçaments en la definició per expressions o valors concrets. L'aplicació parcial és molt útil per compondre funcions i per parametritzar funcions d'ordre superior. Imagina que tenim una funció "sumar4valors" que suma 4 valors enters:

```
let valor1 = sumar4valors(5,1,2,4)      // valor1 = 12
let parcial1 = sumar4valors(5,9,1)
let valor2 = parcial1(3)                // valor2 = 18
let valor3 = parcial1(2)                // valor3 = 17
let parcial2 = sumar4valors(4,1)
let valor4 = parcial2(2,3)              // valor4 = 10
let valor5 = parcial2(3,1)              // valor5 = 9
let parcial3 = parcial2(6)
let valor6 = parcial3(4)                // valor6 = 15
let valor7 = parcial3(1)                // valor7 = 12

let valor8 = sumar4valors(2,1,5)(4)     // valor8 = 12
let valor9 = sumar4valors(7)(1,3)(2)    // valor9 = 13
let valor10 = sumar4valors(1)(3)(2)(6)  // valor10 = 12 [CURRIFICADA]
```

Currificació (currying)

En programació funcional les funcions solen rebre un únic paràmetre d'entrada. Estes funcions són anomenades "funcions unàries". En Java es representen per el tipus Function<T,R>.

¿I què passa amb les funcions que tenen més d'un paràmetre d'entrada? Es poden convertir en funcions que donat un paràmetre d'entrada retornen una altra funció. Per exemple, per al cas que volem sumar dos enters podem pensar en la següent funció:

f(X, Y) -> X + Y

"A la funció li entren dos valors (X i Y) i retorna un valor (X+Y)"

Anem a convertir esta funció en una funció d'ordre superior (funció que retorna altra funció):

f(X) -> f(Y) -> X + Y

"A la funció li entra un valor (X) i retorna una segona funció"

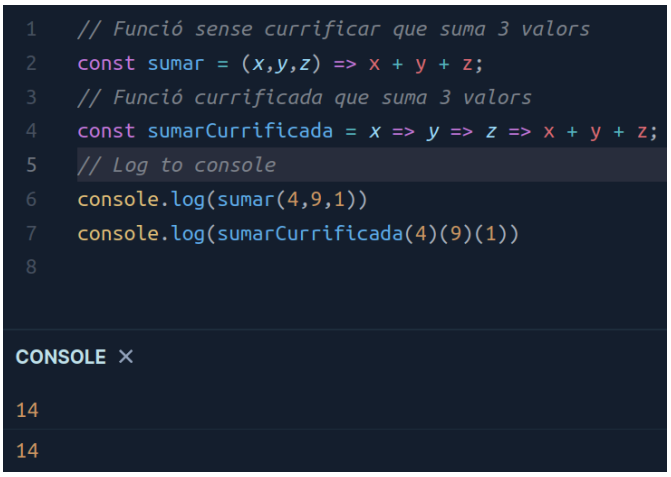
"A esta segona funció li entra un valor (Y) i retorna un valor (X+Y)"

Currying és el **procés de convertir funcions de N arguments a N funcions d'1 argument.** És una forma de reutilitzar funcions convertint-les en factories de funcions. Treballem amb funcions aplicades parcialment que admeten només un paràmetre d'entrada.

```java
// FUNCIÓ SENSE CURRIFICAR
public Boolean esDivisible(Integer dividend, Integer divisor){
    return dividend % divisor == 0;
}

// FUNCIÓ CURRIFICADA
public Function<Integer,Boolean> esDivisible(Integer divisor){
    return (dividend) -> dividend % divisor == 0;
}
```

```java
var esDivisiblePer3 = esDivisible(3);
var esDivisiblePer5 = esDivisible(5);

System.out.println(esDivisiblePer3.apply(10));
System.out.println(esDivisiblePer5.apply(10));
```

Per últim veiem com seria una funció per a sumar 3 números en JavaScript sense currificar i currificada:



POO VS PF

La programació orientada a objectes sol destacar quan el nombre d'operacions no és molt elevat i se centra a tractar dades comunes. També és adequada quan cal tenir un gran nombre d'**entitats o agents treballant independentment per assolir un objectiu,** com per exemple en un joc.

La PF es recomana quan cal un gran nombre d'operacions sobre un flux de dades per exemple quan treballem amb una aplicació de gestió que **tracta dades d'una base de dades, aplicacions de BI, o Data Science**. Evidentment és millor quan cal paral·lelitzar, en sistemes distribuïts; o en API sense estat.

També és molt desitjable quan busquem un màxim rendiment, ja que l'execució sol estar optimitzada per als processadors i és **altament paral·lelitzable** (molt més fàcil de paral·lelitzar per al programador que ho fa de forma declarativa).