UT15.5. Functors, monoides i mònades

Àlgebra

Una àlgebra és un conjunt matemàtic que compleix amb certes propietats i regles d'operació. En general, una àlgebra es compon d'un conjunt d'elements i un conjunt d'operacions que es poden realitzar sobre aquests elements.

De forma resumida, una àlgebra és un **conjunt de valors, un conjunt d'operadors baix els quals està definida i unes lleis que ha d'obeir**.

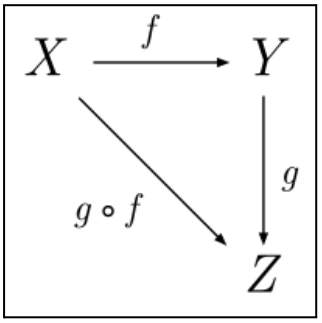
La teoria de categories

La teoria de categories és una **teoria general** de les estructures matemàtiques i les seues relacions que va ser presentada per Samuel Eilenberg i Saunders Mac Lane a mitjans del segle XX en el seu treball fundacional sobre la topologia algebraica. A dia de hui, la teoria de categories s'utilitza a gairebé totes les àrees de les matemàtiques i en algunes àrees de la informàtica. **La programació funcional está basada en la teoria de categories.**

¿Qué és una categoria?

Una categoria és un tipus d'estructura algebraica que **consisteix en un conjunt de punts i fletxes entre ells**. Cada punt d'una categoria s'anomena "**objecte**" i cada fletxa es diu "**morfisme**". Les categories s'utilitzen usualment per a descriure relacions entre objectes i transformacions en contextos abstractes, com ara la teoria de grups o la topologia.

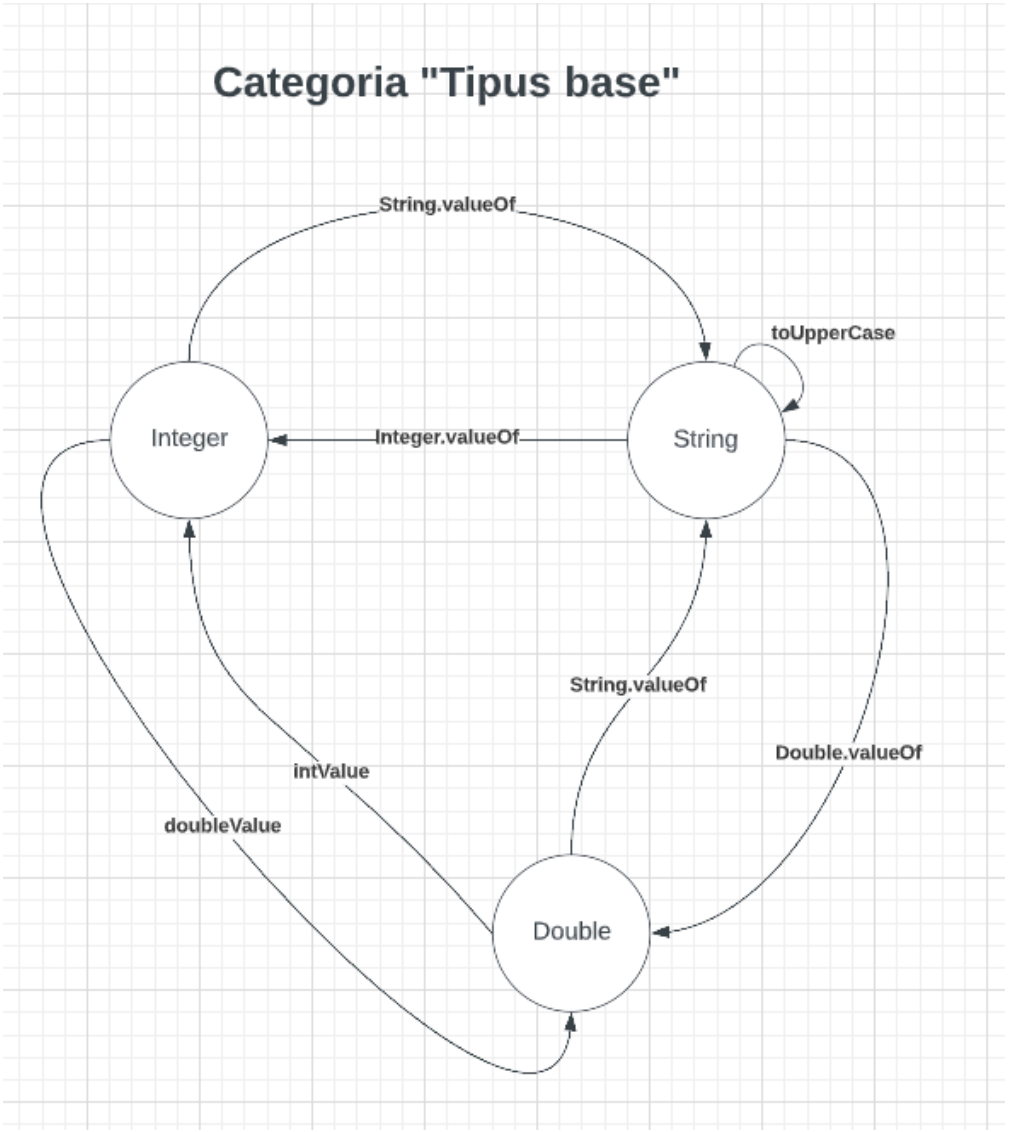
En el context d'una àlgebra, una categoria es pot veure com una **manera de descriure com operen els elements d'una àlgebra i com es relacionen entre si**.



En la imatge tenim una categoria que es composa dels objectes "X", "Y" i "Z", i dels morfismes "f", "g" i "g ∘ f"

Això té moltes més definicions i implicacions matemàtiques que no vorem ara. El que ens interessa és traslladar això a la programació. Imaginem la categoria “tipus” on els objectes son els tipus de dades del nostre llenguatge de programació, on poden haver-hi transformacions entre objectes.

En l’exemple següent no estan tots els objectes i morfismes possibles del llenguatge, només alguns com a mostra:



Com es pot apreciar, “Integer” és un **objecte** d’aquesta categoria. De la mateixa manera, “doubleValue” es tracta d’un **morfisme** d’aquesta categoria entre l’**objecte d’orige** “Integer” i l’**objecte de destí** “Double”. Quan un morfisme té com a orige i com a destí el mateix objecte, l’anomenarem **endomorfisme**. En aquest exemple el morfisme “toUpperCase” és un endomorfisme.

Es recomana vore el següent vídeo per a comprendre millor estos conceptes:
Functional Programming - 17: Category Theory

Els functors

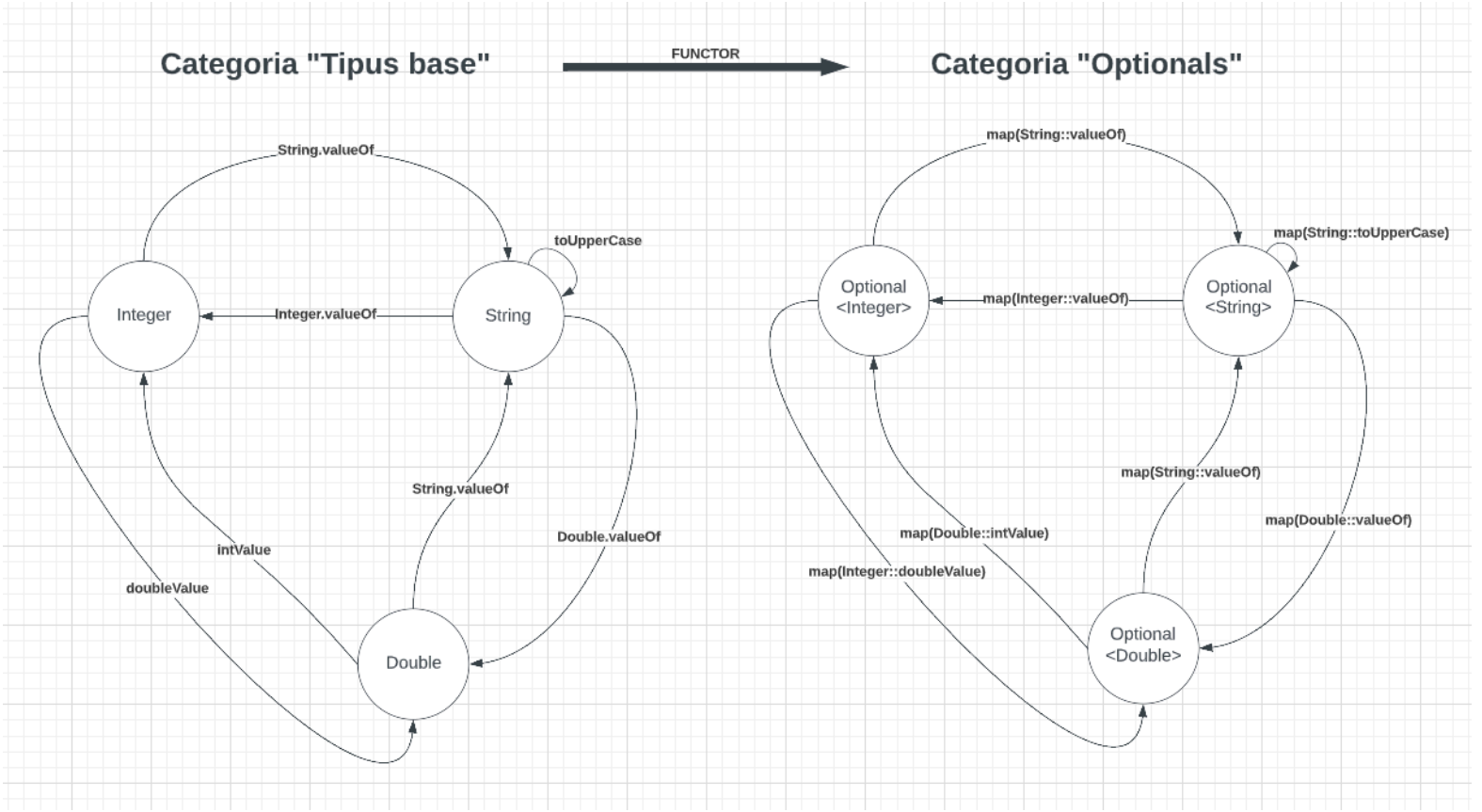
Functor

En teoria de categories, un functor és una **transformació o "mapeig" entre categories que preserva les relacions entre els objectes de cada categoria**. Un functor es defineix com una transformació que assigna a cada objecte d'una categoria un objecte d'una altra categoria de manera que es preserven els morfismes entre aquests objectes.

Els functors són mapejos que preserven l'estructura entre categories. Es poden considerar com a morfismes en la categoria de totes les categories.

Matemàticament per a ser un Functor ha de complir algunes lleis matemàtiques que es voran més endavant.

Si tenim la categoria vista anteriorment “Tipus Base” i una categoria distinta anomenada “Optionals”. Un functor ens permet “mapejar” una categoria a l’altra:



Com es pot apreciar, es preserva l’estructura de “Tipus base” en la categoria transformada, ja que cada objecte de “tipus base” té el seu equivalent en “optionals” i passa el mateix amb els morfismes. També parlem del concepte matemàtic “homomorfisme” ja que els morfismes preserven l’estructura original. Es a dir, que “map(Integer::valueOf)” preserva la mateixa estructura com ho fa “Integer::valueOf” en l’altra categoria.

Per a la programació funcional, **un Functor és simplement una classe envoltori o "wrapper" que segueix certes normes i implementa certs mètodes**. Per exemple "Optional" és un Functor, ja que ens permet "mapejar" objectes d'una categoria a altra, com hem vist en l'exemple anterior podem transformar de la categoria "tipus base" a la categoria "optionals" només fent ús de la "classe envoltori" Optional.

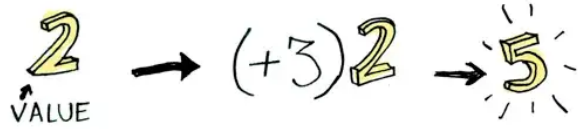
Anomenarem "**lifting**" o "context lifting" al procés d'agafar un valor i retornar un functor que conté eixe valor (posar el valor en context). Per exemple `Optional<Integer> opt = Optional.of(23)` fa un "lifting" ja que transforma el valor 23 en un Optional[23]. D'altra banda anomenarem "**lowering**" al procés contrari, es a dir extreure el valor del context d'un Functor. Per exemple `opt.get()` extreu el valor 23 del functor Optional[23];

Igual com hem vist que un "endomorfisme" té com a orige i destí el mateix objecte, quan tenim com a orige i com a destí la mateixa categoria parlem d'un "**endofunctor**". En programació tots els functors que utilitzarem són endofunctors.

Mètodes que han d'implementar els functors: "map"

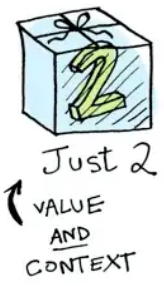
Els functors han d'implementar un mètode anomenat "map". El mètode **map**, també conegut com a **fmap**, pren una funció i aplica la funció al valor que conté el functor, tornant el resultat dins un nou functor.

Quan treballem sense functors podem fer això:

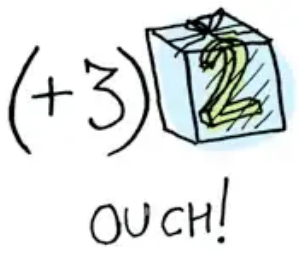



```
int value = 2;

Function<Integer, Integer> mesTres = x -> x + 3;
int resultat = mesTres.apply(value);

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

Ara, treballant amb functors tenim el valor dins d'un context:



Quan un valor està envoltat en un context, no es pot aplicar una funció normal:

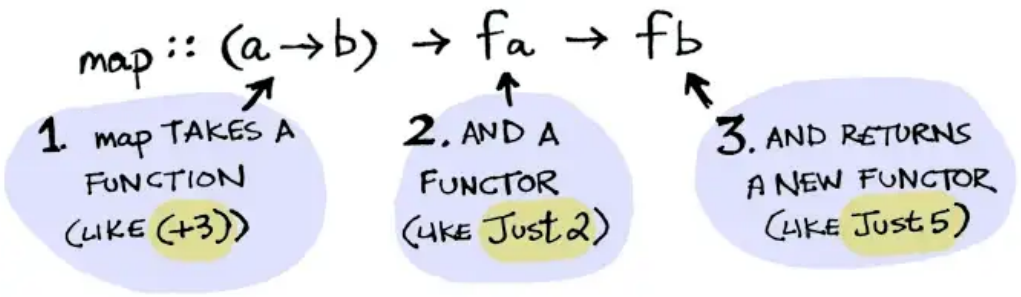



```
Optional<Integer> valueInContext = Optional.of(2);

Function<Integer, Integer> mesTres = x -> x + 3;
Optional<Integer> resultat = mesTres.apply(valueInContext);
```

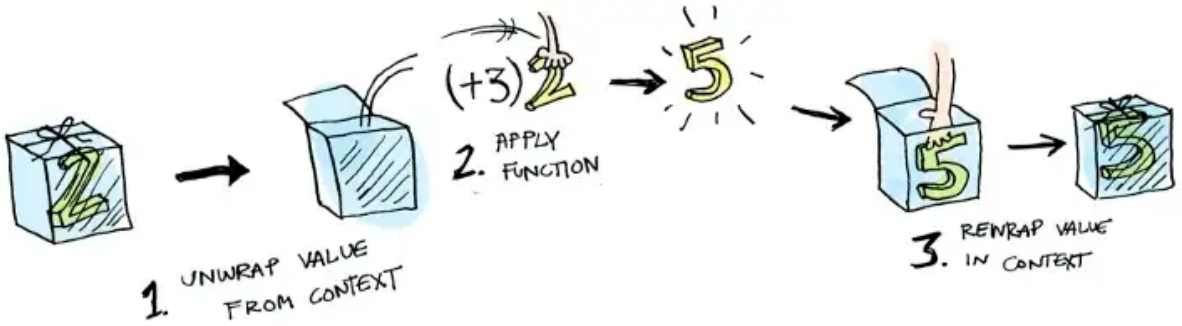
Ací és on entra “map”. “map” és conscient dels contextos, “map” **sap com ha d'aplicar funcions a valors que estan envoltats en un context**. Per exemple, suposa que vols aplicar la funció “(+3)” a un “Optional[2]”. Usant “map”:


```
Optional<Integer> valueInContext = Optional.of(2);
Function<Integer, Integer> mesTres = x -> x + 3;
Optional<Integer> resultat = valueInContext.map(mesTres);
System.out.println(resultat); // Imprimeix Optional[5] !!!
```

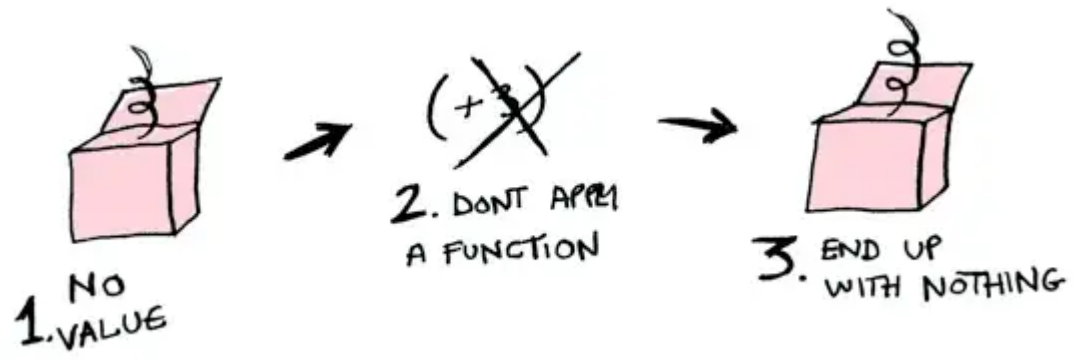
En general el mètode map es defineix de la següent manera:



I en particular per a l’exemple anterior el procediment de “map” és el següent:



I no només això. Com que “map” coneix el context, ha de ser **capaç de gestionar per exemple el que passa quan intentem transformar “empty” o “nothing”**:




```
Optional<Integer> valueInContext = Optional.ofNullable(null);
Function<Integer, Integer> mesTres = x -> x + 3;
Optional<Integer> resultat = valueInContext.map(mesTres);
System.out.println(resultat); // Imprimeix Optional.empty !!!
```

Un functor per tant és un tipus de dada que implementa el mètode map, i que permet aplicar una funció als valors interns de manera "uniforme". Això vol dir que el mètode map ha de seguir certes regles per garantir que el comportament de la funció siga consistent. A continuació vorem quines son eixes regles.

Lleis que han de complir els functors

Llei d’identitat:

Si a map se li dóna una funció d'identitat, ha de retornar exactament el mateix functor.

```
functor.map(x -> x) == functor
```

Per exemple, per a qualsevol tipus d’Optional en general:

```
Optional<Integer> opt1 = Optional.of(3);
Optional<Integer> opt2 = opt1.map(x -> x);
System.out.println(opt1.equals(opt2)); // TRUE
```

Concretament per als Optional<Integer>

```java
Optional<Integer> opt1 = Optional.of(3);
Optional<Integer> opt2 = opt1.map(x -> x * 1);
System.out.println(opt1.equals(opt2)); // TRUE
```

O per als Optional<String>

```java
Optional<String> opt1 = Optional.of("hola");
Optional<String> opt2 = opt1.map(x -> x.concat(""));
System.out.println(opt1.equals(opt2)); // TRUE
```

Llei de composició:

Aquesta llei estableix que en aplicar la funció map al valor “x” amb dues funcions “f” i “g”, el resultat ha de ser igual a aplicar primer la funció “f” i després la funció “g”

`functor.map(x -> f(g(x))) == functor.map(g).map(f)`

Per exemple:

```java
Function<Integer,Integer> mesTres = x -> x+3;
Function<Integer,Integer> mesDos = x -> x+2;

Optional<Integer> base = Optional.of(10);

Optional<Integer> opt1 = base.map(x -> mesTres.apply(mesDos.apply(x)));
Optional<Integer> opt2 = base.map(mesDos).map(mesTres);

System.out.println(opt1.equals(opt2)); // TRUE
```

Functor apuntat o “pointed functor”

Un functor apuntat, també conegut com a "pointed functor", és una variant del concepte de functor que inclou una funció addicional anomenada "of" o “pure” que permet crear un nou functor a partir d'un valor. La funció "pure" és similar a la funció wrap o return en altres llenguatges de programació.

La idea darrere d'un functor apuntat és **proporcionar una manera de crear nous functors a partir de valors simples**, cosa que pot ser útil en certes situacions.

En general, un functor apuntat es pot definir com **un functor que implementa a més el mètode "of"**.

Per exemple “Optional” és un functor apuntat:

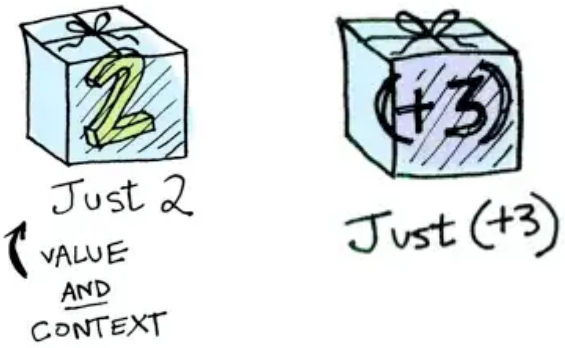
```
var opcional1 = Optional.ofNullable("hola"); // Optional[hola]
var opcional2 = Optional.ofNullable(null); // Optional.empty
```

Functor aplicatiu o “applicative functor”

Un functor aplicatiu és una variant del concepte de functor que inclou una funció addicional anomenada "ap" anomenada “apply” o “<*>” en altres llenguatges de programació.

La idea darrere d'un functor aplicatiu és proporcionar una manera d'aplicar funcions contingudes en functors a valors continguts en altres functors. En general, un functor aplicatiu es pot definir com **un functor apuntat que implementa a més el mètode “ap”** i que compleix certes lleis.

Els Aplicatius ho porten al següent nivell. Amb els Aplicatius, els nostres valors s’envolten en un context, però les nostres funcions també es poden envoltar en contextos!

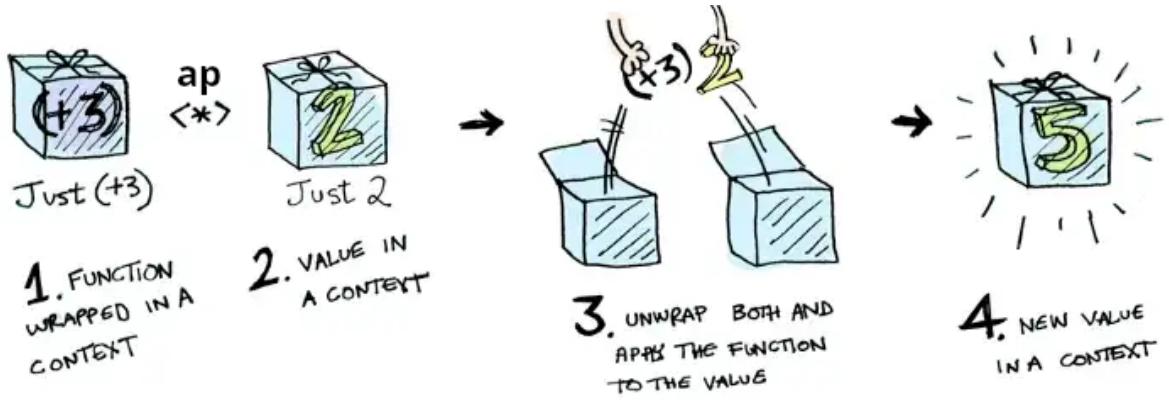


Mètodes que han d’implementar els functors aplicatius: “ap”

El mètode “ap” dels functors aplicatius és una funció que pren dos arguments: un functor que conté una funció i un altre functor que conté un valor. Aplica la funció continguda al primer functor al valor contingut al segon functor i torna el resultat en un nou functor.

La funció ap és útil perquè permet aplicar funcions contingudes en functors a valors continguts en altres functors de manera “uniforme”, sense haver d'escriure codi específic per

a cada tipus de functor. Això fa que siga més fàcil escriure codi genèric que puga treballar amb diferents tipus de functors.



Si tenim que “fg” és un functor que conté una funció i “fx” un functor que conté un valor:

```
ap :: fg -> fx -> f[g(x)]
```

En Java, Optional no disposa del mètode “ap” per tant no és un functor aplicatiu. Podem simular el funcionament amb la classe

```java
public class OptionalApplicative<T> {

    private final Optional<T> opcional;

    private OptionalApplicative(Optional<T> value) {
        this.opcional = value;
    }

    public Optional<T> getOpcional() {
        return opcional;
    }

    public <R> OptionalApplicative<R> ap(OptionalApplicative<R> functor) {
        if (this.opcional.get() instanceof Function<?, ?>) {
            return OptionalApplicative.of(functor.getOpcional().map((Function<R, R>) this.getOpcional().get()));
        }

        return functor;
    }

    public static <T> OptionalApplicative<T> of(Optional<T> value) {
        return new OptionalApplicative<>(value);
    }
```

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

    Optional<Function<Integer,Integer>> ofun = Optional.of(x -> x + 3);
    Optional<Integer> dos = Optional.of(2);

    var oafun = OptionalApplicative.of(ofun);
    var oados = OptionalApplicative.of(dos);

    var resultat = oafun.ap(oados);

    System.out.println(resultat.getOpcional()); // Optional[5]
    System.out.println(resultat.getOpcional().map(x -> x * 2)); // Optional[10]
}
```

Lleis que han de complir els functors aplicatius

A continuació es presenten les lleis que han de complir els functors aplicatius. Com que son conceptes abstractes és possible que ens calga un temps per a comprendre el seu significat. Si amb els exemples posteriors no queda del tot clar, es pot vore amb llenguatges funcionals com haskell https://mmhaskell.com/blog/2017/3/13/obey-the-type-laws

No ens preocupem més del compte en comprendre tots els detalls. Suposem que les estructures que crearem en el llenguatge de programació compleixen aquestes lleis.

Llei d'identitat

Si a un functor aplicatiu que conté una funció d'identitat se li aplica (ap) un segon functor aplicatiu, ha de retornar exactament el segon functor.

`functorAplicatiu[f].ap(functorAplicatiu[x]) == functorAplicatiu[x]`

En exemples posteriors vorem la implementació en Java d'un Functor Aplicatiu. Una volta el tenim podem comprovar la llei d'identitat:

```java
// 1. Identitat
Function<String, String> identityString = x -> x;
var ap1 = ApplicativeFunctor.of("DAW");
var apOp = ApplicativeFunctor.of(identityString);

System.out.println("IDENTITAT: " + ap1.equals(apOp.ap(ap1)));
```

Llei d'homomorfisme

Si a un functor aplicatiu que conté una funció se li aplica (ap) un segon functor aplicatiu que conté un valor, ha de retornar un functor que conté el resultat d'aplicar la funció al valor.

`functorAplicatiu[f].ap(functorAplicatiu[x]) == functorAplicatiu[f(x)]`

```java
// 2. Homomorfisme
Function<String, String> aMajuscules = String::toUpperCase;
var ap2 = ApplicativeFunctor.of(aMajuscules)
        .ap(ApplicativeFunctor.of("Paco"));
var ap3 = ApplicativeFunctor.of(aMajuscules.apply("Paco"));

System.out.println("HOMOMORFISME: " + ap2.equals(ap3));
```

Llei d'intercanvi

Si a un functor aplicatiu que conté una funció 'f' se li aplica (ap) un segon functor aplicatiu que conté un valor 'x' , ha de ser equivalent a si tenim un functor aplicatiu que conté una funció a la qual li entra una funció 'g' i la aplica al valor 'x' i a eixe functor li apliquem (ap) el primer functor aplicatiu.

```
functorAplicatiu[f].ap(functorAplicatiu[x]) ==
functorAplicatiu[g -> g(x)].ap(functorAplicatiu[f])
```

```java
// 3. Intercanvi
ApplicativeFunctor<String> ap4 = ApplicativeFunctor.of(aMajuscules);
String string = "Mollà";

ApplicativeFunctor<String> ap5 = ap4.ap(ApplicativeFunctor.of(string));

Function<Function<String,String>, String> fun = f -> f.apply(string);

ApplicativeFunctor<String> ap6 = ApplicativeFunctor.of(fun).ap(ap4);

System.out.println("INTERCANVI: " + ap5.equals(ap6));
```

Llei de composició

Estableix que la composició de funcions es manté a totes les aplicacions dins del funtor:

```
functorAplicatiu[(f∘g)(x)] ==
functorAplicatiu[f].ap(functorAplicatiu[g].ap(functorAplicatiu[x]))
```

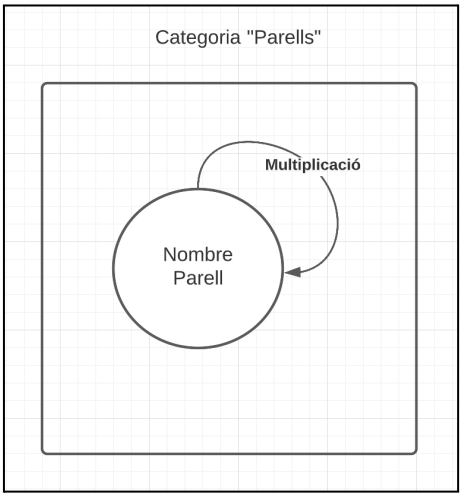
```java
// 4. Composició

ApplicativeFunctor<String> ap7 = ApplicativeFunctor.of(aMajuscules);
ApplicativeFunctor<String> ap8 = ApplicativeFunctor.of(concatPunts);
ApplicativeFunctor<String> ap9 = ApplicativeFunctor.of("Informàtica");

ApplicativeFunctor<String> compost1 = ApplicativeFunctor
        .of(aMajuscules.andThen(concatPunts).apply("Informàtica"));
ApplicativeFunctor<String> compost2 = ap7.ap(ap8.ap(ap9));

System.out.println("COMPOSICIÓ: " + compost1.equals(compost2));
```

Els monoides

Semigrup

Un semigrup és un **conjunt de valors tancats per una operació associativa**, és a dir, **donats dos elements del semigrup, l'operació associativa torna un altre element del mateix semigrup**.

A la teoria de categories, un semigrup es representa mitjançant una categoria amb un únic objecte i un únic morfisme que compleix amb la propietat d'associativitat.



Com hem dit abans un semigrup està format per un conjunt de valors i una operació. Imaginem que el nostre conjunt de valors son els nombres parells (0,2,4,6,8…) i la nostra operació es la de multiplicació. Podem intuir que ací tenim el nostre primer “semigrup”.

Tal i com passava amb els functors, en programació vorem els semigrups com a classes envoltori o “wrapper”. Estes classes han d’implementar cers mètodes i seguir certes regles.

Mètodes que han d’implementar els semigrups: “concat”

El mètode **"concat"** també conegut com a “mappend”, “mconcat” o “combine” és una funció que es fa servir per concatenar dos elements d'un semigrup. Realitza el morfisme del qual hem parlat anteriorment.

Sabem que un semigrup es compon d’un conjunt de valors i d’una operació. **El mètode concat simplement agafa dos valors del conjunt, els aplica l’operació i retorna sempre un valor que pertany al conjunt**.

Per exemple, si tenim el semigrup “OddMultiplication” que hem definit anteriorment, amb el conjunt de valors parells i l’operació de multiplicació entre ells:

```
var si1 = new OddMultiplication(4);
var si2 = new OddMultiplication(2);
var si3 = new OddMultiplication(6);
var si4 = si1.concat(si2); // OddMultiplication[8]
var si5 = si1.concat(si2).concat(si3); // OddMultiplication[48]
```

Lleis que han de complir els semigrups

Llei d'associativitat

L'única regla que han de complir els semigrups és que la seua operació ha de ser associativa. Per a qualsevol element a, b i c del semigrup, i un operador *, es compleix que `(a * b) * c = a * (b * c)`. Això vol dir que l'ordre en què s'apliquen les operacions no afecta el resultat final.

Si tenim el semigrup "OddMultiplication" vist abans, podem comprovar que es compleix la llei d'associativitat:

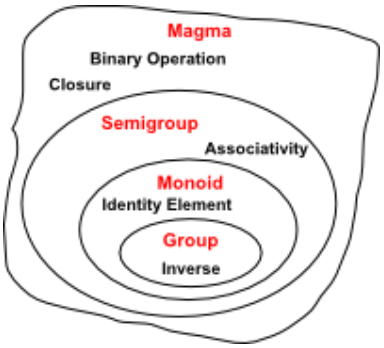
```
boolean multiplicacio = (si1.concat(si2)).concat(si3)
                        .equals(si1.concat(si2.concat(si3)));


System.out.println("Associativitat: " + multiplicacio); // TRUE
```

Si tenim un conjunt format pels nombres enters, i l'operació de divisió entera, NO tindrem un semigrup ja que no complim amb la llei d'associativitat:

```
var sid1 = new DivisionNotSemigroup(282);
var sid2 = new DivisionNotSemigroup(21);
var sid3 = new DivisionNotSemigroup(5);
boolean divisio = (sid1.concat(sid2)).concat(sid3)
                    .equals(sid1.concat(sid2.concat(sid3)));


System.out.println("Associativitat: " + divisio); // FALSE
```

El conjunt format pels nombres enters i l'operació de la divisió entera no és un Semigrup sinó un "**Magma**".

Monoide



Un monoide és un **tipus especial de semigrup**. Bàsicament, per a que un semigrup siga un monoide **ha de contenir en el seu conjunt de valors un valor especial anomenat "neutre" o "identitat"**.

Com sempre, en programació vorem els monoides com a classes envoltori o "wrapper". Estes classes han d'implementar certs mètodes i seguir certes regles.

Per a comprendre millor la relació Magma > Semigrup > Monoide recomane visualitzar este vídeo: Functional Programming - 18: Magma, Semigroup, Monoid

Mètodes que han d'implementar els monoides: "empty"

Els monoides, a més del mètode concat (ja que és un semigrup) han d'implementar un **mètode estàtic anomenat "empty"** també conegut com a "identity" o "mempty". El mètode **"empty" retorna l'element neutre del conjunt de valors del monoide**.

Per exemple, si tenim el monoide "Sum" que es compon del conjunt de valors enters i de l'operació suma, per a aquest monoide el mètode "emtpy()" ha de retornar Sum[0], ja que el 0 és el valor neutre del conjunt de valors per a l'operació que defineix. Si tenim el monoide "Prod" que es compon del conjunt de valors enters i de l'operació de multiplicació, el mètode "empty()" retornarà Prod[1] etc.

```
// MONOIDE "SUM"

var sum1 = new Sum(8);
var sum2 = new Sum(0.5);
var sum3 = new Sum(3);

var sumTotal = sum1
        .concat(sum2)
        .concat(Sum.empty())
        .concat(sum3); // Monoide[11.5]
```

Lleis que han de complir els monoides

Llei d'associativitat

Com que es tracta d'un semigrup, també ha de complir la llei d'associativitat que hem vist als semigrups.

Dreta
Quan a un monoide li concatenem el neutre, retorna el mateix monoide.
`monoide[x].concat(id) = monoide[x]`

Per exemple, si tenim el monoide “Prod” vist anteriorment:

```
var prod2 = new Prod(0.5);
boolean identitatDreta = prod2.concat(Prod.empty())
                              .equals(prod2); // TRUE
```

Esquerra
Quan al neutre li concatenem un monoide, retorna el mateix monoide.
`id.concat(monoide[x]) = monoide[x]`

Per exemple, si tenim el monoide “Prod” vist anteriorment:

```
boolean identitatEsquerra = Prod.empty().concat(prod2)
                                .equals(prod2); // TRUE

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

Per tant, per a ser **un monoide ha de complir amb la llei d’associativitat, amb la d’identitat dreta i la d’identitat esquerra**. Si recordem el semigrup “OddMultiplication” podem vore que NO es tracta d’un monoide, ja que no compleix amb les lleis d’identitat dreta ni esquerra (no té element neutre).

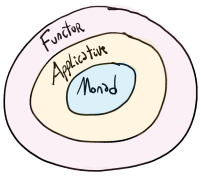
¿I la divisió d’enters, compleix les lleis d’identitat dreta i esquerra?

Les mònades

Arribem a la “joia de la corona”. La definició més estesa la tenim ací: “**A monad is just a monoid in the category of endofunctors**”. També existeix una llegenda urbana que diu que quan comprenem del tot que és una mònada perdem l’habilitat d’explicar a un altre el que és. No cal dir res més… o sí.

Mònada

Fem un pas enrere. Functors i monoides eren classes envoltori al voltant d'un valor que ens permet executar operacions. En el cas dels funtors es tractava d'un “mapeig” en el cas dels monoides de la composició, on la composició és una única operació.



Les mònades són alhora un functor i un monoide. Això no fa que siga més senzill pero podem definir-ho de forma més senzilla. Matemàticament es tracta d’un **functor aplicatiu** (encara que en molts llenguatges de programació només cal que siga un functor apuntat) que també segueix les regles dels monoides.

En un primer moment podrem pensar que tot això no té utilitat en la programació funcional, pero quan més es treballa en aquest paradigma més ens adonem de que les mònades son fonamentals.

L'ús de mònades a la programació funcional és útil perquè permet als programadors **manejar errors i el valor absent de manera segura i controlada**. En utilitzar mònades, els programadors poden escriure codi que manege l'error de manera apropiada sense haver de preocupar-se per la propagació d'excepcions o la validació explícita dels resultats. Això permet als programadors escriure codi més net i fàcil de llegir i mantenir.

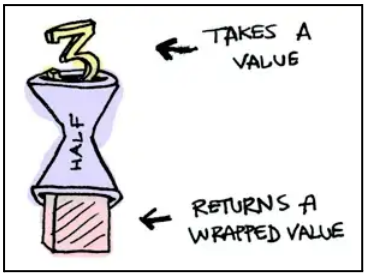
A més, l'ús de mònades permet als programadors utilitzar tècniques com l'**encadenament d'operacions monàdiques** i la composició de funcions monàdiques per a crear codi més expressiu, concís i fàcil d’entendre cosa que pot millorar la productivitat i la qualitat del codi.

Com sempre, en programació vorem les mònades com a classes envoltori o “wrapper”. Estes classes han d’implementar certs mètodes i seguir certes regles.

Mètodes que han d’implementar les mònades: “flatMap”

Les mònades a més d’implementar els mètodes dels functors aplicatius (o almenys dels functors apuntats) també ha d’implementar un mètode “**flatMap**” que en altres llenguatges es coneix com a “bind”, “chain” o “>>=”.

Recordem que els functors apliquen una funció a un valor envoltat en un context, els aplicatius apliquen una funció envoltada a un valor envoltat i per últim **les mònades apliquen, a un valor envoltat, una funció que alhora torna un valor envoltat**. Anem pas per pas.



Al principi hem fet l’exemple dels functors amb “Optional”. Resulta que “Optional” és una mònada (implementa també el mètode “flatMap”). Imaginem que tenim una funció “half” que només funciona per a nombres parells. La funció “half” retorna un Optional, concretament un Optional que conté la meitat del valor si és parell, i si es imparell un Optional.empty.

Tenim la funció “half” a la qual volem aplicar-li la mònada Optional[3]. I falla…

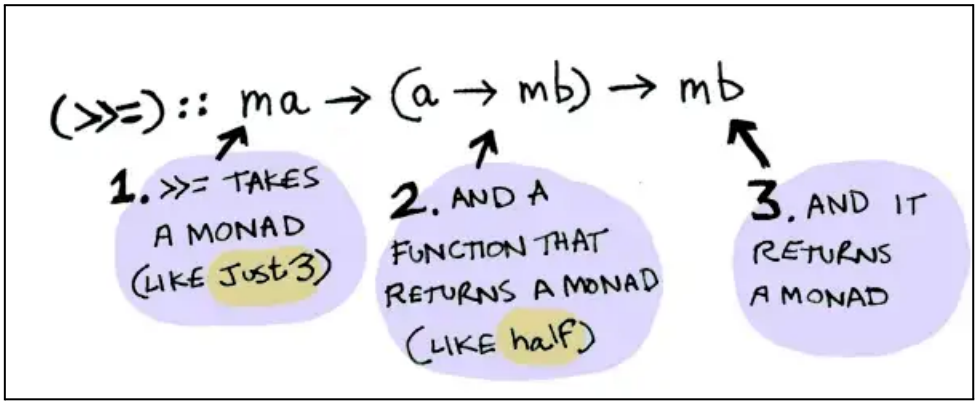

```
Function<Integer, Optional<Integer>> half = x ->
        (x % 2 == 0)
        ? Optional.of(x / 2)
        : Optional.empty();

Optional<Integer> tres = Optional.of(3);
var resultat = half.apply(tres);
```

La funció “half” espera un valor enter i no una mònada Optional[Integer], per això falla. Però que passa si en comptes d’aplicar la funció directament utilitzem “map” com vam vore als functors?

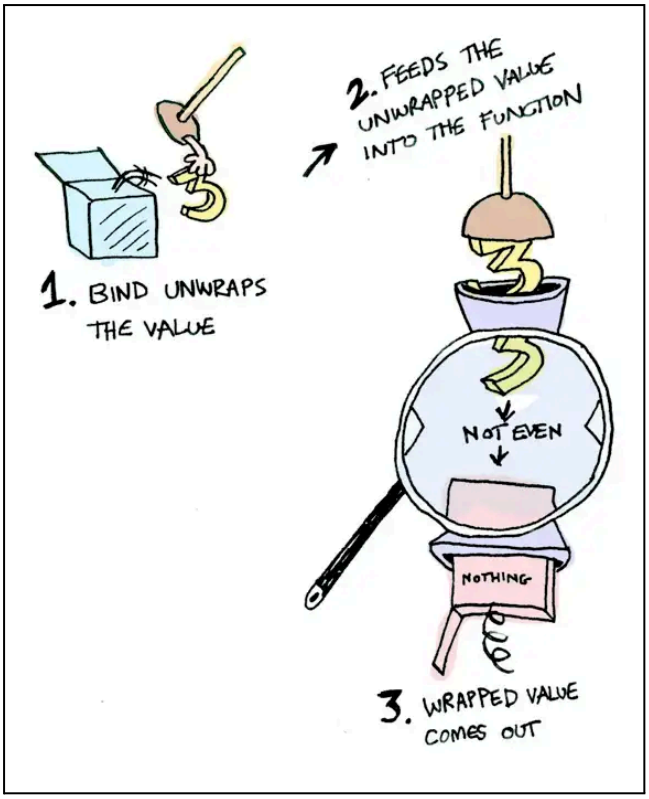
```
Optional<Integer> tres = Optional.of(3);
var resultat = tres.map(half);
System.out.println(resultat); // Optional[Optional.empty]

Optional<Integer> quatre = Optional.of(4);
var resultat2 = quatre.map(half);
System.out.println(resultat2); // Optional[Optional[2]]
```

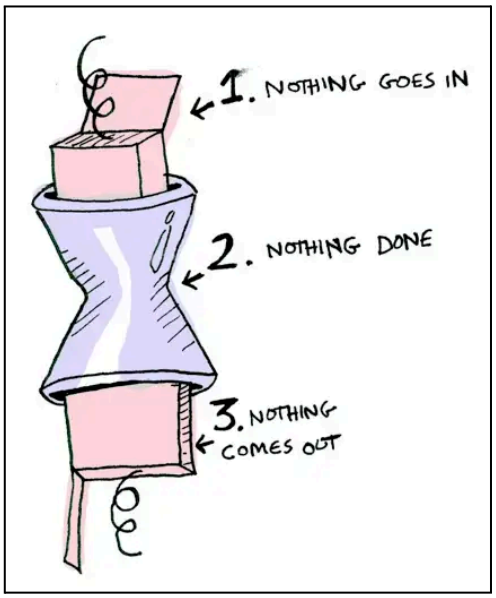
Perfecte! Ja no marca errors el compilador! Però espera… Que passa si imprimim els resultats? Que retorna una mònada dins d’una mònada. Com que el mètode “map” feia un “lifting” del resultat obtingut, al fer un “lifting” del que retorna “half” (retorna una mònada) al final es produeix eixa nidació.

Per a solucionar aquest problema apareix el mètode “flatMap”. El mètode ha de fer primer la transformació (map) i després l’aplanament (flat). Finalment agafa el valor resultant (fora de tota mònada) i li fa un lifting per a retornar una mònada sense cap niuament.



Ací podem vore de forma gràfica el procediment que segueix el mètode “flatMap” o “bind”:



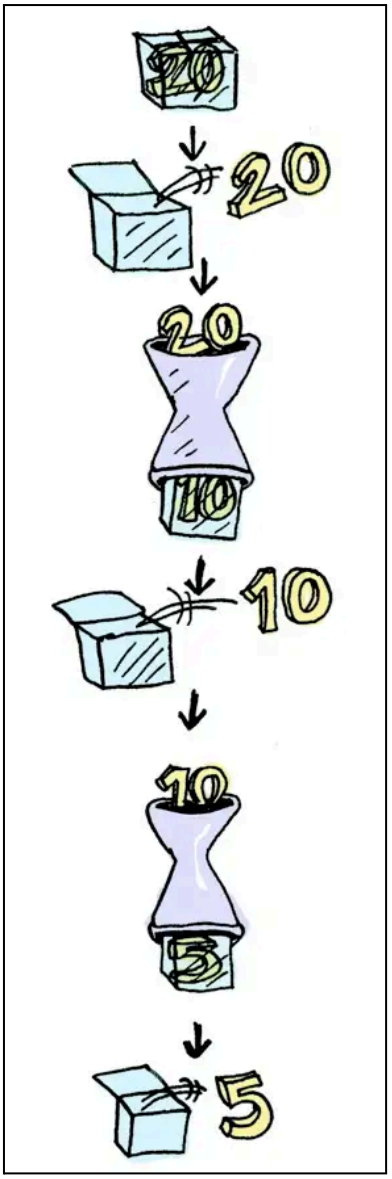


Ara sí!

```
Optional<Integer> tres = Optional.of(3);
var resultat = tres.flatMap(half);
System.out.println(resultat); // Optional.empty

Optional<Integer> quatre = Optional.of(4);
resultat = quatre.flatMap(half);
System.out.println(resultat); // Optional[2]

Optional<Integer> buit = Optional.empty();
resultat = buit.flatMap(half);
System.out.println(resultat); // Optional.empty
```

A més, som capaços d'encadenar operacions monàdiques ja que el resultat sempre serà una mònada d'un únic nivell i per tant li podem aplicar funcions de forma encadenada.

```
Optional<Integer> opcional = Optional.of(20);
System.out.println(opcional.flatMap(half).flatMap(half)); // Optional[5]
```



Mònades ben conegudes són **Maybe (Optional), Either, Try, Result, Stream, IO …**
Railway oriented programming Scott Wlaschin. Monads

Lleis que han de complir les mònades

Com que es tracta d'un functor, qualsevol mònada complirà les lleis dels functors, i ademés complirà les lleis dels monoides.

Exemples

Revisa els exemples proposats en els projectes de NetBeans "AlgebraicStructures" i "Monades"

Point Free Style i Railway Oriented Programming

Ens queden molts més conceptes que podem treballar en la programació funcional així com altres ADT i exemples d'ús. Encara que no ho podem veure tot, és interessant introduir almenys dos metodologies de treball molt utilitzades en la programació funcional com són:

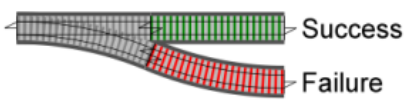
La programació tàcita, també anomenada estil sense punts (Point Free Style), és un estil de programació en què les definicions de funcions no identifiquen els arguments (o "punts") sobre els quals operen. En canvi, les definicions només componen altres funcions, entre les quals hi ha els combinadors que manipulen els arguments.

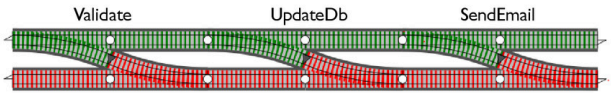
Exemple en JavaScript:

```javascript
const split = separator => str => str.split(separator);
const head = arr => arr[0];

const splitByLine = split("\n");
const getRequestInfo = split(" ");
const getFirstPair = ([first, two]) => [first, two];

const parseRequest = compose(
    getFirstPair,
    getRequestInfo,
    head,
    splitByLine
);

const requestExample = `GET /users HTTP/1.1
  Host: elrincondev.com
  User-Agent: ExampleBrowser/1.0
  Accept: */*
`;

const [method, path] = parseRequest(requestExample);
```

Com a exemple molt bàsic, revisa el paquet "pointfree" del projecte "AlgebraicStructures".

Per altra banda, **"Railway Oriented Programming" (ROP)** és un enfocament per a la programació que proposa un model per a la gestió d'errors i el control de flux que fa servir una metàfora ferroviària. La idea bàsica d'ROP és que cada funció o mètode que s'executa en un programa pot tenir un resultat exitós o fallit. Aquest estil de maneig d'errors utilitza un comportament monàdic: una manera substitutiva de gestionar els errors.





This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Com a exemple en Java, revisa el projecte de NetBeans "RailwayOrientedProgramming".