Type Driven Development & Gestion des Erreurs

(1h30min)

Baptiste FAMCHON & Quentin BURG - QSI

AU MENU CE JOUR

Type Driven Development

Types : c'est quoi ?

Type Driven Development : Value Object, Branded Type & cie

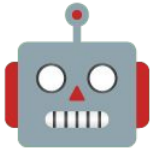
Gestion des erreurs

try / catch : le GOTO déguisé

Modéliser une erreur : comment éviter le try / catch ?

Modéliser l'absence de valeur : éviter les nullPointerException 😌

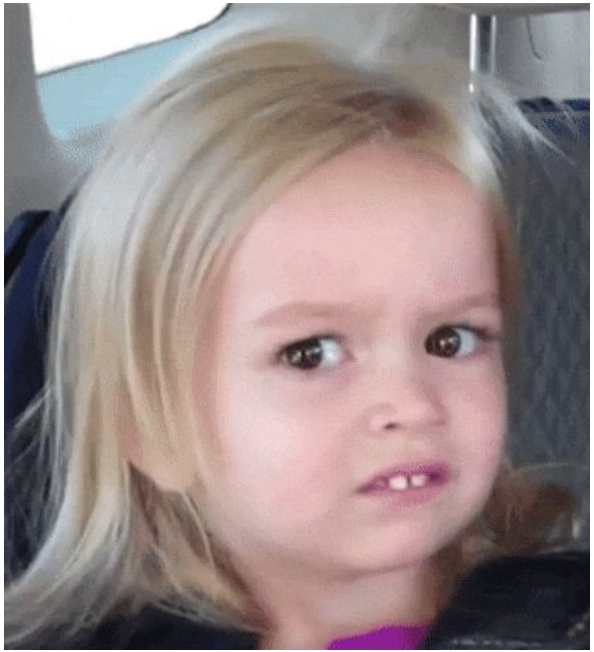
Type Driven Development



TYPE

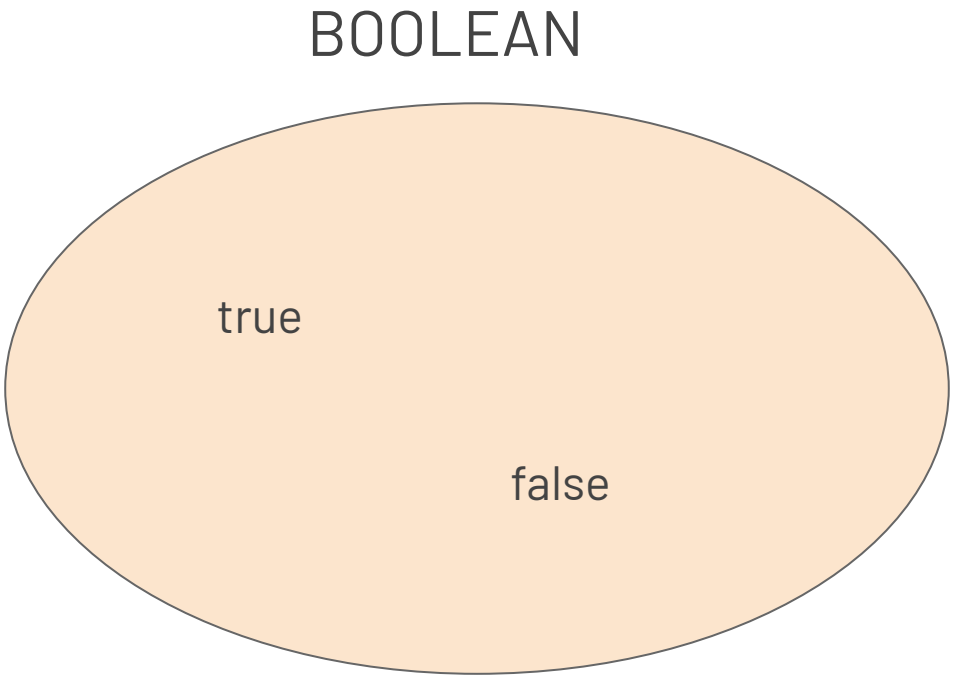
Késako ?

Mais au fait … c’est quoi un type ?



TYPE

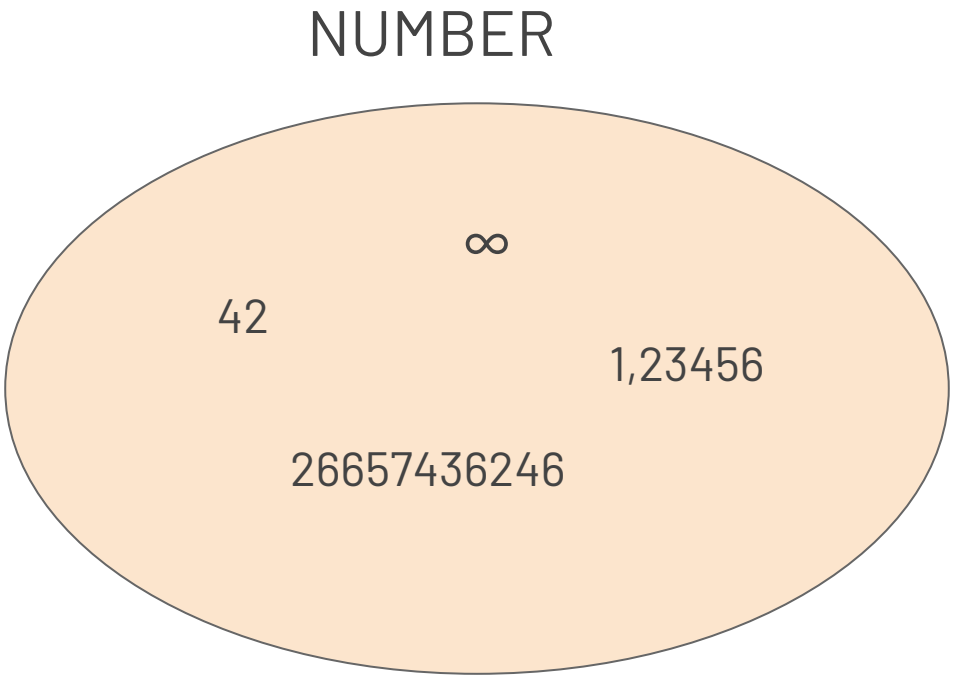
Késako ?



2 valeurs

TYPE

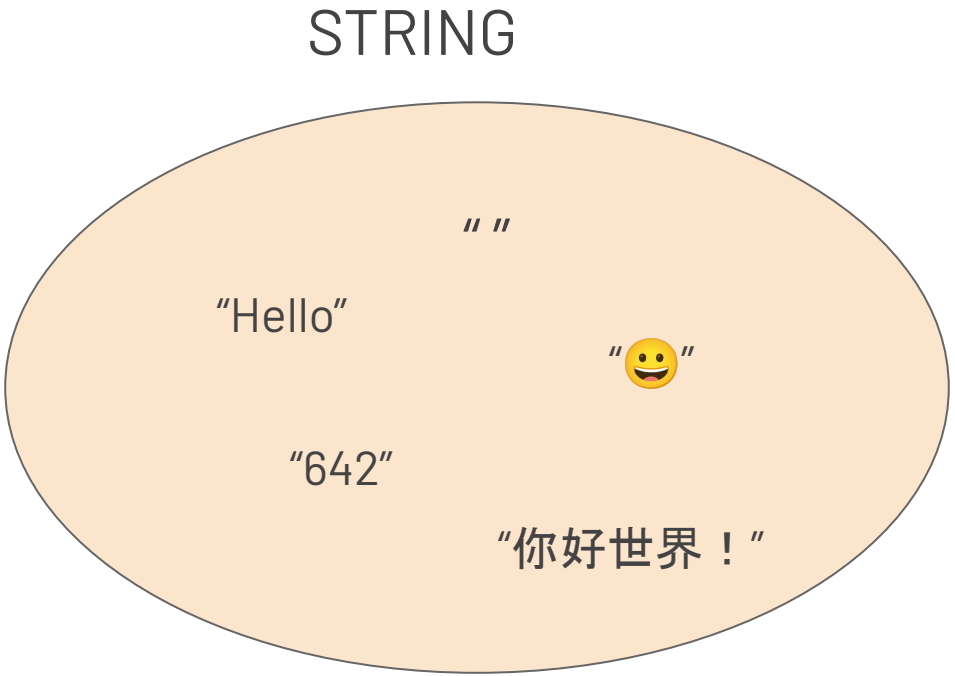
Késako ?



∞ valeurs

TYPE

Késako ?



∞ valeurs

TYPE

Késako ?

Un type est un ensemble de valeurs possibles.

Il existe des **types primitifs** (string, boolean, integer), des **types composés** (union, intersection), ...

Chacun de ces types a une cardinalité, qui représente le nombre de valeurs possibles dans l'ensemble.

Ok, et donc ... ?

TYPE

Exemple avec un status

Quel type peut-on définir pour un statut qui peut valoir “draft”, “validated” ou “paid” ?

```
enum Status {
  DRAFT = "DRAFT",
  VALIDATED = "VALIDATED",
  PAID = "PAID"
}
```

Enumération

```
type Status = "DRAFT" | "VALIDATED" | "PAID"
```

Type string literal

TYPE

Lien avec métier

Dans l'exemple précédent, si j'encode Status avec une **string**, je me retrouve avec une **infinité de valeurs possibles** (CARDINALITE infinie).

Alors que si j'utilise un **type plus restrictif** (type union), je n'ai plus que **3 valeurs possibles** (CARDINALITE de 3)

Pour fiabiliser un système en s'aidant des types, il faut correctement les définir et s'assurer que la c**ardinalité de chaque entité métier soit équivalente à la cardinalité du type qui le défini**.

Le fait d'utiliser massivement les types primitifs n'est pas une bonne chose = **primitive obsession** :

- Masque les notions métiers
- Introduit des valeurs impossibles dans notre système

Maintenant si mes entités dépendent de règles business plus complexe, comment faire pour les encoder dans un système de types ?

PROBLEME

Comment encoder des règles métier dans un système de type ?

```
const sendBookingDetailsByEmail = (email: string, booking: Booking) => {
  if (!isValidEmail(email)) {
    throw new Error("Invalid Email")
  }
  if (booking.status !== 'validated') {
    throw new Error("Booking must be validated")
  }
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

Imaginons une application de réservation de salle. Au moment de la validation de la réservation, le client doit avoir un email valide pour lui envoyer les informations de sa réservation.
Naïvement on fera ce check dans une méthode *sendBookingDetails()* ...

Quel serait le problème ?

VALUE OBJECT

Késako ?

```
class Email {
  constructor(public email: string) {
    if (!isValidEmail) {
      throw new Error("Invalid email")
    }
  }
}
```

On va créer une classe Email qui va contenir notre validation.
On **ENCAPSULE** nos données !
C'est un **Value Object** !

On utilisera ce **Value Object** le plus tôt possible dans notre SI pour n'avoir que des emails valides dans notre système.

VALUE OBJECT

Késako ?

```
const sendBookingDetailsByEmail = (email: string, booking: Booking) => {
  if (booking.status !== 'validated') {
    throw new Error("Booking must be validated")
  }
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

Et du coup on simplifie notre méthode précédente !

PROBLEME

Comment définir des types propres à chaque ID métier ?

Côté métier, un identifiant de produit, ce n'est pas la même chose qu'un identifiant d'utilisateur. Pourtant, dans notre code, ils sont de même type.

```
type UserId = string

type ServiceId = string

type ProductId = string
```

```
const getUserById = (id: UserId) => {
  // Logic to retrieve user
}

const getProductById = (id: ProductId) => {
  // Logic to retrieve product
}

const id: UserId = "user1"

getProductById(id) // No type error, but incorrect usage
```

Comment définir des types propres à chaque type d'ID ?

BRANDED TYPE

Késako ?

```ts
type UserId = string & { __brand: "UserId"}

type ServiceId = string & { __brand: "ServiceId"}

type ProductId = string & { __brand: "ProductId"}
```

Un **Branded Type** est un type avec une étiquette. Elle va permettre au compilateur de faire la différence entre plusieurs types

Pour **construire une valeur** ayant pour type un **Branded Type**, on doit définir des builder.

```ts
const generateProductId = (): ProductId => {
  return v4() as ProductId
}
```

NB: Branded Type sont parfois aussi appelés Phantom Type

BRANDED TYPE

Effect

La définition de ces types peut-être fastidieuse. Effect permet d'en construire plus facilement

```ts
import { Brand } from "effect"

// Define a ProductId type branded with a unique identifier
type ProductId = string & Brand.Brand<"ProductId">

// Build a ProductId value
const ProductId = Brand.nominal<ProductId>()

// Build a ProductId value with rules
const ProductId = Brand.refined<ProductId>(
  // Validation to ensure the value is more than 20 characters
  (p) => p.length > 20,
)
```

Effect permet de définir des types mais aussi ses constructeurs et même de lui affecter des règles (avec **refined**)

BRANDED TYPE

Branded Type by Effect VS Value Object

```ts
import { Brand } from "effect"

type Email = string & Brand.Brand<"Email">

const Email = Brand.refined<Email>(
  (s) => isValidEmail(s),
  (s) => Brand.error(`"${s}" is not a valid email`)
)
```

Vous ne reconnaissez rien ? 😁

PROBLEME

Comment représenter les entités au plus près du métier ?

```
type Product = {
  id: ProductId;
  name: string;
  type: "room" | "service";
  unitPrice: number;
  details: any
}
```

On a un type Product qui représente un produit de type “room” ou “service”. Le problème étant que les détails de ce produit seront différents selon son type !
On a un type “any” pour la propriété “details” ...

PROBLEME

Comment représenter les entités au plus près du métier ?

```
type Booking = {
  items: BookingItems[];
  status: "draft" | "validated" | "paid";
  customer: Customer;
  paymentDate?: Datetime;
}
```

On peut avoir aussi un type Booking représentant une réservation, qui a une date de paiement optionnelle car elle n'est remplie que lorsque la réservation est payée !

PROBLEME

Comment représenter les entités au plus près du métier ?

➡️ Cela traduit une mauvaise conception et peut introduire des bugs : état incohérent, donnée manquante, ...

Comment faire ? 🤔

DISCRIMINATED UNION

Késako ?

```ts
type RoomProduct = {
  id: ProductId;
  name: string;
  type: "room"
  unitPrice: number;
  details: {
    from: Date;
    to: Date;
    roomId: RoomId;
  }
}
```

```ts
type ServiceProduct = {
  id: ProductId;
  name: string;
  type: "service"
  unitPrice: number;
  details: {
    serviceId: ServiceId;
    count: number;
  }
}
```

```ts
type Product = RoomProduct | ServiceProduct;
```

Séparation du type Product en Discriminated Union : **RoomProduct** OU **ServiceProduct**

Une union discriminée est un **type union (somme)** pour laquelle chaque membre a au moins une propriété commune : le **discriminant**

DISCRIMINATED UNION

Késako ?

```
type DraftBooking = {
  items: BookingItems[];
  status: "draft";
  customer?: {
    email: string;
    address: string;
  }
}
```

```
type ValidatedBooking = {
  items: BookingItems[];
  status: "validated";
  customer: Customer;
}
```

```
type PaidBooking = {
  items: BookingItems[];
  status: "paid";
  customer: Customer;
  paymentDate: DateTime;
}
```

```
type Booking = DraftBooking | ValidatedBooking | PaidBooking;
```

Séparation du type Booking en Discriminated Union suivant le status :

- Donne du sens métier aux types
- Permet de ne représenter seulement les données valides de notre système

DISCRIMINATED UNION

Eh, mais au fait ...

On avait simplifié cette méthode en supprimant la validation de l'email pour éviter de la programmation défensive...

Est-ce qu'avec ce que l'on vient de voir, on ne pourrait pas encore simplifier les choses ?

```
const sendBookingDetailsByEmail = (email: string, booking: Booking) => {
  if (booking.status !== 'validated') {
    throw new Error("Booking must be validated")
  }
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

DISCRIMINATED UNION

... Ah ! Voilà qui est mieux !

```
const sendBookingDetailsByEmail = (email: string, booking: ValidatedBooking) => {
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

En spécifiant le type du paramètre, on s'assure que le booking sera forcément validé et donc plus besoin de la condition de validation dans la méthode 🎉🎉

Il reste néanmoins une condition ... A votre avis, comment pourrait-on s'en passer ?

NON EMPTY ARRAY

Tuples

```
type NonEmptyArray = [number, number]
```

Tuple (tableau) de deux nombres

Et du coup, un tableau non vide de nombres ?

```
type NonEmptyArray = [number, ...number[]]
```

Cool, mais on a des booking nous, pas des nombres ... 😒 on fait comment ?

NON EMPTY ARRAY

Généricité

```ts
type NonEmptyArray<T> = [T, ...T[]]
```

En utilisant un **type générique**, on est maintenant capable de représenter un tableau avec au moins 1 élément, quelque soit le type de cet élément !

```ts
type ValidatedBooking = {
  items: NonEmptyArray<BookingItems>;
  status: "validated";
  customer: Customer;
}
```

```ts
const sendBookingDetailsByEmail = (email: string, booking: ValidatedBooking) => {
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

NON EMPTY ARRAY

On simplifie encore notre fonction !

```ts
const sendBookingDetailsByEmail = (email: string, booking: ValidatedBooking) => {
  if (booking.items.length === 0) {
    throw new Error("No items in the booking")
  }
  sendBooking(email, booking)
}
```

⬇️

```ts
const sendBookingDetailsByEmail = (email: string, booking: ValidatedBooking) => {
  sendBooking(email, booking)
}
```

On est passé d'une **programmation défensive à une programmation avec des types réfléchis**, au plus près du métier, pour restreindre les valeurs possibles qui entrent dans notre système.

La puissance du **Type Driven Development** 🔥 ✨

TDD

T pour TYPE et non TEST

Le Type Driven Development est une méthode de développement qui consiste à itérer en étant guidé par les types.

- ✅ Boucle de feedback très rapide (Red, Green, Refactor comme TDD)
- ✅ Réduction de la couverture de tests nécessaires
- ✅ Terminologie métier reprise dans les noms des types (💡 DDD)
- ✅ Réduction des bugs au runtime

Les **types** et les **tests** sont deux méthodes **complémentaires** pour développer des **logiciels de qualité.**
Les tests sont nécessaires lorsque l'on ne peut pas représenter le business à travers le système de types.

SUM UP 🤔

✅ Un **type** est un **ensemble de valeurs possibles** dont le nombre de ces valeurs s'appelle : la **CARDINALITE**

✅ Dans un système, il faut **restreindre la cardinalité de nos types à celles des notions métiers**

✅ Pour représenter une entité métier on peut utiliser des **Branded Types**, des **Value Object** ou des **Discriminated Union**

✅ Les **types "somme" et "produit"** forment ce qu'on appelle les **types algébriques**

En conclusion ...

- Avoir un système de type complet et réfléchi dans son SI n'est pas anodin sur la qualité. Il permet de gagner en fiabilité, lisibilité, maintenabilité et faire l'économie de tests !
- Composez vos types à l'aide des types OU, ET ➡️ ADT
- Servez-vous du système de types des langages (Branded Type, Discriminated Union) pour encoder un maximum de règles business, afin de rendre votre app plus robuste et de vous économisez des tests unitaires !
- Pratiquez le TypeDD afin de gagner en efficacité

Dans la suite du cours, on va voir comment les types peuvent nous permettre d'encoder correctement les erreurs... 👀

GESTION DES ERREURS

LES BUGS, ÇA COÛTE CHER !

Les crashs au runtime peuvent coûter cher aux entreprises ...

Précédemment nous avons vu comment en éviter à l'aide du typage.

Néanmoins les erreurs sont forcément présentes dans un programme et il faut les traiter correctement pour éviter des crashs.

Mais au fait ... c'est quoi une erreur ?

- ➡️ Une absence de valeur
- ➡️ Un chemin alternatif dans le flux d'exécution
- ➡️ Une exception au runtime

TRY / CATCH

Spécifications - Validation d'une commande

Valider son panier

Procéder au paiement

Confirmer la commande

```
class Cart {
  constructor() {
    throw new Error("Cart empty");
  }
}

class Payment {
  constructor() {
    throw new Error("No enough money");
  }
}

class OrderConfirmed {}

const validateCommand = (cart: Cart, payment: Payment) => {
  return new OrderConfirmed();
};
```

TRY / CATCH

Un cas simple

```
const cart = new Cart();
const payment = new Payment();

validateCommand(cart, payment);
```

Evidemment ce code va crasher au runtime mais ...
Est-ce qu'il compile ?

TRY / CATCH

Un problème de type...

```
export const handleLoginUser = async (username: string, password: string): Promise<User> => {
  if (username === '') {
    throw new EmptyUsernameLoginError();
  }
  if (password === '') {
    throw new EmptyPassswordLoginError();
  }
  if (isCorrectUserPasswordCombo(username, password) === false) {
    throw new InvalidCredentialsError();
  }
  const user = await getUserbyUsername(username);
  if (user.active === false) {
    throw new InactiveUserError();
  }
  return user;
}
```

Le type inféré de cette fonction est simplement Promise<User>
Cela n'indique en rien les exceptions qui peuvent être levées durant l'exécution de cette fonction...
C'est problématique !!

TRY / CATCH

Mais alors c'est quoi le vrai problème ?

Le problème ici c'est que les exceptions levées sont des évènements business que le système sait traiter.

Dans ces cas, lever une exception n'est pas la meilleure approche...

TRY / CATCH

Le GOTO déguisé

En réalité, lever une exception est un mauvais choix pour :

- ❌ Modéliser une absence de valeur
- ❌ Modéliser une erreur fonctionnelle
- ❌ Modéliser une erreur asynchrone

Par contre, vous pouvez si :

- ✅ Une erreur n'est pas récupérable par le système (nécessite une intervention manuelle par exemple)

MODÉLISER UNE ERREUR POTENTIELLE

Une erreur, c'est quoi ?

Dans un programme, un traitement peut :

- Réussir à produire une valeur (**succès**)
- Produire une erreur (**échec**)

Les traitements pouvant donner lieu à des erreurs sont de type ...

SOMME

MODÉLISER UNE ERREUR POTENTIELLE

Rappel de la problématique liée au try / catch

❌ **Absence de typage sur la valeur de retour** lorsqu'un traitement peut échouer en levant une exception

❌ try / catch est une **instruction**, qui **casse le flux normal d'exécution** et va appeler un autre traitement alternatif (GOTO)
Dans une petite application, ce n'est pas un problème mais sur une codebase conséquence, celui devient difficile à manipuler

❌ **Difficile d'être exhaustif** sur le traitement des erreurs

❌ **En asynchrone, des erreurs peuvent être "perdues".** Le traitement peut être un succès en apparence mais être un échec en réalité car l'erreur n'est pas traitée...

MODÉLISER UNE ERREUR POTENTIELLE

RESULT

```ts
type Error<E> = { kind: "Error", error: E }
type Ok<A> = { kind: "Ok", ok: A }
type Result<E, A> = Error<E> | Ok<A>

const isNat = (x: number): Result<string, number> =>
  x > 0
    ? { kind: "Ok", ok: x }
    : { kind: "Error", error: "Not a natural number" }

const foldNumber = (y: Result<string, number>, defaultValue: number): number => {
  switch (y.kind) {
    case "Ok": return y.ok;
    case "Error": return defaultValue;
  }
}
```

Un **RESULT** est un type **SOMME**, qui vaut soit `Error<E>` si le traitement a échoué ou `Ok<A>` si le traitement a réussi.

MODÉLISER UNE ERREUR POTENTIELLE

EITHER = RESULT

```ts
type Left<E> = { kind: "Left", left: E }
type Right<A> = { kind: "Right", right: A }
type Either<E, A> = Left<E> | Right<A>

const isNat = (x: number): Either<string, number> =>
  x > 0
    ? { kind: "Right", right: x }
    : { kind: "Left", left: "Not a natural number" }

const foldNumber = (y: Either<string, number>, defaultValue: number): number => {
  switch (y.kind) {
    case "Right": return y.right;
    case "Left": return defaultValue;
  }
}
```

EITHER est plus générique, il peut représenter n'importe quel arbre binaire.
Par convention, les membres sont `Right<A>` (correct) contenant la valeur **Ok** et `Left<E>` (abandon) contenant la valeur **Error**

MODÉLISER UNE ERREUR POTENTIELLE

EITHER avec fp-ts (ou Effect)

```ts
import * as E from 'fp-ts/Either'
import { Either } from 'fp-ts/Either'
import { pipe } from "fp-ts/lib/function";

const isNat = (x: number): Either<string, number> =>
  x > 0
    ? E.right(x)
    : E.left("Not a natural number")

const foldNumber = (y: Either<string, number>, defaultValue: number): number =>
  pipe(
    y,
    E.match(
      (_: string) => defaultValue, // onLeft handler
      (value: number) => value // onRight handler
    )
  )
```

Fp-ts (ou Effect) fournit des fonctions utilitaires pour Either, comme le match expressif

MODÉLISER UNE ERREUR POTENTIELLE

Et JAVA alors ?

```
sealed interface Result<T,E> {
    record Ok<T,E>(T ok) implements Result<T,E> {
        public Ok {
            java.util.Objects.requireNonNull(ok);
        }
    }
    record Err<T,E>(E error) implements Result<T,E> {
        public Err {
            java.util.Objects.requireNonNull(error);
        }
    }
    public static<T> Ok ok(T ok){
        return new Ok(ok);
    }
    public static<E> Err err(E error){
        return new Err(error);
    }
}

record Weapon(String name){
    public Weapon {
        java.util.Objects.requireNonNull(name);
    }
    public static Weapon weapon(String name){
        return new Weapon(name);
    }
}

public class Main
{
    public static Result<Weapon, Exception> mustCarryABow(Weapon w){
        return w.name().equals("bow") ? Result.ok(w) : Result.err(new Exception("bow not carried"));
    }
    public static void main(String[] args) {
        var myWeapon = Weapon.weapon("bow");
        switch (mustCarryABow(myWeapon)){
            case Result.Ok<Weapon, Exception> o -> System.out.println("Cool you carry a ".concat(o.ok().name()));
            case Result.Err<Weapon, Exception> e -> System.out.println("Error:".concat(e.error().getMessage()));
        }
    }
}
```

En Java ni Either ni Result n'existent dans la lib standard.

On peut néanmoins l'encoder assez rapidement avec les génériques. Il faudrait implémenter les fonctions utilitaires.

Pour faire du Java avancé, vous pouvez utiliser Vavr, qui fournit notamment une implémentation d'Either

SUM UP 🤔

✅ RESULT / EITHER sont utiles pour modéliser les cas d'erreur

✅ Sécurisant car typé

✅ Facile à manipuler avec leurs fonctions utilitaires

✅ Parfois encodé avec un seul paramètre pour homogénéiser les erreurs, dans ce cas il s'appelle Try

```
interface MyDomainErrors extends Error { };
type Try<A> = Either<MyDomainErrors, A>;
```

MODÉLISER L'ABSENCE DE VALEUR

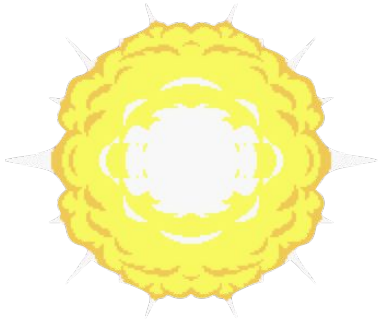
Une valeur pour semer la désolation : null


```java
class Cart {}
class Payment {}
class OrderConfirmed{}

public class Main {
    static Cart cart = null;
    static Payment payment = null;
    static OrderConfirmed validateCommand(Cart c, Payment p) {
        return null;
    }
    public static void main(String[] args) {
    validateCommand(cart, payment).toString();
    }
}
```

Est-ce que ce code compile à votre avis ?

MODÉLISER L'ABSENCE DE VALEUR

Une valeur pour semer la désolation : null



```
Exception in thread "main" java.lang.NullPointerException
```

OUI 😱 😭

MODÉLISER L'ABSENCE DE VALEUR

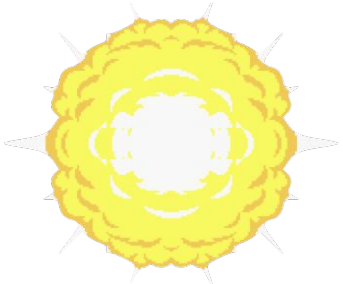
Une valeur pour semer la désolation : null

```
class Cart {}
class Payment {}
class OrderConfirmed {}

const validateCommand = (cart?: Cart, payment?: Payment) => {
  return new OrderConfirmed();
};

const cart = null;
const payment = null;

validateCommand(cart, payment).toString();
```

Et celui-ci, est-ce qu'il compile ?

MODÉLISER L'ABSENCE DE VALEUR

Une valeur pour semer la désolation : null



```
validateCommand(...) is null
```

Sans `strictNullChecks` 💥

```ts
class Cart {}
class Payment {}
class OrderConfirmed {}

const validateCommand = (cart?: Cart, payment?: Payment) => {
  return new OrderConfirmed();
};

const cart = null;
const payment = null;

validateCommand(cart, payment).toString();
```

```
Object is possibly 'null'.
```

Avec `strictNullChecks` 

Cela dépend de votre config TS 🤔

MODÉLISER L'ABSENCE DE VALEUR

Traiter les valeurs optionnelles en Java

```
sealed interface Option<A> {
  record None<A> implements Option<A>
  record Some<A>(A value) implements Option<A> {
    public Some {
      java.utils.Objects.requireNonNull(value)
    }
  }
}
```

Option<A> est un type OU

None<A> représente l'absence de valeur pour le type **Option<A>**. Ce type a exactement 1 valeur pour l'ensemble **A** donné.

Some<A> représente une valeur de type **A**. Ce type a autant de valeur que l'ensemble **A**

En Java, vous pouvez utiliser **Optional** pour décrire l'absence ou non d'une valeur !

L'utilisation d'Optional peut-être fastidieuse dû au manque d'expressivité de Java. Néanmoins, il existe des **fonctions utilitaires** pour manipuler ce type.

Map et flatMap permettent de manipuler les valeurs optionnelles (il en existe d'autres => RTFM)

```
static Optional<Cart> cart = new Optional.empty();
static Optional<Payment> payment = new Optional.empty();
static Optional<OrderConfirmed> validateOrder(
    Optional<Cart> cart, Optional<Payment> payment
) {
  return cart.flatMap(c -> payment.map(p -> new OrderConfirmed()))
}
```

MODÉLISER L'ABSENCE DE VALEUR

Traiter les valeurs optionnelles en TS

```
const validateCommand = (cart: Cart | null, payment: Payment | null):
  OrderConfirmed | null =>
    cart && payment ?
      new OrderConfirmed()
      : null;
```

`null` est un **type littéral** qui a une valeur: `null` (c'est à la fois un type et une valeur, suivant là où on l'utilise)

Si l'on ajoute `strictNullCheck` à `true` dans la config TS, on a vu que les erreurs potentielles liées à null était remontées

Mais il y a beaucoup de confusion sémantique entre `undefined` (valeur non initialisée) et `null` (absence de valeur), ce qui amène encore certaines erreurs

➡️ **La manipulation de type nullable et leur expressivité est perfectible**

MODÉLISER L'ABSENCE DE VALEUR

Traiter les valeurs optionnelles en TS

```
const validateCommand = (cart: Option<Cart>, payment: Option<Payment>):
  Option<OrderConfirmed> =>
    (O.iSome(cart) && O.iSome(payment)) ?
      O.some(new OrderConfirmed())
      : O.none;
```

La librairie **Effect** fournit un module `Option<A>` permettant de modéliser une absence de valeur.

Evidemment sur cet exemple trivial, l'apport est limité.

MODÉLISER L'ABSENCE DE VALEUR

Traiter les valeurs optionnelles en TS

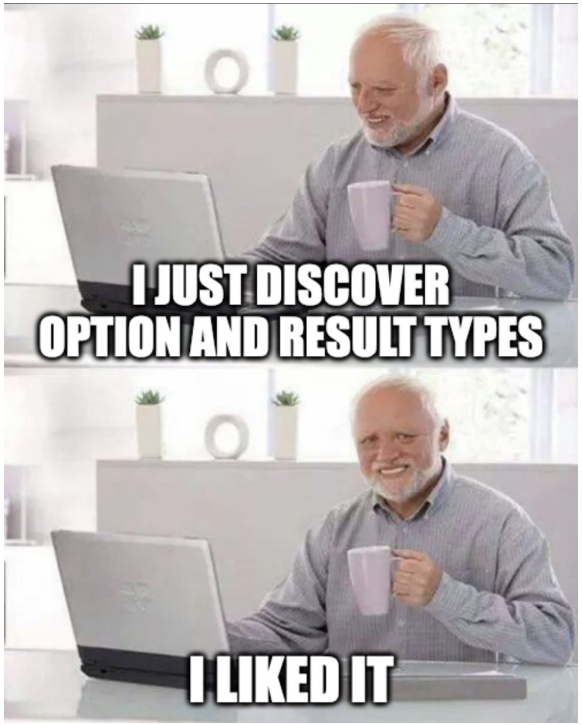
```
const createOrderConfirmer = (c: Cart, p: Payment) => new OrderConfirmed();

const validateCommand = (cart: Option<Cart>, payment: Option<Payment>):
  Option<OrderConfirmed> =>
    O.chain(
      (c: Cart) => O.map(
        (p: Payment) => createOrderConfirmer(c, p),
          payment
      ),
      cart
    )
```

On retrouve les fonctions utilitaires **map** et **chain** (aka flapMap en Java)

```
Map : <A,B> (f: (a: A) => B)  => (oa: Option<A>) => Option<B>

Chain : <A,B> (f: (a: A) => Option<B>) => (oa : Option<A>) => Option<B>
```

En conclusion ...

- OPTION pour modéliser une absence de valeur
- RESULT / EITHER sont utiles pour modéliser les cas d'erreur
- Des librairies existent Effect, servez-vous en ! (ou VAVR pour les Java-istes 😜)



Ressources

- 🌐 https://gcanti.github.io/fp-ts/
- 🌐 https://effect.website/docs/code-style/branded-types/
- 📽️ Types vs Tests https://www.youtube.com/watch?v=HTvII2QWqFo
- 📽️ Le design de l'erreur https://www.youtube.com/watch?v=MSqZyqItgfE

À VOUS DE JOUER