Clean Architecture

Amin NAIRI <anairi@esgi.fr>

Séance 1 : TypeScript

- Types de base
- Interface, type alias
- Type optionnel et union
- Record
- Typage de fonction
- Générique
- Function et type spécifique

Séance 1 : TypeScript — Types de base

Tous les types existants en JavaScript sont utilisés comme annotations en TypeScript (string, number, boolean, …).

Il est possible d'annoter les variables, constantes, arguments, retours de fonctions en TypeScript, permettant d'améliorer l'inférence et l'aide apportée par le compilateur pour nous éviter des erreurs de types au runtime.

Nous verrons plus tard comment typer des choses plus complexes comme des objets et des tableaux.

```ts
const age: number = 42;

const isAbleToDrive: boolean = true;

const message: string = "You have 8 unread messages.";

const id: symbol = Symbol("UNIQUE_IDENTIFIER");

const price = 13.82;

const isAdministrator = false;

const severity = "NOTICE";

const kind = Symbol("ITEM_KIND");
```

Séance 1 : TypeScript — Interface et type

Afin de pouvoir typer des objets en TypeScript, nous pouvons utiliser le mot-clé “object” qui permet d’indiquer que cette variable attend uniquement un objet.

Cependant, typer un objet de cette manière ne nous donne pas beaucoup d’information précise sur sa forme, il est donc plutôt recommandé d’utiliser une interface à la place, qui a exactement le même rôle qu’une interface dans un langage orienté objet.

Il est également possible d’utiliser un type personnalisé. Attention cependant, il n’est pas possible d’implémenter un type, alors qu’il est possible d’implémenter une interface.

```ts
const amin: object = {
  id: 1,
  username: "anairi",
  email: "anairi@esgi.fr"
};

interface User {
  id: number,
  username: string,
  email: string
}

const matteo: User = {
  id: 2,
  username: "matteo",
  email: "mdelandhuy@esgi.fr"
};

type Price = {
  value: number,
  currency: "EUR" | "USD" | "CAD"
};

const shinyShoePrice: Price = {
  value: 57.23,
  currency: "CAD"
};
```

Séance 1 : TypeScript — Type optionnel et union

Il peut être intéressant, parfois, de rendre certains types optionnels, notamment dans le cas d'un argument de fonction qui a besoin d'accepter certains arguments, mais pas forcément tous en même temps.

Pour cela, il est possible d'utiliser un point d'interrogation avant l'annotation de type, afin de faire comprendre au compilateur que nous souhaitons récupérer un type, ou rien du tout.

À l'inverse, nous aimerions occasionnellement pouvoir accepter plus d'un type à la fois pour une même variable ou argument. Pour cela, il est possible d'utiliser une barre verticale dans le but d'aligner toutes les possibilités possibles pour un type.

```
interface Registration {
  email: string,
  password: string,
  favoriteLanguage: "Rust" | "Elm" | "JavaScript",
  firstname?: string,
  lastname?: string
}

const firstRegistration: Registration = {
  email: "anairi@esgi.fr",
  favoriteLanguage: "Elm",
  password: "supersecretpassword"
};

const secondRegistration: Registration = {
  email: "mdelandhuy@esgi.fr",
  password: "supersecretpassword",
  favoriteLanguage: "Rust",
  firstname: "Mattéo",
  lastname: "DELANDHUY"
};
```

Séance 1 : TypeScript — Record

Nous avons déjà vu plusieurs façons de représenter un objet : object, Type ou Interface.

Mais il existe une autre manière de représenter un objet, notamment lorsque nous connaissons sa forme globale, mais pas exacte.

Par exemple, si nous devons représenter une interface représentant les en-têtes HTTP, il existe certes des en-têtes standards que l'on connait à l'avance, mais il est également possible d'avoir des en-têtes non-standard, qui nous servent pour notre métier.

Pour cela, un Record est pratique et permet d'obtenir une forme générique, plutôt qu'un ensemble concret de propriétés.

```
interface BadHeaders {
  accept: string,
  authorization: string,
  encoding: string
}

type Headers = Record<string, string>;
```

Séance 1 : TypeScript — Typage de fonction

Il est tout à fait possible d'utiliser ce que nous avons vu jusqu'à présent pour typer nos fonctions.

Il est possible de typer les arguments qui sont acceptés par une fonction, et il est également possible de typer le retour d'une fonction.

Bien qu'il soit possible de se reposer sur l'inférence de type, il est plutôt recommandé, pour les fonctions, de toujours typer tous les arguments et les retours de fonction, afin d'être sûr que le compilateur ne devine pas quelque d'incorrect.

Enfin, il est tout à fait possible d'utiliser une interface, pour les fonctions qui prennent beaucoup d'arguments, sous la forme d'un argument objet par exemple.

```ts
function add(first: number, second: number): number {
  return first + second;
}

interface UseFetchOptions {
  url: string,
  method: "GET" | "POST" | "PUT" | "PATCH" | "DELETE",
  headers: Record<string, string>,
  signal: AbortSignal,
  body?: string
}

function useFetch({ url, ...options }: UseFetchOptions): Promise<Response> {
  return fetch(url, options);
}
```

Séance 1 : TypeScript — Générique

Comme pour le langage C, il est généralement conseillé d'avoir des tableaux de types homogènes, c'est-à-dire qui ne prennent pas plusieurs formes. Par exemple, un tableau de nombres, un tableau de chaînes de caractères, mais pas les deux en même temps.

Pour cela, il est possible d'utiliser deux syntaxes : l'une avec un tableau juste après l'annotation de type, et une autre qui utilise une syntaxe spéciale qu'on appelle un type générique.

Un type générique est un type personnalisé, qui va pouvoir accepter un type quelconque. C'est très pratique, notamment dans les fonctions, pour éviter de devoir créer une fonction permettant de trier des chaînes de caractères uniquement, ou des nombres uniquement, ou des utilisateurs…

```ts
const numbers: number[] = [1, 2, 3, 4, 5];

const prices: Array<number> = [12.82, 84.28, 4.29];

interface CanadianCurrency<Value> {
  value: Value,
  currency: "CAD"
}

interface AmericanCurrency<Value> {
  value: Value,
  currency: "USD"
}

type Currency<Value> = CanadianCurrency<Value> | AmericanCurrency<Value>

function sortStrings(strings: string[]): string[] {
  //...
}

function sortNumbers(numbers: number[]): number[] {
  //...
}

function sort<Value>(items: Value[]): Value[] {
  //...
}

sortNumbers([4, 7, 2, 5, 8]);
sortStrings(["a", "t", "z", "h", "e"]);

sort([4, 7, 2, 5, 8]);
sort(["a", "t", "z", "h", "e"]);
```

Séance 1 : TypeScript — Function et type spécifique

Accepter des fonctions en argument peut être intéressant, notamment lorsque l'on créera des fonctions d'ordre supérieur, concept que l'on détaillera un peu plus tard.

Par exemple, si nous souhaitons avoir notre propre fonction "map", homologue à la fonction Array.prototype.map, il serait intéressant de typer nos arguments, même lorsqu'il s'agit d'une fonction.

Il est possible d'utiliser le type "Function", qui est un type très large et qui accepte n'importe quelle fonction.

Mais il est également possible de créer son propre type qui sera une fonction, et qui s'écrit comme une fonction fléchée en JavaScript.

```
function filter<Value>(accept: Function, items: Array<Value>): Array<Value> {
  if (items.length === 0) {
    return [];
  }

  const [item, ...remainingItems] = items;

  const acceptedItem = accept(item);
  const nextItems = filter(accept, remainingItems);

  if (acceptedItem) {
    return [
      acceptedItem,
      ...nextItems
    ];
  }

  return nextItems;
}

type MapUpdate<Value, NewValue> = (value: Value) => NewValue

function map<Value, NewValue>(update: MapUpdate<Value, NewValue>, items: Array<Value>): Array<NewValue> {
  if (items.length === 0) {
    return [];
  }

  const [item, ...remainingItems] = items;

  const updatedItem = update(item);
  const nextItems = map(update, remainingItems);

  return [
    updatedItem,
    ...nextItems
  ];
}
```

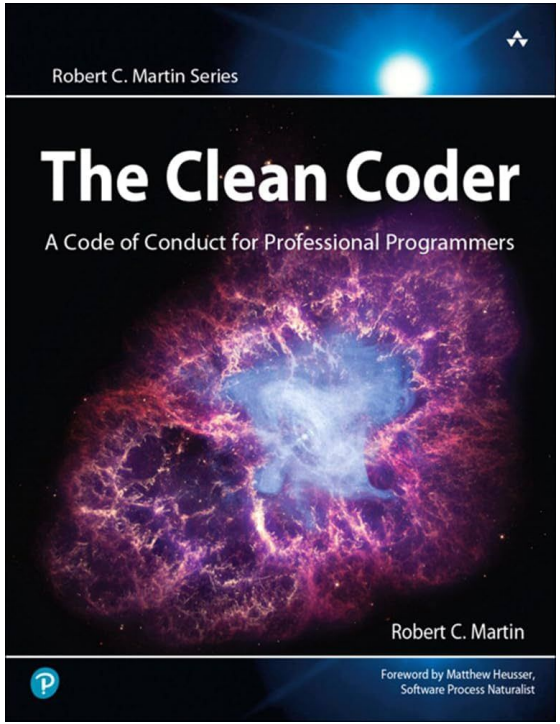
Séance 1 : TypeScript — Exercice

- Coder la définition des fonctions map, filter et reduce, homologue à Array.prototype
- Utiliser des types génériques pour ces trois fonctions
- Utiliser de la récursion à la place des boucles
- Utiliser reduce pour implémenter map et filter au lieu de recoder leur corps de fonction

```
function map(items, update) {}

function filter(items, accept) {}

function reduce(items, initialValue, transform) {}
```

Séance 2 : Clean Code

- Formatage consistent
- Noms explicites
- Responsabilité unique
- Valeurs magiques
- Composition plutôt qu'héritage
- Gérer tous les cas
- Early Return



Séance 2 : Clean Code — Formatage consistent

Il est préférable dans un code-source, notamment lorsqu'on le partage avec d'autres personnes, d'être le plus consistent possible.

Cela passe également par le formatage du code, c'est-à-dire la façon dont est indenté le code, la position des accolades, les espacements entre arguments de fonctions, …

Il existe beaucoup de façon d'écrire un code en TypeScript, et aucune n'est bonne ou mauvaise. Néanmoins, il est important avant de travailler en équipe de se mettre d'accord sur un style de code, les outils comme les linters (ESLint, Prettier) peuvent grandement aider et ne sont pas à sous-estimer !

Séance 2 : Clean Code — Noms explicites

Si vous avez commencé la programmation sur des systèmes embarqués, avec des langages de programmation bas niveau tel que le C, vous avez probablement du prendre l'habitude d'avoir des noms de variables les plus courts possibles, afin de gagner en espace lors d'une future compilation.

Aujourd'hui, les ordinateurs ont largement la puissance et le stockage nécessaire pour embarquer des programmes plus conséquents, et les langages modernes sont capables de faire ce travail d'optimisation pour nous.

Il n'est donc plus nécessaire d'avoir des noms de variables sur une lettre, et c'est notamment une mauvaise pratique.

Privilégiez les noms de variables explicites, même si cela signifie avoir des longs noms.

```
const brd = [
  ["X", "X", "O"],
  ["X", "X", "O"],
  ["X", "X", "O"]
];

for (const x of brd) {
  for (const y of x) {
    // Faire quelque chose pour savoir qui a gagné
  }
}

const tictactoeBoard = [
  ["X", "X", "O"],
  ["X", "X", "O"],
  ["X", "X", "O"]
];

for (const row of tictactoeBoard) {
  for (const cell of row) {
    // Faire quelque chose pour savoir qui a gagné
  }
}
```

Séance 2 : Clean Code — Valeurs magiques

Une valeur magique est une donnée métier qui fait probablement du sens pour un expert du métier, mais qui pourrait être mal, voir pas, interprété correctement par d'autres développeurs.

Par exemple la valeur 3.14159, qui a sûrement tout de suite fait écho aux personnes ayant un background en mathématiques puisqu'il s'agit des premiers nombres de PI, mais qui pour les personnes qui ne savent pas de quoi il s'agit pourrait empêcher de comprendre correctement la signification de ces données, et donc du programme dans son ensemble.

Lorsque vous rencontrez ce genre de données, n'hésitez pas à créer leur contrepartie sous la forme de constantes ou de variables afin de leur donner un nom clair et explicite.

```
class BadCircle {
  constructor(public radius: number) {}

  calculateArea(): number {
    return 3.14159 * this.radius * this.radius;
  }
}

class GoodCircle {
  private static readonly PI = 3.14159;

  constructor(public radius: number) {}

  calculateArea(): number {
    return GoodCircle.PI * this.radius * this.radius;
  }
}
```

Séance 2 : Clean Code — Composition plutôt qu'héritage

Il est possible en TypeScript d'hériter de classes, comme dans la plupart des langages de programmation orientés objets.

Cependant, cela crée souvent des relations très fortes au sein de ces mêmes classes enfants, ce qui a pour but de rigidifier le code, le rendant moins souple et rendant difficile la factorisation de code, notamment lorsqu'un besoin métier évolue drastiquement.

De plus, l'héritage multiple est un concept qui n'existe dans quasiment aucun langage de programmation orienté objet, pour de bonnes raisons, mais cela empêche également une certaine souplesse qui est parfois nécessaire.

C'est pourquoi, il est préférable de composer nos fonctions et nos méthodes de classes, afin de rendre le code plus simple à retravailler si nécessaire.

```
class Animal {
  constructor(public name: string) {}

  move(): void {
    console.log(`${this.name} se déplace.`);
  }
}

class Bird extends Animal {
  fly(): void {
    console.log(`${this.name} vole.`);
  }
}

class Fish extends Animal {
  swim(): void {
    console.log(`${this.name} nage.`);
  }
}
```

Séance 2 : Clean Code — Composition plutôt qu'héritage

Il est possible en TypeScript d'hériter de classes, comme dans la plupart des langages de programmation orientés objets.

Cependant, cela crée souvent des relations très fortes au sein de ces mêmes classes enfants, ce qui a pour but de rigidifier le code, le rendant moins souple et rendant difficile la factorisation de code, notamment lorsqu'un besoin métier évolue drastiquement.

De plus, l'héritage multiple est un concept qui n'existe dans quasiment aucun langage de programmation orienté objet, pour de bonnes raisons, mais cela empêche également une certaine souplesse qui est parfois nécessaire.

C'est pourquoi, il est préférable de composer nos fonctions et nos méthodes de classes, afin de rendre le code plus simple à retravailler si nécessaire.

```typescript
class Animal {
  constructor(public name: string) {}

  move(): void {
    console.log(`${this.name} se déplace.`);
  }
}

class Flyer {
  fly(name: string): void {
    console.log(`${name} vole.`);
  }
}

class Swimmer {
  swim(name: string): void {
    console.log(`${name} nage.`);
  }
}

// Composition avec le canard
class Duck {
  private flyer: Flyer;
  private swimmer: Swimmer;
  private animal: Animal;

  constructor(name: string) {
    this.animal = new Animal(name);
    this.flyer = new Flyer();
    this.swimmer = new Swimmer();
  }
}
```

Séance 2 : Clean Code — Gérer tous les cas

Bien que cela reste difficile à faire sans outils ou librairie, il est extrêmement important de savoir gérer tous les cas possibles que peut prendre nos données dans un programme.

Un cas qui ne serait pas géré peut, dans le meilleur des cas, entraîner des incohérences dans nos interfaces, voir pire, l'arrêt du fonctionnement total de nos applications.

Il est important d'avancer uniquement lorsque nous sommes sûrs de pouvoir gérer tous les cas, notamment pour une promesse par exemple.

Il existe des librairies, comme exhaustive, nous permettant de nous aider à gérer tous les cas possibles, néanmoins elle n'est pas parfaite dans certains cas plus complexes que nous rencontrerons, mais aide beaucoup dans tous les autres.

```ts
import { exhaustive } from "exhaustive";

interface User {
  email: string,
  username: string,
  state: "WAITING_CONFIRMATION" | "CONFIRMED" | "BANNED"
}

const justCreatedUser: User = {
  email: "email@domain.com",
  username: "someone",
  state: "WAITING_CONFIRMATION"
}

const message = exhaustive(justCreatedUser.state, {
  "WAITING_CONFIRMATION": () => {
    return "Please, confirm your account before continuing"
  },
  "CONFIRMED": () => {
    return "Welcome, user!"
  },
  "BANNED": () => {
    return "You account has been banned"
  }
});
```

Séance 2 : Clean Code — Early Return

Cette astuce est à la fois simple à mettre en place, et permet dans la plupart des cas de gagner en fiabilité dans notre code.

Lorsque nous avons à gérer plusieurs cas différents, il peut être difficile de se relire lorsque le code comporte plusieurs if-elseif-else.

Dans le pire des cas, cela peut même engendrer des incohérences, comme une requête HTTP qui n'est pas géré dans un certain cas spécifique.

Pour cela, il est préférable d'utiliser de retourner (return) dans notre fonction le plus tôt possible, afin de visuellement voir clairement quels sont les cas que nous avons traités, et quels sont les cas que nous n'avons pas encore traités.

Et cela améliore grandement l'expérience du développeur qui a besoin de relire ce même code, que ce soit vous ou un autre.

```
const server = createServer(async (request, response) ⇒ {
  if (request.method === "POST") {
    if (request.url === "/users") {
      const body = await getBodyFromRequest(request);

      if (typeof body === "object") {
        if (typeof body["email"] === "string") {
          await saveUser(body["email"]);
          response.end("Created.");
        } else {
          response.end("Bad request.");
        }
      } else {
        response.end("Bad request.");
      }
    } else {
      response.end("URL not found.");
    }
  } else {
    response.end("Bad method.");
  }
});
```

Séance 2 : Clean Code — Early Return

Cette astuce est à la fois simple à mettre en place, et permet dans la plupart des cas de gagner en fiabilité dans notre code.

Lorsque nous avons à gérer plusieurs cas différents, il peut être difficile de se relire lorsque le code comporte plusieurs if-elseif-else.

Dans le pire des cas, cela peut même engendrer des incohérences, comme une requête HTTP qui n'est pas géré dans un certain cas spécifique.

Pour cela, il est préférable d'utiliser de retourner (return) dans notre fonction le plus tôt possible, afin de visuellement voir clairement quels sont les cas que nous avons traités, et quels sont les cas que nous n'avons pas encore traités.

Et cela améliore grandement l'expérience du développeur qui a besoin de relire ce même code, que ce soit vous ou un autre.

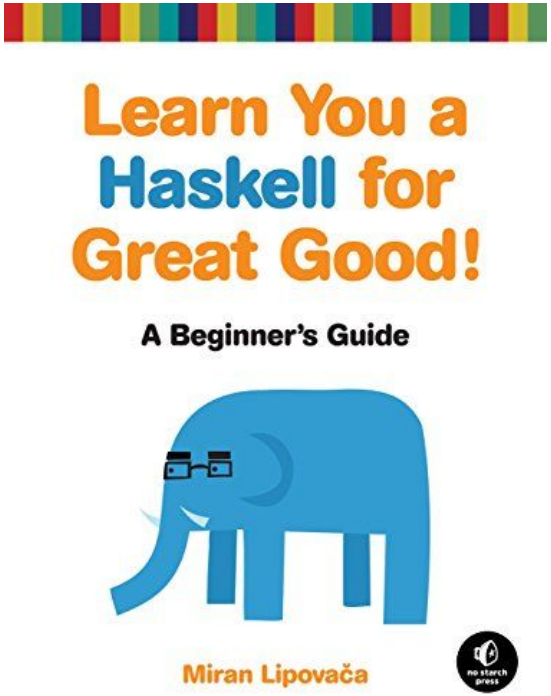
```
const server = createServer(async (request, response) => {
  if (request.method !== "POST") {
    response.end("Bad method.");
    return;
  }

  if (request.url !== "/users") {
    response.end("URL not found.");
    return;
  }

  const body = await getBodyFromRequest(request);

  if (typeof body !== "object" || typeof body["email"] !== "string") {
    response.end("Bad request.");
    return;
  }

  await saveUser(body["email"]);
  response.end("Created.");
});
```

Séance 3 : Contrôle Continu #1

- Durée : 15mn
- Correction : 15mn
- Sujet : non-communiqué
- Objectif : savoir appliquer les bonnes pratiques de Clean Code en TypeScript

Séance 4 : Programmation Fonctionnelle

- Immutabilité
- Effets de bords
- Fonctions pures
- Fonctions d’ordre supérieur
- Récursion
- Result vs Try/Catch



Séance 4 : PF — Immutabilité

L’immutabilité est le fait de ne pas modifier ses données, mais plutôt d’en créer de nouvelles.

Il n’est alors pas question de créer des variables en programmation fonctionnelle, mais uniquement des constantes.

Bien entendu, respecter ce principe à la lettre n’est pas possible complètement, car il y a forcément des moments où l’on doit modifier une ou plusieurs variables.

L’idée est, lorsque c’est possible, d’éviter un maximum les mutations, et de privilégier les constantes.

```
// Programmation impérative

let price = 12;
let vatInPercentage = 20;

price = price + (price * vatInPercentage / 100);

// Programmation fonctionnelle

const price = 12;
const vatInPercentage = 20;
const priceWithVat = price + (price * vatInPercentage / 100);
```

Séance 4 : PF — Effets de bords

Un effet de bords est une action qui aura pour conséquence la modification d'un contexte extérieur.

Par exemple, créer un fichier sur le disque dur, modifier un document HTML, afficher des données dans la console, etc.

Si l'on prend le langage JavaScript dans le contexte d'un navigateur Web, et qu'on lui enlève tous ses effets de bords, il ne reste plus que le langage avec ses structures de données, ses fonctions, sa syntaxe, mais sans rien d'autre d'intéressant.

De manière générale, un effet de bords en JavaScript pourra être tout ce qui est en dehors du contexte du langage et de sa syntaxe.

Éviter les effets de bords permet de grandement garantir un code prédictible et maintenable, mais cela demande une certaine rigueur et un changement de mentalité conséquent par rapport à un style impératif de code.

Il n'est pas question de supprimer tous les effets de bords ! Il est simplement question de les minimiser, afin de minimiser les erreurs.

```
// Effet de bord
fetch("https://esgi.fr");

// Effet de bord
console.log("Hello, world!");

// Pas d'effet de bord
const age = 42;

// Effet de bord
const today = new Date();

// Pas d'effet de bord
Math.abs(12.15);

// Effet de bord
Math.random();
```

Séance 4 : PF — Fonctions pures

Le concept de fonction pure rejoint très étroitement celui des effets de bords, puisqu'on peut résumer une fonction pure à une fonction qui ne génère pas d'effets de bords.

Généralement, les fonctions ne renvoyant rien, mais qui effectuent une certaine action peuvent être considérées comme des fonctions impures (qui génèrent des effets de bords).

De la même façon que pour les effets de bords, avoir un maximum de fonction pures permet non seulement d'avoir un code plus prédictible et maintenable, mais cela a aussi un effet très positif sur la simplicité à mettre en place toute une série de tests unitaires, puisqu'avec des fonctions pures, cela devient trivial à concevoir.

```
function getArea(radius: number, pi: number): void {
  console.log(pi * radius ** 2);
}

getArea(5, 3.14159); // undefined

getArea(5, 1.14159); // undefined
```

Séance 4 : PF — Fonctions pures

Le concept de fonction pure rejoint très étroitement celui des effets de bords, puisqu'on peut résumer une fonction pure à une fonction qui ne génère pas d'effets de bords.

Généralement, les fonctions ne renvoyant rien, mais qui effectuent une certaine action peuvent être considérées comme des fonctions impures (qui génèrent des effets de bords).

De la même façon que pour les effets de bords, avoir un maximum de fonction pures permet non seulement d'avoir un code plus prédictible et maintenable, mais cela a aussi un effet très positif sur la simplicité à mettre en place toute une série de tests unitaires, puisqu'avec des fonctions pures, cela devient trivial à concevoir.

```
function getArea(radius: number, pi: number): number {
  return pi * radius ** 2;
}

const result = getArea(5, 3.14159);

console.log(result);

fetch("https://api.math.com", {
  method: "POST", body: result.toString()
});

navigator.clipboard.writeText(result.toString());
```

Séance 4 : PF — Fonctions d’ordre supérieur

Une fonction d’ordre supérieur est une fonction qui accepte un ou plusieurs arguments sous la forme de fonctions.

En effet, dans la plupart des langages de programmation, les fonctions sont aussi des valeurs, au même titre qu’une chaîne de caractères ou de nombre par exemple.

Mais une fonction d’ordre supérieur peut aussi être une fonction qui renvoie elle-même une fonction. C’est particulièrement pratique si l’on souhaite factoriser du code.

C’est ce qu’on appelle la composition de fonction.

```
function add(first: number, second: number): number {
  return first + second;
}

function addOne(value: number): number {
  return value + 1;
}

function addTwo(value: number): number {
  return value + 2;
}

function addThree(value: number): number {
  return value + 3;
}

add(1, 2); // 3
addOne(2); // 3
addTwo(1); // 3
addThree(0); // 3
```

Séance 4 : PF — Fonctions d’ordre supérieur

Une fonction d’ordre supérieur est une fonction qui accepte un ou plusieurs arguments sous la forme de fonctions.

En effet, dans la plupart des langages de programmation, les fonctions sont aussi des valeurs, au même titre qu’une chaîne de caractères ou de nombre par exemple.

Mais une fonction d’ordre supérieur peut aussi être une fonction qui renvoie elle-même une fonction. C’est particulièrement pratique si l’on souhaite factoriser du code.

C’est ce qu’on appelle la composition de fonction.

```
function add(first: number) {
  return function(second: number) {
    return first + second;
  }
}

const addOne = add(1);
const addTwo = add(2);
const addThree = add(3);

add(1)(2); // 3
addOne(2); // 3
addTwo(1); // 3
addThree(0); // 3
```

Séance 4 : PF — Fonctions d’ordre supérieur

Une fonction d’ordre supérieur est une fonction qui accepte un ou plusieurs arguments sous la forme de fonctions.

En effet, dans la plupart des langages de programmation, les fonctions sont aussi des valeurs, au même titre qu’une chaîne de caractères ou de nombre par exemple.

Mais une fonction d’ordre supérieur peut aussi être une fonction qui renvoie elle-même une fonction. C’est particulièrement pratique si l’on souhaite factoriser du code.

C’est ce qu’on appelle la composition de fonction.

```
const values = [1, 2, 3, 4, 5];

const evenValues = values.filter(value => {
  return value % 2 === 0;
});

const oddValues = values.filter(value => {
  return value % 2 !== 0;
});
```

Séance 4 : PF — Récursion

La récursion est une manière de parcourir un ensemble de données, à la façon d'une boucle par exemple.

Le principal souci d'une boucle, spécialement les boucles for (hormis for…of et for…in) est qu'elles reposent sur des variables, choses qu'il est préférable d'éviter en programmation fonctionnelle.

Il est tout à fait possible de «boucler» sur un tableau sans utiliser de boucle for, bien que les boucles for…of et for…in ne génèrent pas d'effets de bords (lorsque l'on utilise des constantes).

Une fonction récursive est une fonction qui s'appelle elle-même dans sa propre définition de fonction. Elle dispose d'un «cas terminal» (ou cas de base) qui permet d'éviter les tour de boucle infini et qui sonne la fin de l'itération.

```ts
type Update<Value, NewValue> = (value: Value) => NewValue

function mapWithSideEffect<Value, NewValue>(values: Value[], update: Update<Value, NewValue>) {
  let updatedValues: NewValue[] = [];

  for (let index = 0; index < values.length; index++) {
    let value = values[index];

    updatedValues.push(update(value));
  }

  return updatedValues;
}

function map<Value, NewValue>(values: Value[], update: Update<Value, NewValue>) {
  if (values.length === 0) {
    return [];
  }

  const [value, ...remainingValues] = values;

  return [
    update(value),
    ...map(remainingValues, update)
  ];
}
```

Séance 4 : PF — Result vs Try/Catch

La gestion des erreurs est un élément essentiel dans toute application.

Cependant, dans les langages de programmation orientés objets, la gestion des erreurs se fait souvent via l'utilisation de levée d'exception.

C'est un concept qui n'a pas sa place en programmation fonctionnelle, car une exception levée est un effet de bord direct : une fonction ne déclare pas explicitement les erreurs levées, on ne sait pas à quel endroit est-ce qu'elle seront levées, on ne sait pas si elles ont déjà été gérées auparavant, …

Afin de contrecarer cet inconvénient majeur, il est possible, plutôt que de lever des exceptions, de renvoyer à la place une valeur spéciale.

Cette valeur spéciale pourra prendre deux formes : une forme de succès, et une forme d'echec. C'est ce qu'on appelle communément en programmation fonctionnelle un Result, mais le terme Either est aussi très utilisé dans d'autres langages comme Haskell.

```
function divide(numerator: number, denominator: number): number {
  if (!Number.isFinite(numerator) || !Number.isFinite(denominator)) {
    throw new Error("Numerator and/or denominator is not finite");
  }

  if (Number.isNaN(numerator) || Number.isNaN(denominator)) {
    throw new Error("Numerator and/or denominator should be a number");
  }

  if (denominator === 0) {
    throw new Error("Denominator should not be zero");
  }

  return numerator / denominator;
}

const firstResult = divide(1, 2); // 0.5
const secondResult = divide(firstResult, 0); // ???
const thirdResult = divide(secondResult, 2); // ???
```

Séance 4 : PF — Result vs Try/Catch

La gestion des erreurs est un élément essentiel dans toute application.

Cependant, dans les langages de programmation orientés objets, la gestion des erreurs se fait souvent via l'utilisation de levée d'exception.

C'est un concept qui n'a pas sa place en programmation fonctionnelle, car une exception levée est un effet de bord direct : une fonction ne déclare pas explicitement les erreurs levées, on ne sait pas à quel endroit est-ce qu'elles seront levées, on ne sait pas si elles ont déjà été gérées auparavant, …

Afin de contrecarrer cet inconvénient majeur, il est possible, plutôt que de lever des exceptions, de renvoyer à la place une valeur spéciale.

Cette valeur spéciale pourra prendre deux formes : une forme de succès, et une forme d'échec. C'est ce qu'on appelle communément en programmation fonctionnelle un Result, mais le terme Either est aussi très utilisé dans d'autres langages comme Haskell.

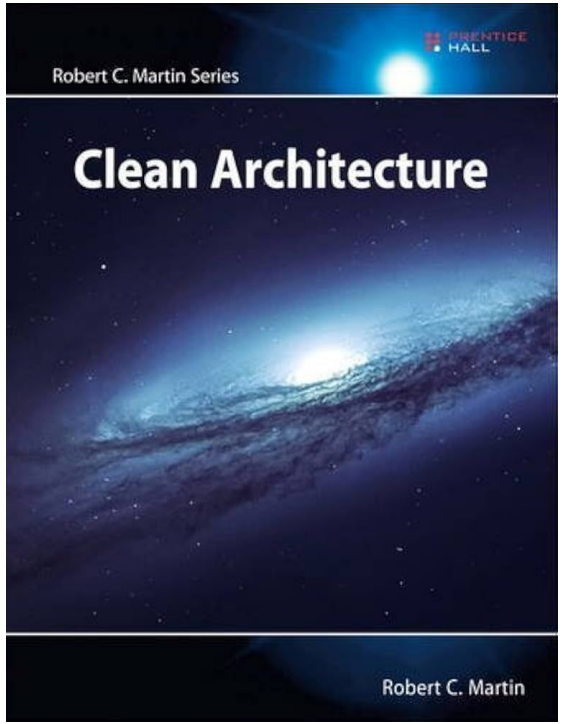
```
function divide(numerator: number, denominator: number): Result<number, "INFINITE" | "NAN" | "ZERODIV"> {
  if (!Number.isFinite(numerator) || !Number.isFinite(denominator)) {
    return new Failure("INFINITE");
  }

  if (Number.isNaN(numerator) || Number.isNaN(denominator)) {
    return new Failure("NAN");
  }

  if (denominator === 0) {
    return new Failure("ZERODIV");
  }

  return new Success(numerator / denominator);
}
```

Séance 4 : Exercice

- Implémenter la méthode Array.prototype.map sous la forme d'une fonction acceptant deux arguments (un tableau et une fonction) en utilisant de la récursion et sans mutation
- Implémenter la méthode Array.prototype.filter sous la forme d'une fonction acceptant deux arguments (un tableau et une fonction) en utilisant de la récursion et sans mutation
- Implémenter la méthode Array.prototype.reduce sous la forme d'une fonction acceptant trois arguments (un tableau, une valeur initiale et une fonction) en utilisant de la récursion et sans mutation

Séance 5 : Clean Architecture

- Ubiquitous Language
- Domain, Application & Infrastructure
- Entities
- Value Objects
- Repositories
- Use cases
- Services



Séance 5 : CA — Ubiquitous Language

Le Ubiquitous Language (ou langue universelle) est un concept permettant au code écrit d'être le plus proche possible de notre domaine métier.

La Clean Architecture permet une claire séparation entre le domaine métier d'un client et le domaine technique lié aux développeurs d'une application à destination d'un client.

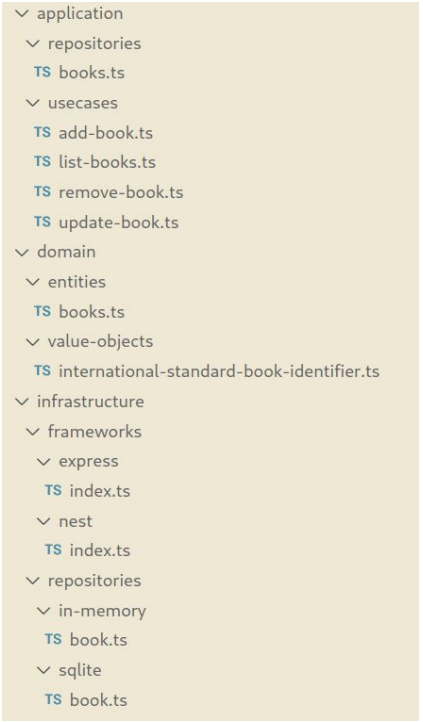
Plus le code est proche du domaine métier, plus il est simple à retravailler par la suite, permettant même au client de savoir de quoi l'on parle, même s'il ne sait pas coder puisque les termes du métier viennent directement faire parti de notre code-source.

```
export class Book {
  public constructor(
    public isbn: string,
    public title: string,
    public publicationDate: Date,
    public language: string,
    public availableCopies: number
  ) { }
}
```

Séance 5 : Domain, Application & Infrastructure

Le but d'organiser son code en séparant le domaine métier (domain), les cas d'utilisation de notre application (application), et la partie technique liée aux librairies & frameworks mais aussi l'implémentation concrète des interfaces métier (infrastructure) nous permet de simplifier largement le travail de maintenance plus tard, moyennant un surcoût lors du développement initial.

Avec une telle séparation, il est désormais possible de passer d'un framework à un autre, d'une version à une autre, de tester de nouvelles technologies, sans risquer de devoir mettre en pause tout le développement et l'implémentation des fonctionnalités métier chères à nos clients.



Séance 5 : Entities

Une entité (entity) est un élément central du métier d'un client, qui dispose de propriétés métier, de méthode métiers ainsi que d'un identifiant unique.

C'est une association directe avec le domaine métier d'un client et le code qui s'y trouve doit respecter les termes techniques liés au domaine métier associé.

Par exemple pour une société de livraison, une commande pourrait représenter une entité métier, de même que pour un colis, ou une facture de livraison.

```
export class Book {
  public constructor(
    public isbn: string,
    public title: string,
    public publicationDate: Date,
    public language: string,
    public availableCopies: number
  ) { }

  public borrow() {
    if (this.availableCopies === 0) {
      throw new Error("Not enough copies available.");
    }

    this.availableCopies--;
  }

  public returnBook() {
    this.availableCopies++;
  }
}
```

Séance 5 : Value Object

Un Value Object est une classe qui permet de représenter une valeur qui peut être validée à l'aide de règle métier.

C'est souvent un objet qui ne dispose pas d'identifiant unique, contrairement à une entité et c'est la principale différence entre les deux.

En dehors de cette différence, un Value Object, tout comme une entité, pourra contenir des règles métier tout comme des propriétés.

C'est souvent utile pour représenter des objets métiers, comme une devise financière, un email devant respecter un certain format, ou un identifiant de livre unique.

```
export class InternationalStandardBookIdentifier {
  public readonly value: string;

  public constructor(value: string) {
    if (!this.validate(value)) {
      throw new Error('Invalid ISBN format');
    }

    this.value = value;
  }

  public validate(value: string): boolean {
    const isbn10Regex = /^[0-9]{9}[0-9X]$/;
    const isbn13Regex = /^[0-9]{13}$/;

    return isbn10Regex.test(value) || isbn13Regex.test(value);
  }

  public equals(other: InternationalStandardBookIdentifier): boolean {
    return this.value === other.value;
  }
}
```

Séance 5 : Port et Adapter

Un port est une interface vers un système externe. Cela peut permettre de représenter par exemple un système de stockage (base de données) ou un service externe (API HTTP).

Un port est organisé bien souvent dans le dossier application, puisqu'il représente une manière d'interagir avec l'extérieur, mais n'induit aucune implémentation concrète afin de garder le code du domaine métier et de l'application le plus pure possible, facilitant les tests.

Du côté de l'infrastructure, cela se représente bien entendu sous la forme d'une implémentation concrète qui viendra respecter l'interface du domaine métier. Cela peut être par exemple une base de données SQLite ou une base de données MongoDB selon les besoins techniques.

```
import { Book } from "@/domain/entities/books";
import { InternationalStandardBookIdentifier } from "@/domain/value-objects/...";

export interface BookRepositoryInterface {
  add(book: Book): Promise<void>;
  remove(isbn: InternationalStandardBookIdentifier): Promise<void>;
  update(isbn: InternationalStandardBookIdentifier, book: Partial<Book>): Promise<void>;
  getAll(): Promise<Array<Book>>;
  getOne(isbn: InternationalStandardBookIdentifier): Promise<Book | null>;
}
```

Séance 5 : Port et Adapter

Un port est une interface vers un système externe. Cela peut permettre de représenter par exemple un système de stockage (base de données) ou un service externe (API HTTP).

Un port est organisé bien souvent dans le dossier application, puisqu'il représente une manière d'interagir avec l'extérieur, mais n'induit aucune implémentation concrète afin de garder le code du domaine métier et de l'application le plus pure possible, facilitant les tests.

Du côté de l'infrastructure, cela se représente bien entendu sous la forme d'une implémentation concrète qui viendra respecter l'interface du domaine métier. Cela peut être par exemple une base de données SQLite ou une base de données MongoDB selon les besoins techniques.

```ts
import { BookRepositoryInterface } from "@/application/repositories/books";
import { Book } from "@/domain/entities/books";
import { InternationalStandardBookIdentifier } from "@/domain/value-objects/...";

export class InMemoryBookRepository implements BookRepositoryInterface {
  public constructor(private readonly books: Array<Book>) {}

  public async add(book: Book): Promise<void> {
    this.books.push(book);
  }

  public async remove(isbn: InternationalStandardBookIdentifier): Promise<void> {
    const foundBookIndex = this.books.findIndex(book => {
      return isbn.equals(book.isbn);
    });

    if (foundBookIndex) {
      this.books.splice(foundBookIndex, 1);
    }
  }

  public async update(isbn: InternationalStandardBookIdentifier, book: Book): Promise<void> {
    //...
  }

  public async getAll(): Promise<Book[]> {
    return this.books;
  }

  public async getOne(isbn: InternationalStandardBookIdentifier): Promise<Book> {
    // ...
  }
}
```

Séance 5 : Use Case

Un Use Case est un cas d'utilisation qui provient de l'exploitation de l'activité métier de notre client.

C'est le point d'entrée vers toutes les fonctionnalités de notre métier, et c'est par ici que tout commence lorsque l'on souhaite implémenter concrètement le métier d'un client au sein d'un framework technique.

C'est généralement une classe contenant une méthode execute, bien que cela n'existe que par convention, il est tout à fait possible aussi de ne pas utiliser de classe et de n'avoir que des fonction acceptant des arguments.

```
import { Book } from "../../domain/entities/books";
import { BookRepositoryInterface } from "../repositories/books";

export class ListBooksUsecase {
  public constructor(private readonly bookRepository: BookRepositoryInterface) { }

  public async execute(): Promise<Array<Book>> {
    const books = await this.bookRepository.getAll();

    if (books.length === 0) {
      throw new Error("No books available in store.");
    }

    return books;
  }
}
```

Séance 5 : Exercice 1

- Créer une application Node.js permettant de lister l'ensemble des livres d'une bibliothèque, un livre dispose des attributs suivants
  - Titre
  - Date de parution
  - Nom de l'auteur
  - Quantité en stock
- L'application doit également pouvoir permettre d'ajouter, supprimer ou modifier un livre
- Ne pas faire la partie front-end, tester les routes depuis Postman, Thunder Client, HTTP, …
- Laisser la possibilité de choisir l'implémentation pour le stockage d'un livre en utilisant une implémentation en mémoire-vive
- Ajouter la possibilité d'envoyer un mail sur l'adresse d'un administrateur de l'application lorsqu'un livre est ajouté, modifié ou supprimé en utilisant l'implémentation en mémoire-vive
- Une fois l'application testée, utiliser Drizzle avec SQLite pour l'implémentation du système de stockage
- Utiliser resend.com pour l'implémentation du système d'envoi de mail

Séance 5 : Exercice 2

- Créer une API HTTP qui permet de faire une réservation d'un vol aérien
- Ajouter la possibilité de créer un vol, qui est représenté par un numéro de vol (ex: AF1452), un nombre de sièges, et un nom (Paris - Londre)
    - Pour réserver un vol, le client doit fournir un nombre de siège, et un numéro de vol
    - Vérifier que le nombre de siège soit suffisant, et dans le cas contraire, envoyer une erreur
- Ajouter la possibilité de lister les toutes les réservations
- Ajouter la possibilité de pouvoir confirmer une réservation
    -