UTC 503 - Cours

00 - Pour bien démarrer

Pour tester les codes ci-dessous on peut utiliser la console de son navigateur en appuyant sur F12.

1. ***Soit le code suivant en Javascript :***

```
var v = 5;
function f(v, x){
v = v + 1;
return v + x;
}
var z = f(10, 2);
```

Que valent v et z ?

- V = 5
- Z = 13

Attention au scope de la fonction, la valeur de v dans f() n'est valable qu'à l'intérieur de celle-ci.

2. ***Soit le code suivant en Javascript***

```
function f(v){
    if (v == 0){
        return 0;
    }
return v + f(v - 1);
}

var r = f(4);
```

Que vaut r ?

Fonction récursive, f(v) s'appelle elle même avec v == 0 comme condition de sortie. En passant comme paramètre 4. La fonction effectue le calcul : 4+3+2+1 = 10

- r = 10

3. ***Soit le code suivant en Javascript :***

```
function f(){
    return 4;
}

var x = f;
var y = f();
var z = x() + y;
```

Que valent x, y et z ?

- x = f

  x est une **fonction** avec la même définition que f

- y = 4

  Appel de la fonction f() qui retourne 4

- z = 8,

  Appel de la fonction f() contenue dans x (voir premier point) qui retourne 4, puis ajoute la valeur de y vue au deuxième point.

4. ***Lors d'un appel de fonction, les paramètres sont copiés sur quelle mémoire ?***
   - La pile (Stack)
   - Le Tas (Heap)
5. ***Lors d'une allocation dynamique de mémoire, quel type de mémoire est utilisée ?***
   - La pile (Stack)
   - Le Tas (Heap)

Difference Between Stack and Heap (with Comparison Chart) - Tech Differences

Stack and Heap

6. ***Quel concept POO est utilisé en combinant des méthodes et des attributs dans une classe ?***
    1. Polymorphisme
    2. Encapsulation
    3. Abstraction
    4. Héritage
7. ***Quand on parle de membre de classe, on parle de :***
    1. Uniquement de méthode
    2. Uniquement d'attribut (propriété)
    3. Méthode ou attribut
8. ***Quelle garantie nous apporte le principe d'encapsulation ?***
    1. Il nous garantit uniquement la validité des types des données de nos objets
    2. Il nous garantit la validité des types et des valeurs des données de nos objets
    3. Il nous garantit que l'utilisateur pourra manipuler des objets

       Note Webconf : Différence entre la valeur et le type.
9. ***Si l'on respecte le principe d'encapsulation, comment procédons-nous pour accéder et modifier la valeur des attributs de nos objets ?***
    1. Nous nous servons d'accesseurs et de mutateurs

       Note Webconf : Voir Getter et Setter
    2. Nous créons une classe à part chargée de récupérer et modifier ces valeurs
    3. Nous accédons par référence à ces attributs afin de récupérer leurs valeurs et de les modifier depuis l'extérieur de la classe
10. ***Quelle est l'utilité d'un constructeur ?***
    1. Déclarer l'ensemble des caractéristiques et fonctionnalités de l'objet afin de nous en servir
    2. Allouer la mémoire nécessaire pour que l'objet créé puisse exister
    3. Effectuer des opérations d'initialisation dès la création de l'objet
11. ***Quand on parle de méthode d'instance, on parle de :***
    1. Une méthode n'agissant que sur un seul objet (instance de la classe) à la fois.
    2. Une méthode indépendante de toute instance de la classe.
12. ***Quelle est la différence entre une classe et un objet ?***
    1. Une classe est une description comportant un ensemble de caractéristiques et de fonctionnalités dont chaque objet créé à partir de ce modèle héritera.

       Note Webconf : Analogie, moule a gaufre → gaufre. Le moule est la classe et l'instance serait la gaufre.
    2. Un objet est une description comportant un ensemble de caractéristiques et de fonctionnalités dont chaque classe créée à partir de ce modèle héritera
13. ***Quelle est la différence entre un objet et une instance de classe ?***

    Aucune, ce sont des synonymes.
14. ***En Java, C#, C++, PHP, ECMAScript 6, quel mot-clé permet de créer une nouvelle instance ?***
    1. instanceof
    2. new
    3. create

15. ***En Java, C#, C++, PHP, quels mots-clés sont utilisés pour spécifier la visibilité des propriétés et des méthodes ?***
    1. final
    2. private
    3. abstract
    4. protected

       un élément protected (protégé) est accessible uniquement aux classes d'un package et à ses classes filles. (private : accessible uniquement de la classe elle même).
    5. public
16. ***Quelle visibilité est la moins permissive ?***
    1. private
    2. protected
    3. public
17. ***Quel est le principe de l'héritage ?***
    1. Créer une classe possédant une partie des caractéristiques et fonctionnalités d'une autre classe
    2. Créer une classe capable de partager ses fonctionnalités avec une autre classe, et inversement
    3. Créer une classe possédant au moins toutes les caractéristiques et fonctionnalités d'une autre classe
18. ***Quelle condition nécessaire et suffisante doit-on avoir pour dire qu'une classe B doit hériter d'une classe A ?***
    1. On doit pouvoir dire : « B est un A »
    2. On doit retrouver des fonctionnalités en commun entre la classe B et la classe A
    3. On doit retrouver des caractéristiques en commun entre la classe B et la classe A
19. ***En Java, PHP, ECMAScript 6, quel mot-clé permet de procéder à un héritage ?***
    1. herits
    2. extends
    3. childof
20. ***En quoi consiste la redéfinition d'une méthode ?***
    1. Réécrire une méthode déjà implémentée dans la classe mère
    2. Déclarer deux fois la même méthode au sein d'une même classe
    3. Empêcher l'héritage d'une méthode au sein d'une classe fille
21. ***Quelle est la différence entre redéfinition de méthode et surcharge d'une méthode ? ***

    Aucune, ce sont des synonymes.

    Voir override
22. ***Qu'est-ce qu'une classe abstraite ?***
    1. C'est une classe déclarée avec le mot-clé abstract, ne pouvant pas être instanciée et pouvant contenir des méthodes abstraites, donc sans corps de méthode.
    2. C'est une classe ne pouvant pas être héritée.
    3. C'est une classe ne pouvant pas avoir de constructeur.
    4. C'est une classe du langage Java permettant de créer plusieurs objets en son sein.

```
abstract class Food {
    abstract showCalories() : void;
}
```

```
class Tomatoes extends Food {
    showCalories(): void {
    console.log(16)
    }
}

class Lemon extends Food {
    showCalories(): void {
    console.log(28)
    }
}
```

23. ***Dans quel cas décide-t-on de déclarer une classe abstraite ?***

    1. Lorsque l'on souhaite empêcher toute classe d'hériter de la dite classe
    2. Lorsque l'on souhaite représenter une nature en commun pour plusieurs classes
    3. Lorsque l'on souhaite imposer aux classes enfant de réécrire chacune des méthodes de la classe mère

    Une classe abstraite n'est pas nécessairement une interface bien qu'une interface soit une classe abstraite

24. ***Dans le langage Java, qu'est-ce qu'une interface ?***

    1. Une fenêtre permettant à l'utilisateur d'interagir avec le programme
    2. Une classe présente dans un package du langage Java servant de base à tous les objets du langage.
    3. Une classe abstraite
    4. Une classe 100% abstraite permettant de créer un nouveau super type et jouir du polymorphisme

```
public interface Ingestable {  // déclaration de l'interface
    public void whatHapens();
}

abstract class Food {  // déclaration de la classe abstraite
    abstract void showCalories();
}

public class Tomatoes extends Food implements Ingestable { // A noter
l'héritage de la classe abstraite et le contrat d'interface
    void showCalories(); {
        System.out.println("16") // On redéfinie la fonction
    }

    void whatHappens() {
    System.out.println("miam"); // On implante la fonction (obligatoire par
"contrat")
    }
}

class Lemon extends Food implements ingestable {
    void showCalories() {
        System.out.println("28");
    }

    void whatHappens() {
```

```
        System.out.println("Frisson dans le dos...")
    }
}

class Poison implements Ingestable { // on implante l'interface mais PAS la classe abstraite
    void whatHappens() {
        System.out.println("Oups ...")
    }
}
```

JAVA (Intermédiaire) - 34 - Differences entre classes abstraites et Interfaces

Une interface n'est pas une classe ! Toute classe qui implémente une interface doit obligatoirement redéfinir la ou les méthodes de cette dernière, c'est un "contrat". A la différence une classe abstraite peut contenir une définition de base et une classe qui hérite peut la redéfinir pour la spécialiser (override).

25. ***Une classe peut être héritée par combien de classes filles ?***
    1. Qu'une seule
    2. Ca dépend des langages
    3. une infinité
26. ***De combien de classes une classe fille peut-elle hériter ?***
    1. Qu'une seule
    2. Ca dépend des langages

    Python par ex permet l'héritage multiple, mais d'autre langage ne le permette pas (java par ex).

    1. une infinité

---

02 - Programmation impérative

Éléments de base

En programmation impérative on a :

- **La séquence d'instruction**
  - La machine va exécuter les instructions les unes après les autres dans l'ordre de lecture
  - Une instruction par ligne et/ou séparée par un `;` selon le langage
  - Exemple en C :

    ```
    nb_oeuf =4; verse_dans-bol(nb_oeuf);active_fouet(5);chauffe(300);
    ```

- **L'assignation ou affectation**
  - identifiant ← expression
  - Stocke dans la zone mémoire "pointée" par *identifiant* le résultat de l'exécution (évaluation) de *expression*.
  - Exemple en C:

```
x = 2;
rendement = valeur_courante * 100 / valeur_initiale;
solde_courant = lit_solde(id_compte);
```

- **L'instruction conditionnelle**
  - *si condition est vraie alors exécute {instructions}*
  - *condition* est une expression booléenne : vraie ou fausse, 0 ou 1.
  - Exemple en C:

```
if (jauge < 10) alert('Réserve');
if (meteo == PLUIE){prendreParapluie(); prendreManteau();}
if (coursUTC503Recu(){meMettreACoder()};)
```

  - Si condition est vraie alors exécute {instructions A} sinon exécute {instructions B}
  - Exemple en C:

```
if (age < 18) {
    AffichetarufReduit();
}
else {
afficheTarifPlein();
}
```

    Attention : En C, ne confondez pas le '=' de affectation avec le '==' qui teste l'égalité. D'autant plus que pour le compilateur, le code *if (c=3)* a une signification : J'affecte 3 à c et comme c est différent de 0 la condition est réalisée. Alors que vous vouliez plutôt tester que c vaut 3 !

- **La boucle**
  - Tant que condition est vraie exécute {instructions}
  - La condition doit évoluer lors des itérations de manière à pouvoir sortir de la boucle. Sinon, on crée une "boucle infini"
  - Si la condition est fausse dès sa première évaluation, on exécute jamais les instructions.
  - Exemple en C:

```
while (credit > 0) {
    int gain = nouveauTirage();
    credit = credit - 1 + gain;
}
// comment gagner sans jamais pouvoir partir avec ses gains :D
```

  - Répète N fois {instructions}
  - La boucle "for" permet d'exécuter un nombre prédéfini d'itérations exécutant les instructions.
  - Elle se définit par quatre blocs : initialisation, condition de sortie de boucle, code évolution boucle, instructions à exécuter à chaque boucle.
  - Exemple en C :

```
for (int compteur = 0; compteur < age; compteur = compteur +1) {
    poseUneBougieSurLeGateau();
    }
```

- **Instructions de branchements sans condition**
  - Permet à la séquence d'instructions d'être déroutée vers un autre endroit du programme
  - Deux manières :
    - **goto**, "go to label", "allez à cette étiquette"
      - Souvent considérée comme l'instruction à éviter, car rendant le code difficile à comprendre et source de bug, elle peut être salutaire dans des cas d'exceptions :

        Goto (informatique)
      - Exemple en C:

```
connexion = connectionAuServeur()
if (connexion == ERREUR) goto erreur_fatale
recupererDonnes(connexion);
goto fin
erreur_fatale:
printf(" Echec de connexion au serveur ");
fin:
printf(" Fin du programme ");
```

    - **appel de fonction, exécute fonction**
      - C'est un "goto", saut, avec passage de paramètres, nouveau contexte de variables et surtout avec un saut de retour vers l'instruction suivant l'appel.
      - La fonction répond aussi au besoin de factorisation, i.e. éviter de dupliquer du code identique
      - Exemple en C:

```
void melange(int vitesse, int duree){
    actionneMoteur(vitesse);
    attend(duree);
    arreteMoteur();
}
void recetteCrepe(){
    ajoutOeufs();
    ajouteSucre();
    melange(10, 5);
    ajouteFarine();
    melange(5, 10);
    ajouteLait();
    melange(5, 30);
```

---

03 - Programmation orientée Objet

Principe du paradigme

- S’appuie sur des « objets »
- Un objet est une entité, du monde physique par exemple : animal, voiture, personne, service
- Un objet a une structure interne et un comportement
- Un objet interagit avec les autres objets par des messages

La POO c’est donc :

- Modéliser ces objets
  - Définir ces propriétés (ou attributs)
  - Définir ces relations avec les autres objets (composition, héritage, délégation, etc ...)
  - Définir son interface, i.e. l'ensemble des messages qu'on pourra lui appliquer
- La modélisation pourra se faire via UML par exemple

La POO c’est donc, encore plus concrètement :

- Définir les "types " de ces objets
- Ces types sont appelées "classes"
- Une classe définit
  - Attributs : données de la structure interne de l'objet, son état
  - Méthodes : une méthode étant une fonction définissant le comportement à la réception d'un message
- La classe peut être vue comme le moule permettant la création d’un objet
- D'une même classe, sont générés plusieurs (instances d') objets ayant les memes attributs, mais potentiellement pas le même état, ie pas les mêmes valeurs d'attributs.

Exemple en TypeScript : zoo-v1.ts

```
class ZooAnimal {

    // Attributs
    nom: string;
    age: number;
    poids: number;
    quantiteNourritureTotale: number;
    bienNourri: boolean;

    // Constructeur
    constructor(nom: string, age: number, poids: number) {

        this.nom = nom;
        this.age = age;
        this.poids = poids;
        this.quantiteNourritureTotale = 0;
        this.bienNourri = false;

    }
    // Méthodes
```

```
    nouvelleJournee() {

        this.quantiteNourritureTotale = 0;
        this.bienNourri = false;

    }
    recoitNourriture(quantite: number) {

        this.quantiteNourritureTotale += quantite;
        if (this.quantiteNourritureTotale > (this.poids / 5)) {
            this.bienNourri = true;
        } else {
            this.bienNourri = false;
        }

    }

    etatAlimentation() {

        let etat = this.nom + ", " + this.age + " ans, a reçu " +
            this.quantiteNourritureTotale + "Kg de nourriture."
        if (this.bienNourri) {
            etat += " " + this.nom + " est bien nourri.";
        }
        return etat;

    }
}

// Création d'instances de classes: objets

let flipper = new ZooAnimal("Flipper", 30, 150);
let pandi = new ZooAnimal("Pandi", 10, 80);

// Appels de méthodes sur les objets
flipper.nouvelleJournee();
pandi.nouvelleJournee();
flipper.recoitNourriture(10);
pandi.recoitNourriture(10);
flipper.recoitNourriture(25);
alert(flipper.etatAlimentation());
alert(pandi.etatAlimentation());
```

Encapsulation

- Un objet s'appuie sur le principe d'encapsulation : regrouper des données et les fonctions qui les manipulent.
- Un objet doit être considéré comme une "boîte noire" dont l'état interne doit rester intègre, cohérent.
- Pour maintenir cette intégrité, l'objet ne doit être manipulé par les autres objets qu'au travers de son "interface", i.e. ses méthodes publiques.
- Dans notre exemple, il est actuellement possible de casser l'intégrité : insérez à la ligne 47 flipper.bienNourri = false; casse la cohérence entre le poids et l'attribut bienNourri.
- Pour garantir l'intégrité, on peut « masquer » certains attributs et méthodes en modifiant leur niveau d'accessibilité : « private » ou « public » par défaut

```
class ZooAnimal {

    // Attributs
    private nom: string;
    private age: number;
    private poids: number;
    private quantiteNourritureTotale: number;
    private bienNourri: boolean;

    // Constructeur
    constructor(nom: string, age: number, poids: number) {
        this.nom = nom;
        this.age = age;
        this.poids = poids;
        this.quantiteNourritureTotale = 0;
        this.bienNourri = false;
    }

    // Méthodes
    nouvelleJournee() {
        this.quantiteNourritureTotale = 0;
        this.bienNourri = false;
    }

    recoitNourriture(quantite: number) {
        this.quantiteNourritureTotale += quantite;
        if (this.quantiteNourritureTotale > (this.poids / 5)) {
            this.bienNourri = true;
        } else {
            this.bienNourri = false;
        }
    }

    etatAlimentation() {
        let etat = this.nom + ", " + this.age + " ans, a reçu " +
            this.quantiteNourritureTotale + "Kg de nourriture."

        if (this.bienNourri) {
            etat += " " + this.nom + " est bien nourri.";
        }

        return etat;
    }
}

// Création d'instances de classes: objets
let flipper = new ZooAnimal("Flipper", 30, 150);
let pandi2 = new ZooAnimal("Pandi", 10, 80);

// Appels de méthodes sur les objets
flipper.nouvelleJournee();
pandi.nouvelleJournee();
flipper.recoitNourriture(10);
pandi.recoitNourriture(10);
flipper.recoitNourriture(25);
alert(flipper.etatAlimentation());
alert(pandi.etatAlimentation());
```

Méthodes spéciales

- Constructeur : est appelé automatiquement lors de la création de l'objet pour initialiser ses attributs
- Exemple en TypeScript :

```
constructor(nom: string, age: number, poids: number)
```

- Destructeur / Finaliseur : est appelé à la fin de vie de l'objet pour libérer des ressources détenues par l'objet.
- Existe en C++ et Java
- N'existe pas en TypeScript
- Accesseurs
  - getters : permet de lire un attribut (virtuel ou pas) privé de l'objet
  - setters : permet de modifier un attribut privé de l'objet selon des contraintes d'intégrité.

Accesseurs : exemple modifiés à la façon Java : zoo-vts

```
getNom() {
return this.nom;
}
setNom(nouveauNom: string) {
if (nouveauNom == null || nouveauNom.trim().length == 0) {
console.log("Erreur: le nouveau nom n'est pas correct");
return;
}
this.nom = nouveauNom;
}
[...]
flipper.setNom("Flipper Junior");
```

Accesseurs : exemple modifié à la façon TypeScript: zoo-V4.ts

```
private _nom: string;
[...]
get nom() {
return this._nom;
}
set nom(nouveauNom: string) {
if (nouveauNom == null || nouveauNom.trim().length == 0) {
console.log("Erreur: le nouveau nom n'est pas correct");
return;
}
this._nom = nouveauNom;
}
[...]
flipper.nom = "Flipper Junior";
```

Héritage

L'héritage établit une relation de généralisation-spécialisation qui permet d'hériter dans la déclaration d'une nouvelle classe (appelée classe dérivée, classe fille, classe enfant ou sous-classe) des caractéristiques (propriétés et méthodes) de la déclaration d'une autre classe (appelée classe de base, classe mère, classe parent ou super-classe).

En déclarant une nouvelle classe B par héritage de la classe A, ajoutant de nouveaux membres, on peut alors dire que :

• A est une *généralisation* de B et B est une spécialisation de A ; • A est une *super-classe* de B et B est une sous-classe de A ; • A est la *classe mère* de B et B est une classe fille de A.

Lorsqu'une classe fille hérite d'une classe mère, elle peut alors utiliser les caractéristiques de la classe mère.

Héritage : des Dauphins et des Pandas

- Un Dauphin est un animal du Zoo
- Un Dauphin a un poisson préféré
- Un panda est un animal du Zoo
- Un Panda a une plante préféré

```
class ZooDauphin extends ZooAnimal {
    private poissonPrefere: string;
    constructor(name: string, age: number, poids: number, poissonPrefere:
string) {
        super(name, age, poids);
        this.poissonPrefere = poissonPrefere;
    }
}
class ZooPanda extends ZooAnimal {
    private plantePreferee: string;
}
constructor(name: string, age: number, poids: number, plantePreferee: string){
    super(name, age, poids);
    this.plantePreferee = plantePreferee;
}
let flipper = new ZooDauphin("Flipper", 30, 150, "hareng");
let oum = new ZooDauphin("Oum", 20, 100, "sardine");
let pandi = new ZooPanda("Pandi", 10, 80, "bambou");
```

Polymorphisme par héritage

- Consiste à redéfinir une méthode pour affiner son comportement
- Dans notre exemple, redéfinissons

  etatAlimentation

  pour affiner le résultat en fonction de la classe de l'animal:
  - Evoquer le poisson préféré pour ZooDauphin
  - Evoquer la plante préféré pour ZooPanda

```
//Polymorphisme par héritage : zoo-v6.ts

class ZooDauphin extends ZooAnimal {
```

```
    [...]
    etatAlimentation() {
        let etat = this.nom + ", " + this.age + " ans, a reçu " +
this.quantiteNourritureTotale + "Kg de nourriture. »
        if (this.bienNourri) {
            etat += " " + this.nom + " est bien nourri.";
        }
        etat += " Son poisson préféré est " + this.poissonPrefere + ".";
        return etat;
    }
    class ZooPanda extends ZooAnimal {
        [...]
        }
        etatAlimentation() {
        let etat = this.nom + ", " + this.age + " ans, a reçu " +
this.quantiteNourritureTotale + "Kg de nourriture. »
        if (this.bienNourri) {
        etat += " " + this.nom + " est bien nourri.";
        }
        etat += " Sa plante préférée est " + this.plantePreferee + ".";
        return etat;
    }
}
```

- Le but est atteint, mais il y a du code dupliqué
- Pour optimiser, appuyons-nous sur la méthode *etatAlimentation* de la classe mère pour réaliser une redéfinition partielle.
- Pour invoquer une définition de la classe mère, utilisons le mot clé *super.*
- super.etatAlimentation()

```
unAnimal of lesZclass ZooDauphin extends ZooAnimal {
    [...]
    etatAlimentation() {
        return super.etatAlimentation() + " Son poisson préféré est " +
this.poissonPrefere + ".";
    }
}
class ZooPanda extends ZooAnimal {
    [...]
}
etatAlimentation() {
    return super.etatAlimentation() + " Sa plante préférée est " +
this.plantePreferee + ".";
}
```

Classe Abstraite

- La règle concernant *bienNourri* devrait-être spécifique à chaque type d'animal.
- Par conséquent, définissons dans ZooAnimal une méthode *calculBienNourri* qui sera appelé par *recoitNourriture* et qui sera redéfinie par chaque classe fille.
- Ecrire un code "par défaut" dans la méthode *calculBienNourri* de *ZooAnimal,* c'est prendre le risque qu'elle ne soit pas redéfinie par la classe fille et donc d'avoir une valeur erronée.
- Comment forcer les classes filles à redéfinir la méthode *calculeBienNourri* ?
- En déclarant *calculeBienNourri* comme **méthode abstraite**.

- Par conséquent, *ZooAnimal* sera déclarée en **classe abstraite.**
- En plus, *ZooAnimal* maintenant abstraite, ne pourra plus être instanciée. Ce qui a du sens : seules ses classes filles non abstraites pourront l'être.

```
abstract class ZooAnimal {
    [...]
    abstract calculeBienNourri()
    recoitNourriture(quantite: number) {
        this.quantiteNourritureTotale += quantite;
        this.bienNourri = this.calculeBienNourri();
    }
}
class ZooDauphin {
    [...]
    calculeBienNourri() {
        return this.quantiteNourritureTotale > this.poids / 8;
    }
}
class ZooPanda {
    [...]
    calculeBienNourri() {
        return this.quantiteNourritureTotale > (this.poids / 4 - this.age / 10);
    }
}
```

« Ha Bravo !? Ça ne compile pas !?! »

Les méthodes *calculeBienNourri* des classes filles accèdent à des attributs de la classe mère, déclarés en *private,* Pour que les classes filles y accèdent, il faut l'autoriser en changeant *private* en *protected* pour ces attributs uniquement.

Tant qu'on est dans les "modifiers", *calculeBienNourri* sera aussi déclaré en *protected*, car ça ne concerne que le système interne de l'objet et non son interface (publique).

Upcasting, ou l'intêret du polymorphisme par héritage

- L'upcasting, c'est voir un objet selon l'interface de la classe mère, mais en appelant les méthodes redéfinies.
- Mettons les dauphins et pandas dans un tableau typé de ZooAnimal ; on vient d'upcaster.
- Pourtant, quand on prendra un élément du tableau et qu'on appellera la méthode *etatAlimentation,* c'est bien celle de ZooDauphin ou ZooPanda qui sera appelé et non celle de ZooAnimal.

```
let flipper = new ZooDauphin("Flipper", 30, 150, "hareng");
let oum = new ZooDauphin("Oum", 20, 100, "sardine");
let pandi = new ZooPanda("Pandi", 10, 80, "bambou");
let lesZAnimaux:ZooAnimal[] = [flipper, oum, pandi];
for(let unAnimal of lesZAnimaux){
unAnimal.nouvelleJournee();
}
flipper.recoitNourriture(10);
pandi.recoitNourriture(10);
flipper.recoitNourriture(25);
flipper.nom = "Flipper Junior";
oum.recoitNourriture(15);
// Remplacer console.log par alert pour un usage dans le playground
```

```
for(let unAnimal of lesZAnimaux){
console.log(unAnimal.etatAlimentation());
}
```

Un autre polymorphisme : ad hoc ou surcharge

- Le polymorphisme ad hoc consiste à surcharger une méthode.
- Il existe alors plusieurs méthodes de même nom renvoyant le même type de résultat, mais avec des arguments différents.
- Usages :
    - Prise en charge de conversion d'arguments
    - Gestion de valeurs par défaut, par exemple dans les constructeurs
        - *Pas vrai en TypeScript, car on peut définir des valeurs par défaut dans les arguments et on ne peut pas surcharger une méthode en TypeScript*

Le panda peut aussi recevoir sa nourriture par sac de différentes marques : RoyalPanda, Pamboo.

En Java, ça donnerait :

```
public void recoitNourriture(int quantiteSac, String marqueSac){
    int poidsSac = 0;
    switch(marqueSac){
        case « RoyalPanda » : poidsSac = 10; break;
        case « Pamboo » : poidsSac = 8; break;
        default: throw RuntimeException(« Marque inconnue »);
    }
    public void recoitNourriture(int quantiteSac, String marqueSac){
int poidsSac = 0;
switch(marqueSac){
case « RoyalPanda » : poidsSac = 10; break;
case « Pamboo » : poidsSac = 8; break;
default: throw RuntimeException(« Marque inconnue »);
}
this.recoitNourriture(quantiteSac * poidsSac);
}this.recoitNourriture(quantiteSac * poidsSac);
}
```

Le panda peut aussi recevoir sa nourriture par sac de différentes marques : RoyalPanda, Pamboo. En Java, ça donnerait :

```
public void recoitNourriture(int quantiteSac, String marqueSac){
    int poidsSac = 0;
    switch(marqueSac){
        case « RoyalPanda » : poidsSac = 10; break;
        case « Pamboo » : poidsSac = 8; break;
        default: throw RuntimeException(« Marque inconnue »);
}
this.recoitNourriture(quantiteSac * poidsSac);
}
```

---

04 - Générique

Plan

- Problème
- Solution générique
- Classe générique
- Usage
- Exercice

Problème

```
// Retourne x si non nul defaut sinon.
// x et défaut sont des nombres

function defautSiNul(x : number, defaut: number) : number {
    if (x == null)
    return defaut;
    else
    return x;
}

let maValeurParDefaut = 4;
let maValeur = 1;
console.log(defautSiNul(maValeur, maValeurParDefaut));
maValeur= null;
console.log(defautSiNul(maValeur, maValeurParDefaut));
```

- Fonctionne très bien pour des nombres
- Si on veut gérer aussi des string ?
- Une solution: créer une fonction par type ?

```
function defautSiNulNumber(x: number, defaut: number): number {
    if (x == null)
        return defaut;
    else
        return x;
}
function defautSiNulString(x: string, defaut: string): string {
    if (x == null)
        return defaut;
    else
        return x;
}
let maValeurParDefaut = "Quatre";
let maValeur = "Un";
console.log(defautSiNulString(maValeur, maValeurParDefaut));
maValeur = null;
console.log(defautSiNulString(maValeur, maValeurParDefaut));
```

- Les codes des 2 fonctions sont identiques
- Seuls les noms et les types diffèrent

- Dommage de dupliquer le code pour tous les types
- Et si on utilisait le type `any` ?

```
function defautSiNul(x: any, defaut: any): any {
    if (x == null)
        return defaut;
    else
        return x;
}
let maValeurParDefaut = 4;
let maValeur = 1;
let r: number = defautSiNul(maValeur, maValeurParDefaut);
console.log(r);
// On mélange les types string et number.
// Que va-t-on récupérer ?
let maValeur2 = "Deux";
r = defautSiNul(maValeur2, maValeurParDefaut);
console.log(r);
```

- Ca marche ! Enfin presque ...
- On perd la vérification de la cohérence entre x et défaut
- De même entre le type de retour et les types des paramètres
- Idéalement :
    - garder le principe à la "any" pour fonctionner avec plusieurs types connus qu'à l'appel
    - Mais avec une contrainte mettant la cohérence entre les types des arguments et celui de la fonction

La solution : la programmation générique !

```
function defautSiNul<T> (x: T, defaut: T): T {
    if (x == null)
        return defaut;
    else
        return x;
}

let nombreParDefaut = 4;
let nombre = 1;
let nombreRes: number = defautSiNul(nombre, nombreParDefaut);
console.log(nombreRes);
let chaineParDefaut = "Quatre"
let chaine = "Un"
let chaineRes: string = defautSiNul(chaine, chaineParDefaut);
console.log(chaineRes);
```

Comment lire la définition de la fonction ?

```
function defautSiNul<T> (x: T, defaut: T): T
```

"La fonction *`defautSiNul`* va fonctionner avec un type paramétré qu'on appellera `T` dont on ne sait rien à la définition de la fonction. Ce type inconnu T sera celui des paramètres `X` et `defaut` ainsi que celui de retour.

T sera défini, deviné, "inféré", pour chaque appel de la fonction grâce au type du 1er argument passé : si X est un `number` alors `defaut` sera aussi un `number` et la valeur retournée par la fonction sera aussi `number`."

Remarques

- On aurait pu choisir une autre lettre que T : X, Y, Z, etc ...
- Dans une fonction générique, tous les paramètres ne sont pas obligatoirement génériques
- Le type paramétré peut être intégré à la déclaration d'un tableau
- On peut utiliser plusieurs types paramétrés
- Exemples :

```
function warningSiTailleMax<E>(Array<E>, tailleMax: number) : bool
```

```
function f1<X, Y>(x: X, y : Y) : X
```

Même principe que pour la fonction

```
class NomClasse<T>
```

- Le type générique paramétré peut être utilisé pour définir le type des
  - attributs
  - paramètres de méthodes
  - résultat de méthodes

Exemple: classe-gen-v1.ts

- Soit un objet `CompteurAcces` qui contient un autre objet x de type string
- L'objet x est passé à la construction de l'objet de classe CompteurAcces
- La classe CompteurAcces a en interne un compteur et un attribut x
- Elle a aussi deux méthodes:
  - getX : retourne l'objet x et incrémente compteur
  - getCompteur: retourne la valeur du compteur

```
class CompteurAcces<T> {

    private compteur: number;
    private x: T;

    constructor(x: T) {
        this.x = x;
        this.compteur = 0;
    }

    getX(): T {
        this.compteur++
        return this.x;
    }

    getCompteur(): number {
        return this.compteur;
```

```
    }

}
```

Exemple d'exécution:

```
let test : CompteurAcces<number> = new CompteurAcces(23);

console.log(test.getCompteur());
console.log(test.getX());
console.log(test.getCompteur());

// [LOG]: 0
// [LOG]: 23
// [LOG]: 1
```

Contrainte générique

Limite du type générique paramétré simple (i.e. sans contrainte)

- Exemples :

  ```
  function maFonction<T>(v: T){}
  ```

  ```
  class maClasse<T> { private v: T;}
  ```

- Avec v on peut :
  - Tester sa nullité
  - L'affecter
  - La stocker dans une autre variable ou autre structure de donnée
- Avec V on ne peut pas :
  - Utiliser un de ses attributs. Ex: v.length
  - Appeler une de ses méthodes. Ex: v.add(5)
- Car on ne sait rien sur le type de v
- Il faudrait pouvoir en dire plus sur T sans être trop spécifique i.e. définir une **contrainte** sur T.

Exemple:

- En indiquant de quel classe (abstraite) ou interface hérite T
- Exemples:

```
<T extends MaClasse>
<X extends MaClasseAbstraite>
<Z extends MonInterface>
interface Mesurable{
    length: number;
}

class Ruban implements Mesurable {
    length: number;
    constructor(length: number){
```

```
        this.length = length;
    }
}

function selectionnePetit<T extends Mesurable>(element: T, limite: number) : T {
    return element.length < limite?element:null;
}

let ruban = new Ruban(10);
let chaine = "myLengthIs12";
let petitRuban = selectionnePetit(ruban, 20);
let petiteChaine = selectionnePetit(chaine, 20);

console.log(petitRuban);
console.log(petiteChaine);
Type 'null' is not assignable to type 'T'.
```

[TS2322: Type 'null' is not assignable to type 'string | void' · Issue #8322 · microsoft/TypeScript]

Quick background, if you are not using `--strictNullChecks` both `null` and `undefined` are in the domain of all types. so `T | undefined | null` is equivalent to `T`. if you are using `--strictNullChecks` however, that is not the case. `T` does not include `null` or `undefined`. About the meanings, `null` is the type of the js value `null`. at runtime it is an object (i.e. `typeof null === "object"`), and it means the sentinel object. `undefined` is the type of the js value `undefined` which means uninitialized. so an uninitialized variable has type `undefined` for instance. an optional parameter is implicitly known to include the `undefined` type, e.g. `x?: string` here `x` is of type `string|undefined`. `void` is the absence of a type, you could think of it as another name to undefined. the main use for `void` is to distinguish a function that is expected to returns nothing (void) from one that would return a value but possibly `undefined` (this way you can not assign `()=>void` to something like `()=>string|undefined` by mistake). So in short, only use `void` in return types of functions, to mean nothing is returned. use `undefined` to mean either uninitialized, or the js `undefined` value. use `null` to mean the js value `null`.

Implantation du type Nullable :

```
type Nullable<T> = T | undefined | null;

interface Mesurable{
    length : Nullable<number>;
}

class Ruban implements Mesurable {
    length: number;
    constructor(length: number){
        this.length = length;
    }
}

function selectionnePetit<T extends Mesurable>(element: T, limite: number) : T {
    return element.length < limite? element : null;
}

let ruban = new Ruban(10);
let chaine = "myLengthIs12";
let petitRuban = selectionnePetit(ruban, 20);
let petiteChaine = selectionnePetit(chaine, 5);
```

```
console.log(petitRuban);
console.log(petiteChaine);
```

On pourrait être tenté d'écrire...

```
function selectionnePetit(element: Mesurable, limite: number) : Mesurable {
    return element.length < limite? element : null;
}
```

- Pourquoi cette version de `selectionnePetit` n'est pas satisfaisante ?
- Parce que sa signature ne garantit pas que le type de retour et le type de `element` soient identiques.
- `element` pourrait être un `ruban` et la fonction pourrait, sur une erreur de codage, retourner un `string` et ça respecterait pourtant la signature de la fonction.
- Bien que `string` et `ruban` soient des `Mesurable`, ce ne sont pas les mêmes types.
- La version précédente de `selectionnePetit` garantit que le type de retour sera du même type que `element`.

Usage

On peut dire que la généricité

- Est une forme de polymorphisme, le **polymorphisme de type** dit aussi **paramétrage de type :** en effet, le type de donnée général (abstrait) apparaît comme un paramètre des algorithmes définis, avec la particularité que ce paramètre-là est un type.
- Consiste à définir des algorithmes identiques opérant sur des données de types différents.
- Un algorithme de tri d'objets est par exemple toujours le même, peu importe le le type de données.
- L'algorithme doit uniquement disposer de :
  - Un itérateur sur les éléments, générique
  - Une fonction de comparaison, générique
- Les algorithmes génériques sont ainsi très présents dans le domaine des collections.
- Exemple en typescript : Array

---

05 - Patron de conception

- Définition
- Les patrons du GoF
- Exercices

Un patron de conception ...

- Plus conny sous le de "Design Pattern"

- "Décrit une organisation de classes fréquemment utilisée pour résoudre un problème récurrent. Le patron de conception parle d'instances, de rôles et de collaboration"
- Issu de l'expérience des concepteurs
    - Voir le livre "Design Patterns" écrit par le Gang of Four (GoF)
- Le nom du patron sert de vocabulaire commun entre le concepteur et le programmeur
    - Le dialogue évite de longues explications et se limite alors à "j'ai un utilisé un SingleTon"

Concrètement, un patron est décrit par ...

- Un nom
- Une description du problème à résoudre
- Une description de la solution
- Une illustration, un exemple

Orthogonalité du patron

- "**Chaque patron doit correspondre à une approche différente**, qui ne répète pas les idées ou stratégies présentes dans d'autres patrons".
- "Cette qualité permet au concepteur d'analyser un problème et d'en résoudre chaque aspect d'une façon organisée, ainsi que de c**ombiner les patrons pour construire une solution**"

Un livre

- Patrons de Conceptions issus du livre "Design Patterns: Elements of reusable Software" paru en 1994
- Co-Ecrit par le "Gang of Four", GoF:
    - Erich Gamma, Richard Helm, Ralph Johnson (en) et John Vlissides

Un catalogue de 23 patrons en 3 familles

- **Créateurs :** ils définissent comment faire l'instanciation et la configuration des classes et des objets
- **Structuraux :** ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de implémentation)
- **Comportementaux:** ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués

Accessible sur Wikibook.org

- https://fr.wikibooks.org/wiki/Patrons_de_conception

Comment utiliser ce catalogue ?

- Lire cette page
- Les apprendre au fur et a mesure de la vie du développeur
- Décrire son besoin, puis "fouiller" dans ce catalogue
- Des très connu :
    - Itérateur
    - Observeur
    - Factory, Builder
    - Singleton

---

06 - Programmation fonctionnelle

En quelques mots

La programmation fonctionnelle...

- Paradigme de programmation de type déclaratif
- Le calcul est l'évaluation de fonctions mathématiques
- Une méthode de développement par composition en **fonction pure**
- Permet d'éviter les problèmes inhérents à :
    - La mutation des données (mutable data)
    - Les états partagés (shared state)
    - Les effets de bords (side-effects)

Histoire

- Années 1930, Alonzo Church travaille sur les concepts de fonction et d'application : le 'Lambda calcul"
- Avant la présentation de la "machine de Turing", base de la programmation impérative en 1936
- 1954, Fortran (FORmula TRANslator)
- 1958, Lisp: fonctionnel et impératif
- 1957, Scheme: pur fonctionnel
- 1987 Caml: multi paradigme
- 1990, haskel : pur fonctionnel

Apparition des langages fonctionnels

Choix du langage

- On aurait pu choisir Haskell, Lisp ou encore Scheme pour présenter les concepts de la programmation fonctionnelle
- Cependant, ces concepts ont intégré les langages impératifs et OO :
    - Cette façon de programmer est une tendance forte
    - La rigueur apportée permet d'éviter certains bugs
    - Aussi, il est intéressant de poursuivre en TypeScript

Pour la culture, un exemple en Haskell

```
fact :: int -> Int
fact 0 = 1
fact n = n * fact (n -1)
main = do
    putStrLn "The factorial of 5 is:"
    print (fact 5)
```

Concepts

Fonction

- C'est l'unité de traitement de ce paradigme
- il s'agit de fonction **pure**, i.e. fonction au sens mathématique

Fonction pure (vs impure)

- Ne dépend QUE des arguments
- Ne dépend PAS d'un état externe
- N'a donc pas d'effets de bords, dû à un changement d'état
- **Donnera toujours le même résultat pour un paramètre donné**

```
// Fonction pure

function add(value: number) {
    return value + 1;
}
add(3); // -> 4
add(3); // -> 4
//Fonction impure

let value = 3;

function add(){
    value += 1;
    return value + 1;
}

add(); // -> 4
add(); // -> 5
```

Transparence référentielle

- Le fait qu'une fonction
    - Renvoie toujours le même résultat pour un ensemble de paramètres d'entrée
    - N'appelle pas d'autres fonctions susceptibles de modifier un état et donc de créer un effet de bord

Transparence référentielle: avantages

- Décomposition d'un problème complexe en parties plus simples, plus facile à comprendre et à tester
- Optimisation de l'exécution grâce à un système de cache de résultat :
    - add(3) : mise en cache de la valeur 4 pour add avec 3 en paramètre
    - add(3) : inutile de faire l'appel : add(3) est en cache, retourne 4 immédiatement
- Parallélisation de l'exécution des fonctions car elles sont indépendantes les unes des autres et d'un contexte

Immutabilité

- Prérequis de la programmation fonctionnelle où les variables ne changent pas après affectation
- Car la mutation d'un élément peut entraîner des effets de bord
- Simplifie la gestion d'état de l'application en réduisant le nombre de bugs intrioduits avec une mutation

```
// Immutabilité en Typescript
// Pour les types primitifs (number, string)
```

```
const my_immutable = "immutable";
my_immutable = "nouvelle valeur" // impossible grâce au const

// pour les objets, const ne suffit

const un_animal = new Animal("Lassy");
un_animal = new Animal("Rex") // Impossible grâce au const
un_animal.name = "Rex" // Impossible grâce au readOnly
// Immutabilité en Typescript
// Pour les objets, il faut ajouter readonly dans la classe

class Animal {
    readonly name: string;

    constructor(name: string) {
        this.name = name;
    }
}

const un_animal = new Animal("Lassy");
un_animal = new Animal("Rex") // Impossible grâce au const
un_animal.nam = "Rex" // Impossible grâce au readOnly
```

Assignation de fonction en Typescript

Les fonctions sont des objets. Elles peuvent être :

- Affectées à des variables
- Passées en paramètre à une fonction
- retournées par une fonction

```
let func_doubleA = function double(a) {
    return a * 2;
}

// le nom de la fonction est facultatif

let func_doubleB = function (a) {
    return a * 2;
}

// func_doubleA et func_doubleB sont
// des variables ayant pour valeur le code d'une fonction
// et sont equivalentes.
// Pour les executer, il suffit de les appelers comme des
// fonctions normales

func_doubleA(10);
func_doubleB(10);

// autre exemple d'affectation

let x = func_doubleA;

x(10);
```

Fonctions Lambda

- Fonction
  - Anonyme
  - utilisée de manière ponctuelle
  - n'effectuant généralement qu'une seule opération
- Syntaxe

```
x => x * 2
```

- Équivaut à

```
function (x) { return x * 2 ;}
```

Concepts

```
// Fonction Lambda

// De même que pour les autres fonctions,
// elle peut être affectée à une variable ou
// être passée en paramètre d'une autre fonction

// Exemple affectation de fonction
let my_func = x => x * 2;
// Exemple de passage en paramètre
map(x => x * 2);
```

Fonction d'ordre supérieur

- Fonction
  - Soit elle prend une ou plusieurs fonctions en paramètres
  - Soit elle renvoie une fonction

```
// Fonction d'ordre supérieur : exemple

function format_devise(devise: string) {
    return function (valeur: number) {
        return valeur + ""
            + devise;    }
}

let format_dollars = format_devise("$");
let format_euros = format_devise("Euros");

console.log(format_dollars(10));
console.log(format_euros(100));
console.log(format_euros(10));
// Fonction d'ordre supérieur: exemple avec Lambda

let format_devise = d => v + "" + d;

let format_dollars = format_devise("$");
let format_euros = format_devise("Euros");
```

```
console.log(format_dollars(10));
console.log(format_dollars(100));
console.log(format_euros(10));
```

Fonction d'ordre supérieur: intérêts

- Générer des fonctions pures, avec un seul argument
  - Principe de "curryfication" (prochain séance)
  - Permettra la composition de fonctions
- Meilleure expressivité du code:
  - met l'accent sur la tâche à accomplir : `format_dollar`
  - plutôt que sur la façon de l'accomplir: `format_devise`

Concepts

La "closure" ou fermeture

- Fonction qui utilise des variables définies dans la portée englobante
- Elle se souvient de l'environnement dans lequel elle a été créée
- Elle "capture" son environnement

```
// la "closure" ou fermeture
//closure-1.ts

function logger(name) {
  let separator = ": ";
  function log(message) {
    console.log(name + separator + message);
  }
  return log;
}
let my_logger = logger("UTC503");
my_logger("J'ai capturé name et separator !");
// closure-2.ts

let logger = (separator) => (name) => (message) => console.log(name + separator
+ message);
let colonLogger = logger(": ");
let my_logger = colonLogger("UTC503");
// Ou aussi let my_logger = logger(": ")("UTC503");
my_logger("J'ai capturé name et separator !");
```

- `log` est capable de manipuler les variables `name` et `separator` qui ne lui sont pas passées en argument et qui sont sensées disparaître à la fin de logger.
- `log` et `separator` sont dits fermés, d'où la closure.

La curryfication

- Transformation d'une fonction à plusieurs arguments en fonction à un seul argument qui retourne une fonction sur le reste des arguments
- Pour :
  - créer une fonction pure
  - permettre la réutilisabilité de code pour de nouvelles combinaisons et compositions

Exemple

- Précédemment la fonction logger aurait pu s'écrire :

```
function logger(name, message) {
  let separator = ": ";
  console.log(name + separator + message);
}
```

- Grâce à la technique de closure, nous avons pu écrire une forme "curryfiée" *(2 premiers exemples)*.
- En effet, logger ne prend qu'un argument, `name` et renvoie une fonction qui prendra en argument message
- Dans

  closure-2.ts

  la curryfication est double, et on aperçoit la capacité de réutilisabilité du code;

  - Création de `colonLogger`
  - On pourrait créer un `spaceLogger` qui aurait un espace comme séparateur

L'application partielle

- La curryfycation permet l'application partielle
- L'application partielle permet d'avoir une fonction réalisant uniquement le calcul demandé
- Dans les exemples précédents, les applications partielles de la fonction `logger` sont :

```
//closure-1:
let my_logger = logger("UTC503");
//closure-2:
let colonLogger = logger(": ");
//closure-2:
let my_logger = colonLogger("UTC503");
```

La récursion

- En programmation fonctionnelle, on privilégie la récursion aux instructions de boucle comme `for` ou `while`
- Pas d'état (compteur) à maintenir, donc pas d'effets de bords

```
// La récursion: exemple avec factorielle

let factorial = n => {
    // Avec boucle for, result et x comme état
    let result = 1;
    for(let x=1; x <= n; x+= 1){
    result = x * result;
    }return result;
}
let recursiveFactorial = n => {
// Version récursive: aucun état
    if (n < 2){
    return n;
    }
    // Else
    return n * recursiveFactorial(n - 1);
}
```

Recherche et transformation de listes

- Domaines où les boucles sont habituelles en langage impératif
- En programmation fonctionnelle, privilégiez
  - **filter()** pour la recherche
  - **map()** pour la transformation

```
// Rechercher avec filter
// list.filter(fonction_de_filtrage_booleenne) renvoie une liste filtrée
// Exemple : filter-1.ts

let list = [1, 9, 2, 8, 5];
let nombrePair = n => n % 2 == 0;
let result = list.filter(nombrePair);
console.log(result); // [2, 8]
// Chaînage des filtres
// Exemple : filter-2.ts

let list = [1, 9, 2, 8, 5];
let nombrePair = n => n % 2 == 0;

// Chaînage de filter
let result = list.filter(nombrePair).filter(n => n > 4);
console.log(result); // [8]
// Mieux vaut composer les fonctions de filtrage
// Moins d'itérations, moins de tableaux temporaires
// Exemple : filter-3.ts

let list = [1, 9, 2, 8, 5];
let nombrePair = n => n % 2 == 0;

// Composition des fonctions de filtrage
let result = list.filter(n => nombrePair(n) && n > 4);
console.log(result); // [8]
// Transformation avec map

// liste_de_A.map(fonction_de_transformation_de_A_vers_B) renvoie une liste_de_B
// Exemple : map.ts

let authors = [{name: 'franquin'}, {name: 'uderzo'}, {name: 'hergé'}];
let upperized_names = authors.map(n => n.name.toUpperCase());
console.log(upperized_names); // [ 'FRANQUIN', 'UDERZO', 'HERGÉ' ]
```

Reduce

- Consiste à prendre un tableau, et à accumuler les éléments de celui-ci à l'aide d'une fonction associative
- Le résultat est UNE valeur dont le type dépend de la fonction appliquée et de la valeur d'initialisation
- Prend 2 paramètres
  - La fonction associative
    - Dont les 2 premiers paramètres sont :
      - La valeur de l'accumulateur (vaut la valeur initiale à la première itération)
      - La valeur du tableau de l'itération courante

        - Renvoie la valeur de l'accumulateur de la prochaine itération
    - Une valeur initiale

```
//reduce-1.ts

let numbers = [1, 2, 3, 4, 5];
// Somme des valeurs; valeur initiale: 0

let sum = numbers.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(sum); // 15
//reduce-2.ts

let letters = ['e', 'm', 'i', 'r'];
// Inverse les éléments du tableau

let reverse = letters.reduce((rev, letter) => [letter].concat(rev), []);
console.log(reverse); // [ 'r', 'i', 'm', 'e' ];
```