Revision UTC503 Covid edition

1. Définitions générales

Le tas:
utilisé lors de l'allocation dynamique de la mémoire durant l'exécution d'un programme informatique. Ou à la demande du programme.

> *Wikipedia:*
> La plupart des programmes ayant des besoins en mémoire dépendant de l'usage qu'on en fait, il est nécessaire de pouvoir, à des moments arbitraires de l'exécution, demander au système d'exploitation d'allouer de nouvelles zones de mémoire, et de pouvoir restituer au système ces zones (libérer la mémoire).

La pile d'exécution (stack):
Allocation dynamique de mémoire qui se fait automatiquement lors d'un appel de sous-routine ou de fonction.

> *Wikipedia:*
> L'exécution d'un programme utilise généralement une pile contenant les cadres d'appel aux routines (fonctions ou procédures) du langage de programmation utilisé.
> Schématiquement, les variables lexicales, c'est-à-dire les variables définies dans la portée textuelle d'une routine, sont :
> . allouées lors de l'entrée dans la routine (entrée de la portée), c'est-à-dire que l'espace est réservé pour ces variables;
> . désallouées automatiquement lors de la sortie de la routine (sortie de la portée), c'est-à-dire que l'espace réservé pour ces variables est dorénavant libre et disponible pour d'autres variables.
>
> Un segment mémoire, dit segment de pile, est utilisé pour ces allocations/libération. Aucun mot clef n'est nécessaire dans le code source du langage supportant la notion de variable lexicale : l'espace est alloué et libéré selon la discipline de pile par définition.

Certains langages, comme le C ou le C++, parlent de variables automatiques au lieu de variables lexicales, mais il s'agit de la même chose.

Instructions:
C'est une ligne de code que l'ordinateur doit exécuter (déclarer une variable, exécuter une fonction...).

Mot-clé:
C'est un mot qui est déjà reconnu comme ayant une fonctionnalité au sein du langage de programmation.

Variable:
C'est une référence à une adresse mémoire qui porte un nom pour plus de facilité d'utilisation, une valeur ainsi qu'un type (Exemple: int).

identifiant:
un identificateur ou identifiant est un mot choisi par le programmeur et qui, tel une étiquette, désigne une donnée du programme : variable, constante, procédure, type, etc. Un identifiant et sa valeur forment une sorte de symbole, comparables à ceux des mathématiques, à la différence qu'en programmation courante la valeur peut changer au cours du temps.

Type
un type de donnée, ou simplement un type, définit la nature des valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent lui être appliqués.

> *Wikipedia*
> Type booléen : valeurs vrai ou faux — ou respectivement 1 ou 0 ;
> Type entier signé ou non signé : valeurs codées sur 8 bits, 16 bits, 32 bits voire 64 bits.
> Les caractères sont parfois assimilés à des entiers codés sur 8 ou 16 bits (exemples : C et Java) ;
> Type réel en virgule flottante.

Structure de données:
Une structure de données est une manière d'organiser les données pour les traiter plus facilement. Une structure de données est une mise en œuvre concrète d'un type abstrait.

tableaux:
Un tableau est une structure de données (variable complexe) représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position, ou indice, dans la séquence. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation.

structures de contrôle :
Les structures de contrôle décrivent l'enchaînement des instructions. Elles permettent des traitements séquentiels, conditionnels ou répétitifs (itératifs).

Exception:
Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend. Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.
`Throw exception, try catch`

Module:
Un module désigne originellement un fichier de code de programmation ou un fichier de librairie statique ou dynamique. Pour reprendre l'image de la programmation objet, un module est une instance unique qui n'utilise pas d'héritage et ne contient aucun module fils. Chaque module peut exporter ou importer certains symboles comme des variables, des fonctions ou des classes. Les modules peuvent se regrouper en package (espace de noms) éventuellement hiérarchique.

> TL:DR : identifie une structure de programmation prermettat d'ajouter de nouvelles fonctionnalités

Import de module:
Rendre un module accessible à un autre fichier.

Commentaire:
Partie de code qui ne s'exécute pas et qui apporte un renseignement pour les autres développeurs, sur le fonctionnement d'une partie du code.

Language interprété:
Le code source peut être directement interprété par un programme interpréteur , python,javascrypt...

Language Compilé:
Le code source peut être compilé par un programme compilateur en code machine directement exécutable par le processeur , C sera compilé par le compilateur gcc.

Language type Bytecode ou code intermédiaire :
Le code source peut être compilé par un programme compilateur en bytecode qui sera interprété ou compilé à la volée par un programme interpréteur , moncode.java compilé par le programme javac en pseudo-code moncode.class interprétable par le programme java.

langage de programmation
sert de moyens de communication... :

- programmeur avec machine
- programmeur avec d'autre programmeur

paradigme :
du mot *παράδειγμα* (paradeïgma) en grec ancien qui signifie « modèle » ou « exemple » une représentation du monde, une manière de voir les choses, un modèle cohérent du monde qui repose sur un fondement défini (modèle théorique) il fournit des problèmes types et des solutions.

2. Paradigmes

Programmation impérative:
Décrit les opérations en séquence d'instructions pour modifier l'état du programme • Introduit les concepts de procédures, structures de contrôle, et structures de données.
exp : PHP

Programmation orientée objet:
Consiste à définir et assembler des briques logicielles appelées objets communiquant entre eux par messages (appel de méthode) pour consulter et modifier leur état

exp :JAVA

Programmation déclarative:
Wikipedia
La programmation déclarative est un paradigme de programmation qui consiste à créer des applications sur la base de composants logiciels indépendants du contexte et ne comportant aucun état interne. Autrement dit, l'appel d'un de ces composants avec les mêmes arguments produit exactement le même résultat, quel que soit le moment et le contexte de l'appel.

En programmation déclarative, on décrit le quoi, c'est-à-dire le problème. Par exemple, les pages HTML sont déclaratives car elles décrivent ce que contient une page (texte, titres, paragraphes, etc.) et non comment les afficher (positionnement, couleurs, polices de caractères...). Alors qu'en programmation impérative (par exemple, avec le C ou Java), on décrit le comment, c'est-à-dire la structure de contrôle correspondant à la solution.

C'est une forme de programmation sans effets de bord, ayant généralement une correspondance avec la logique mathématique.

Programmation descriptive:
Décrit des mises en forme **d'objets graphiques**
• C'est le moteur de rendu du navigateur qui va interpréter et afficher le résultat
Exemples : HTML , CSS

Programmation fonctionnelle:
Un programme est une **fonction**
• Considère le calcul du programme en tant qu'évaluation de fonctions au sens mathématique du terme.
• Difficile à aborder pour les programmeurs impératifs, car façon de penser très différente.
• Très « tendance », car adopté par les langages récents (fonction lambda entre autres) et adapté aux problématiques exprimables sous forme de flux asynchrones.
Exemples : Haskell, OCaml, F#.

Programmation logique:
• consiste à exprimer les problèmes et les algorithmes sous forme de prédicats, règles.
• Utilisé notamment dans le traitement automatique du langage naturel avant le retour des réseaux de neurones artificiels (« deep learning »).
Exemples : Prolog.

3. Revues webconf, Cours, exercices

00 - Pour bien démarrer

Webconf du 28/02/2020_

- Soit le code suivant:

```
var v=5;

functionf(v, x){
   v=v+1;
   return v+x;
}

varz= f(10, 2);
```

Que valent v et z ?

V = 5
Z = 13

- 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 ?
r = 10

- Soit le code suivant en Javascript :

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

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

x est une fonction avec la même définition que f
y = 4
z = 8

- Lors d'un appel de fonction, les paramètres sont copiés sur quelle mémoire ?

  (1) La pile (stack)
  (2) Le tas (heap)

- Lors d'une allocation dynamique de mémoire, quel type de mémoire est utilisée ?

  (1) La pile (stack)
  (2) Le tas (heap)

> Lien en rapport avec ces 2 questions > Lien complémentaire :

- Quel concept POO est utilisé en combinant des méthodes et des attributs dans une classe ?

  (1) Polymorphisme
  (2) Encapsulation
  (3) Abstraction
  (4) Héritage

> **On encapsule des attributs et des méthodes dans une classe**

- Quand on parle de membre de classe, on parle de :

  (1) Uniquement de méthode
  (2) Uniquement d'attribut (propriété)
  (3) Méthode ou attribut

- 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

> **Un objet doit respecter l'interface qu'il promet d'implanter**

- Si l'on respecte le principe d'encapsulation, comment procédons-nous pour accéder etmodifier la valeur des attributs de nos objets ?

  (1) Nous nous servons d'accesseurs et de mutateurs
  (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

> Voir Getter / Setter

- 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

- 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.

> On créé une instance et c'est uniquement sur cette instance que l'on va agir.

- 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.
  (2) Un objet est une description comportant un ensemble de caractéristiques et de fonctionnalités dont chaque classe créée à partir de ce modèle héritera.

> Penser à la gauffre et au moule à gauffre. La gauffre est l'instance (objet)

- Quelle est la différence entre un objet et une instance de classe ?
  Aucune, ce sont des synonymes
- En Java, C#, C++, PHP, ECMAScript 6, quel mot-clé permet de créer une nouvelle instance ?

  (1)instanceof
  (2)new
  (3)create
- 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
  (5)public

> Private : interne à la classe uniquement. Protected : Accessible uniquement à la classe et ses dérivés (héritage)

- Quelle visiblité est la moins permissive ?

  (1)private
  (2)protected
  (3)public

> On accède uniquement à la méthode ou attribut à l'intérieur de la classe uniquement

- 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 autreclasse
- Quelle condition nécessaire et suffisante doit-on avoir pour dire qu'une classe B doithériter 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
- En Java, PHP, ECMAScript 6, quel mot-clé permet de procéder à un héritage ?

(1) herits
(2) extends
(3) childof

- 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

> Override, on pourrait avoir une classe enfant qui hérite d'une classe mère,sert souvent de valeur par défaut et on va override pour les cas particuliers, dans une application par ex on a plein d'animaux qui ont un comportement par défaut mais pour certains qui ne réagissent pas pareil on va redéfinir.

- Quelle est la différence entre redéfinition de méthode et surcharge d'une méthode ?

  Aucune, ce sont des synonymes

- 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);
  }
}
```

> Dès lors que l'on met `extends` il nous faudra obligatoirement une méthode showCalories()

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

  (1) Lorsque l'on souhaite empêcher toute classe d'hériter de la dite class
  (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

- 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.

> Interface != classe abstraite

- exemple

```
public interface Ingestable {
   public void whatHappens();
}

abstract class Food {
   abstract void showCalories;
}

public class Tomatoes extends Food implements Ingestable {
   void showCalories() {
      System.out.println(16);
   }

   void whatHappens() {
      System.out.println("Hum j'adore quand c'est umami");
   }
}

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

   void whatHappens() {
      System.out.println("Frissons dans le dos");
   }
}

public class poison implements Ingestable {
   void whatHappens() {
      System.out.println("Ouuuups...");
   }
}
```

> Un chien et un chat hérite de Animals, mais pas une voiture. Cependant, une voiture, un chat et un chien peuvent tous `bouger()` dans ce cas la on choisira une interface. On va pouvoir lier via l'interface des comportements similaires.

- Une classe peut être héritée par combien de classes filles ?

  (1) Qu'une seule
  (2) Ça dépend des langages
  (3) Une infinité

> Un aliment peut être hérité d'une infinité de classe fille

- De combien de classes une classe fille peut-elle hériter ?

  (1) Qu'une seule
  (2) Ça dépend des langages

> Certains langages autorisent l'héritage multiple (Python par ex), java ne l'autorise pas.

01 - Découverte de Git et Github / Aperçu de Typescript

- ## Webconf du 13/03/2020_

- @todo : Voir si il est pertinent d'approfondir cette webconf

02 - Paradigme de programmation impérative

- ## Webconf du 03/04/2020

- 5 points à retenir :
- **Notion de séquence d'instructions**

```
var nbOranges = 4;

for (i = 0; i < nbOranges; ++i) {
  couperOrange();
  presserOrange();
}
```

Une séquence d'instruction c'est une "suite" d'instruction, ici on assigne une valeur à une variable `nbOranges` et on fait une boucle sur ce nombre d'orange avec des instructions. C'est cette suite d'instruction qui fera notre programme.

- **Notion d'assignation.**

On peut assigner à notre boîte id plusieurs formes d'expressions.

```
var id = "exp";
// Ou encore
var id = (exp / 10) * 10;
// Ou encore
var id = exp();
```

La boite est notre identifiant et on est capable d'assigner une expression a cette id sous plusieurs forme :

  - une chaine de caractère
  - un calcul
  - une fonction

On peut assigner toute sorte de chose à notre identifiant.

! Attention de pas confondre :

```
var id = "exp";
```

> une boite id qui stocke "exp" (une chaine de catactère)

avec

```
id == "exp";

//true
```

> Ici on teste l'égalité du contenu id avec la valeur "exp"

- **Instruction conditionnelle**

```
   var (jaiFaim) {
      return "miam, je mange";
   } else {
      return "je mange quand même";
   }
```

Exemple en JS, notre if va essayer de savoir si la condition est respectée et va retourner une valeur selon.

Exercice if / else:

```
function fizzBuzz(number) {
  if (number % 3 == 0 && number % 5 == 0) {
    return "FizzBuzz";
  }

  if (number % 3 == 0) {
    return "Fizz";
  }

  if (number % 5 == 0) {
    return "Buzz";
  }

  return number;
}

console.log(fizzBuzz(15));
console.log(fizzBuzz(10));
console.log(fizzBuzz(9));
```

> Rappel modulo donne le reste de la division euclidienne 6 % 3 =0 ; 9 % 7 = 2

- **boucle for**

ind 0	ind 1	ind 2	ind 3	ind 4	ind 5
8	5	-1	5	4	3

> ! le premier indice d'un tableau commence à l'indice 0.

```
var tableau = [1, 2, 3]; // ex initialisation tableau en javascript
```

Si l'on souhaite récupérer la valeur :

```
tableau[indice];
```

Pour boucler dans un tableau et afficher toutes les valeurs :

```
for (i = 0; i < jeSuisUnTableau.length - 1; ++i) {
  console.log(jeSuisUnTableau[i]); // instruction qui sera réalisée à chaque
passage (boucle)
}
```

Ex: la boucle for, on a 3 parties: l'initialisation `i=0`, la condition de sortie `i < jeSuisUnTableau.length - 1`, et le "pas" de l'itération à chaque passage : `++i` .
"tant que i est inférieur a (6-1) (5), on fait ++i a chaque boucle.

`++i` ou `i++` ?

```
var i = 42;
var plop = i++;
console.log(plop); // 42
```

plop va prendre la valeur de `i`, 42, puis `i` sera incrémenté **après** l'affectation à plop

```
var i = 42;
var plop = ++i;
console.log(plop); //43
```

Ici plop aura la valeur 43 car l'incrémentation de `i` sera faite avant l'affectation à plop.

- **boucle while**

ind 0	ind 1	ind 2	ind 3	ind 4	ind 5
8	5	-1	5	4	3

```
var i = 0;
while (i < jeSuisUnTableau.length) {
  console.log(jeSuisUnTableau[i]);

  if (jeSuisUnTableau[i] < 0) {
    --i;
  } else {
    i += 2;
  }
}
```

```
Tant que i est inférieur à la longueur du tableau
afficher le contenu de l'index i

si la valeur contenue dans l'index est inférieure à 0
   décrémenter i de 1 (on va donc à l'index (case précédent)
sinon
   incrémenter i de 2 (on va deux cases plus loin)

on affiche donc : 8 -1 5 5 3
```

- **Branchement avec les goto**

```
intmain(){
// Factorielle de N = N * (N - 1) * (N - 2) * (N - i) * ... * 1
int n = 5;
int factorielle = 0;

int i = n;
```

```
factorielle = 1;
boucle_debut:
factorielle = factorielle * i;
i = i - 1;
if (i > 0)
goto boucle_debut;

printf("La factorielle de %d est égale à %d\n", n, factorielle);
}
```

```
initialisation de n à 5
initialisation de factorielle à 0

initialisation de i à la valeur de n, ici 5
factorielle vaut désormais 1

déclaration du marqueur "boucle_debut"
factorielle vaut désormais factorielle * i (5 au démarrage)
i vaut désormais sa valeur amputé de 1 (4 ensuite et ainsi de suite)
si i est supérieur à 0 (condition de sortie)
on retourne a boucle_debut tant que la condition de sortie est vraie (true)
```

> L'instruction goto ("va a"), renvoi l'exécution du programme vers la section passée en paramètre, c'est une instruction qu'il est déconseillé d'utiliser, l'enchevêtrement de goto donne un code peu lisible, difficile a maintenir.

- **Fonctions / procédures**

```
intfactorielle(int n){
   if (n == 0) // si n en paramètre vaut 0 (prévention)
   return 0; // souvent quand il n'y a qu'une seule instruction après le if, les {} sont facultatives
   if (n == 1) // condition de sortie
   return 1;
   return n * factorielle(n - 1); // on retourne cette même fonction avec un nouveau paramètre (récursivité)
}

int main() {
   // Factorielle de N = N * (N - 1) * (N - 2) * (N - i) * ... * 1
   int n = 5;
   printf("La factorielle de %d est égale à %d\n", n, factorielle(5));
}
```

> Notion de responsabilité: un jeux vidéo comporte une fonction Game Over, à cette fonction on lui passe 2 **arguments**, le premier "hero" le deuxième c'est "vilain", si `hero` alors fin du combat, si `vilain` alors fin du combat.

```
function finDuCombat(hero, vilain) {
  if (hero.pv === 0) {
    return true; // fin du combat
  } else if (vilain.pv === 0) {
    return true; // fin du combat
  }
  return false;
}
```

cette fonction sert juste a retourner true si `hero` ou `vilain` n'a plus de pv, sinon on retourne `false`. Le problème ici c'est que notre méthode a plusieurs responsabilités, elle doit dire si le combat est fini ou pas, et également si hero ou vilain est encore en vie. Elle a 3 responsabilité. En général on essaye de rester **SOLID**, S => *single responsability principle*.
En POO principalement, **nos méthodes et nos classes ne doivent avoir qu'une seule responsabilité**, il nous faut la découper.

```
function finDuCombat(hero, vilain) {
  if (isDead(hero)) {
    // notre fonction n'évalue plus les pv du personnage
    return true;
  } else if (isDead(vilain)) {
    return true;
  }
  return false;
}

function isDead(personnage) {
  return personnage.pv <= 0;
}
```

Exercices

1. Sachant qu'une instruction de bouclage peut être vue comme la combinaison d'une instruction de branchement conditionnel et d'une instruction de saut, codez l'algorithme suivant sans instruction de bouclage, mais en utilisant goto et if/else :

```
int main(){
   // factorielle de N = N * (N - 1) * (N - 2) * ( N - i) * ... 1
   int n = 5;
   int factorielle = 0;
   // Codez ici
   printf(" La factorielle de %d est égale à %d\n", n, factorielle);
}
```

correction :

```
int main() {
// Factorielle de N = N * (N - 1) * (N - 2) * (N - i) * … * 1
int n = 5;
int factorielle = 0;

int i = n;
factorielle = 1;

boucle_debut:
```

```
factorielle = factorielle * i;
i = i - 1;
if (i > 0)
   goto boucle_debut;

printf("La factorielle de %d est égale à %d\n", n, factorielle);
}
```

> Rien de particulier a commenter, il s'agit de l'exemple repris plus haut dans la webconf.

2. Codez la factorielle avec une instruction de bouclage

```
int main(){
// factorielle de N = N * (N - 1) * (N - 2) * ( N - i) * ... 1
int n = 5;
int factorielle = 0;
// Codez ici
printf(" La factorielle de %d est égale à %d\n", n, factorielle);
}
```

correction :

```
int main() {
// Factorielle de N = N * (N – 1) * (N – 2) * (N – i) * … * 1

int n = 5;
int factorielle = 0;
factorielle = 1;
int i = n;

for(int i = 1; i <= n; i++) // utilisation d'une boucle for
   factorielle = factorielle * i;

// Ou encore
while (i > 0){  // utilisation d'une boucle while
factorielle = factorielle * i;
i --;
}

// Ou
while(i > 0)
factorielle *= i--;

printf("La factorielle de %d est égale à %d\n", n, factorielle);
}
```

> Premier cas, on utilise une boucle `for`, on se basera sur la valeur de `i`, notre cas de sortie sera `i <= n`, soit quand `i` atteind **5**. On boucle 5 fois car `i++` incrémente `i` de 1 à chaque passage.
>
> Deuxième cas, on utilise une boucle `while`, "tant que" `i > 0` on constinue notre boucle récursive, à l'intérieur de notre boucle, on décrémente a chaque passage `i` de 1.
>
> toisième cas, pas de recurisivité, `factoriel` sera égal à `factoriel * i`. **faire `i--` décrémente `i`, une fois l'affectation réalisée** (voir plus haut i++ || i--).

3. Codez la factorielle avec une fonction contenant une instruction de bouclage :

```
  int factorielle(int n) {
  // codez ici
}
int main() {
  int n = 5;
  printf(" La factorielle de %d est égale à %d\n", n, factorielle);
}
```

correction :

```
 int factorielle(int n){
 if (n == 0)
   return 0;

 // Sinon
 int factorielle = 1;
 while(n > 0){
   factorielle = factorielle * n;
   n = n - 1;
 }
 return factorielle;
 }

 int main() {
 // Factorielle de N = N * (N - 1) * (N - 2) * (N - i) * … * 1
 int z = 5;
 printf("La factorielle de %d est égale à %d\n", n, factorielle(z));
 }
```

> On crée donc une fonction `factorielle` qui prend en argument un entier `n`.
> Si `n` est égal à 0, on retourne 0 sinon on rentre dans une boucle `while` sans récursion comme vu dans l'exercice 2.

4. Codez la factorielle avec une fonction récursive (sans instruction de bouclage)

```
int factorielle(int n){
  //codez ici
}
int main(){
  int n = 5;
  printf(" la factorielle de %d est égale à %d\n"), n, factorielle(n));
}
```

correction :

```c
int factorielle(int n){
 if (n == 0)
   return 0;
 if (n == 1)
   return 1;
 return n * factorielle(n - 1);
}

 int main() {
 // Factorielle de N = N _ (N - 1) _ (N - 2) _ (N - i) _ … \* 1
 int n = 5;
 printf("La factorielle de %d est égale à %d\n", n, factorielle(5));
 }
```

> On utilise ici une fonction récursive, c'est à dire que cette fonction s'appelle elle même jusqu'a satisfaire la condition de sortie. Ici cette condition est :

```c
if (n == 0)
  return 0;
if (n == 1)
  return 1;
```

> la condition de sortie donne en quelque sort la première solution à la pile d'appel de fonction.
> Imaginons que quelqu'un demande:
> "Quelle est la valeur de 5! ?"
> Que son interlocuteur demande à son tour, "Ben déjà c'est quoi 4!" ?
> et ainsi de suite jusqu'a ce qu'on finisse par dire: "Mais 1! = 1", donc 2! = 2, 3 = 6 etc etc etc...

5. Limite de la récursivité

   Quelle limite identifiez-vous aux appels récursifs ?

   Indice : un site très populaire chez les codeurs porte son nom.

   Stack overflow

   [wikipedia]

   > En informatique, un dépassement de pile ou débordement de pile (en anglais, stack overflow) est un bug causé par un processus qui, lors de l'écriture dans une pile, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.

   Les appels récursifs si ils sont robustes, sont aussi très gourmand en mémoire car a chaque appel on recopie dans celle-ci la fonction avec ses variables et instructions. Un trop grand nombre d'appel saturera l'espace alloué à notre processus.

6. Pour chaque valeur d'un tableau d'entrée, map consiste à calculer une autre valeur selon un algorithme et à la stocker à la même position dans un tableau résultat. Complétez les 4 fonction mapX.

```c
#include <stdio.h>

#define N 5 //constante de préprocesseur

//Cette directive permet de définir une constante de préprocesseur.
```

```
//Cela permet d'associer une valeur à un mot. Voici un exemple :

void mapDouble(int e[], int r[]){

  //Aide : lire la ième case du tableau e : e[i]
  //Aide : affecter la ième case du tableau r : r : r[i] = valeur;
  //Aide : le tableau est accessible de 0 à N - 1, ie de e[0] à e[4] en
l'occurence

}

void mapDouble (int e[], int r[] ){
  //Codez ici
}
void mapTriple (int e[], int r[] ){
  //Codez ici
}
void mapSquare( int e[], int r[] ){
  //Codez ici
}
void mapMaximize3(int e[], int r[] ){
  //Codez ici
}

int main() {
  int e[N] = {1, 5, 2, 4, 3};
  int r_double[N];
  mapDouble(e, r_double); // r_double contiendra {2, 10, 4, 8, 6}
  int r_triple[N];
  mapTriple(e, r_triple); // r_triple contiendra {3, 15, 6, 12, 9}
  int r_square[N];
  mapMaximize3(e, r_square); //r_square contiendra {1, 25, 4, 16, 9}
  int r_maximize3[N];
  mapMaximize3(e, r_maximize3); // r_maximize3 contiendra {1, 3, 2, 3, 3}
}
```

correction :

```
#define N 5

void mapDouble(int e[], int r[]){
for (int i = 0; i < N; i++)
r[i] = e[i] * 2;
}

void mapTriple(int e[], int r[]){
for (int i = 0; i < N; i++)
r[i] = e[i] * 3;
}
void mapSquare(int e[], int r[]){
for (int i = 0; i < N; i++)
r[i] = e[i] * e[i];
}
void mapMaximize3(int e[], int r[]){
for (int i = 0; i < N; i++)
if (e[i] <= 3)
  r[i] = e[i];
```

```
else
  r[i] = 3;

// Ou remplacer le if / else par l'opérateur ternaire ()?:
r[i] = e[i] <= 3?e[i]:3;
}

int main() {
int e[N] = {1, 5, 2, 4, 3};

int r_double[N]; mapDouble(e, r_double); // r_double contiendra {2, 10, 4,
8, 6}

int r_triple[N];
mapTriple(e, r_triple); // r_triple contiendra {3, 15, 6, 12, 9}

int r_square[N];
mapSquare(e, r_square); // r_square contiendra {1, 24, 4, 16, 9}

int r_maximize3[N];
mapMaximize3(e, r_maximize3); // r_maximize3 contiendra {1, 3, 2, 3, 3};
}
```

> Dans les faits l'exécution ici est assez simple, on initialise un premier tableau `int e[N] = {1, 5, 2, 4, 3};` , puis on en crée un nouveau pour chaque opération voulue `r_double ; r_triple` ... on appelle ensuite la fonction de l'opération voulue en lui passant le tableau de base `e[]` et le résultat est stocké dans le deuxième tableau en argument. Simple. Cette technique montre vite ces limites, on réécrit beaucoup de code identique et les fonctions elles-même n'ont aucune modularité.

7. Dans ex6, d'une fonction à l'autre, n'avez-vous pas éprouvé une sensation de "déjà-vue" ?

   Peut-on factoriser ce code ?
   Quel mécanisme nous faudrait-il ?

   Lambda fonction ou pointeur de fonction.

```
#define N 5

void mapDouble(int e[], int r[]){
  for (int i = 0; i < N; i++)
    r[i] = e[i] * 2;
}

void mapTriple(int e[], int r[]){
  for (int i = 0; i < N; i++)
    r[i] = e[i] * 3;
}
void mapSquare(int e[], int r[]){
  for (int i = 0; i < N; i++)
    r[i] = e[i] * e[i];
}
void mapMaximize3(int e[], int r[]){
  for (int i = 0; i < N; i++)
    r[i] = e[i] <= 3?e[i]:3;
}
```

```
int my_double(int v){
    return v * 2;
}

int my_triple(int v){
    return v * 3;
}

void map(int e[], int r[], int (*pointeurSurFonction)(int)){
  for (int i = 0; i < N; i++)
    r[i] = pointeurSurFonction(e[i]);
}

int main() {
  int e[N] = {1, 5, 2, 4, 3};
  int r_double[N];

  mapDouble(e, r_double); // r_double contiendra {2, 10, 4, 8, 6}
  int r_triple[N];

  mapTriple(e, r_triple); // r_triple contiendra {3, 15, 6, 12, 9}
  int r_square[N];

  mapSquare(e, r_square); // r_square contiendra {1, 24, 4, 16, 9}
  int r_maximize3[N];

  mapMaximize3(e, r_maximize3); // r_maximize3 contiendra {1, 3, 2, 3, 3};
  int r_double_2[N];

  // Passage de la fonction 'my_double' en paramètre de map

  map(e, r_double_2, &my_double);
  int r_double_2[N];

  // Passsage de la fonction 'my_triple' en paramètre de map
  map(e, r_double_2, &my_triple);

}
```

> Pointeur vers une fonction retournant un int et prenant un int comme paramètre: `int (*pf)(int);` . La fonction `void map(int e[], int r[], int (*pointeurSurFonction)(int))` prend donc en argument la table de départ `int e[]` , la table de sortie int `r[]`, et le pointeur vers la fonction qui effectuera l'opération `int (*pointeurSurFonction)(int))`.
> l'appel s'effectue de cette manière: `map(e, r_double_2, &my_double);` avec `&my_double` l'appel à l'adresse de la fonction (le pointeur).

8. Analyse de 3 codes:

   code 1:

```
dessineTriangle( int x, int y, bool gras ) {
  positionneCrayon(x, y);
  traceTriangle();
  if (gras) {
    positionneCrayon(x, y);
    pause(5);
```

```
    traceTriangle();
  }
}

dessineEtoile(int x, int y, bool gras, int nbBranche){
  positionneCrayon(x, y);
  traceEtoile(nbBranche);
  if (gras){
    positionneCrayon(x, y);
    pause(5);
    traceEtoile(nbBranche);
  }
}
```

code 2:

```
dessine(int figure, int x, int y, bool gras, int nbBranche){
  positionneCrayon(x ,y);
  swicth(figure){
    case TRIANGLE: traceTriangle(); break;
    case ETOILE: traceEtoile(nbBranche); break;
  }
  if (gras){
    PositionneCrayon(x, y);
    pause(5)
    switch(figure) {
      case TRIANGLE: traceTriangle(); break;
      case ETOILE: traceEtoile(nbBranche); break;
    }
  }
}
```

code 3:

```
dessine(int figure, int x, int y, bool gras, int nbBranche) {
  positionneCrayon(x, y);
  trace(figure, nbBranche);
  if(gras){
    PositionneCrayon(x, y);
    pause(5);
    trace(figure, nbBranche);
  }
}
```

Que pensez-vous de ces 3 codes : avantages / inconvénients ?

Ce qu'il faudrait c'est le paradigme de programmation Objet,
car on pourrait redefinir la méthode 'trace' avec un comportement spécifique pour les classes Triangle et Etoile.
Ca s'appelle du polymorphisme.

9. Pour le fun

   Dessiner 5 étoiles en Logo

```
to star
; dessine une étoile
```

```
rt 18
repeat 5 [ fd 50 rt 144 fd 50 lt 72]
lt 18
end
to move
; se déplace à droite
penup
rt 90
fd 150
lt 90
pendown
end
to n_stars :nombre
; dessine 5 étoiles
repeat :nombre [ star move]
; cache la tortue
hideturtle
end
to five_stars
n_stars 5
end
to init
clearscreen
; pinceau de couleur jaune
setpencolor 6
; se placer à gauche de l'écran
penup
lt 90
fd 300
rt 90
pendown
end
init
five_stars
hideturtle
```

03 - Programmation Orientée Objet

Webconf du 17/04/2020

Big Picture

POO Une histoire de gaufres

> Analogie de la gaufre et de la moule à gaufre

- Le moule à gaufres est notre classe
- La gaufre est l'objet *(instance)*

```
// Notre moule à gaufres
class Gaufre {
  // Du code
}

// Une gaufre
var gaufre = new Gaufre();
```

Pourquoi ? Quel est l'intérêt d'un moule à gaufres et des gaufres ?
L'élément central est de ne pas se répéter au niveau du code -> **D**on't **R**epeat **Y**oursel (D.R.Y)

Pour commencer nous souhaitons faire des gaufres colorés.
On va créer une classe gaufre qui précisera la couleur.

```
classGaufre{
  couleur: string; // attribut/propriété initialisé par le constructeur

  construct(couleur: String) {
    this.couleur = couleur; // Notre constructeur affecte à la propriété
d'instance couleur la valeur passée en argument
    }
  }

  var gaufreRose = new Gaufre('rose');  // instance avec paramètre de
couleur "rose"
  var gaufreVerte = new Gaufre('verte'); // instance avec paramètrede
"verte"
```

Et nous créons 2 instances de gaufre (2 objets).
On a d'un coté un moule à gaufre qui est capable de générer des objets a partir de la classe gaufre en lui précisant les paramètres attendus à la construction.

`this.couleur = couleur;` Pour construire notre gaufre nous devons spécifier la valeur du paramètre `couleur` de l'instance (`this` = cet(te))

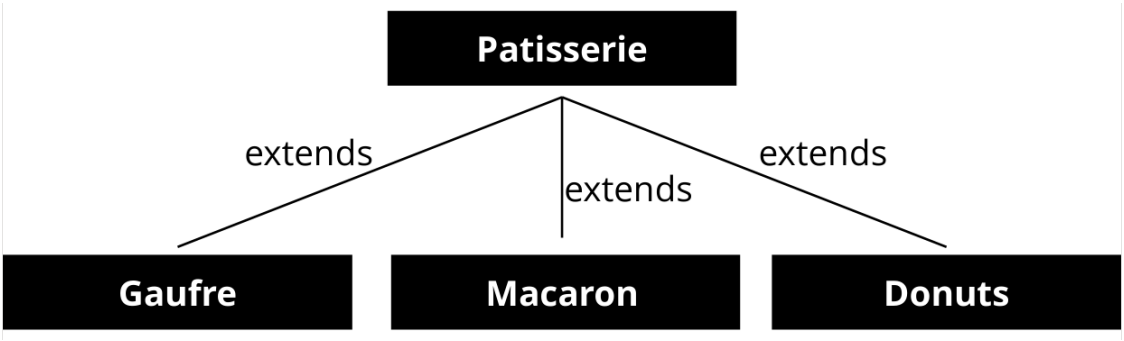
On souhaite que notre gaufre se comporte différemment en fonction de sa couleur. Une gaufre n'est pas comestible du moment ou elle n'a pas une couleur verte.

```
class Gaufre {
  couleur: string;

  constructor(couleur: string) {
    this.couleur = couleur;
  }
  estComestible() {
    if (this.couleur == "verte") {
      return false;
    }
    return true;
  }
}
var gaufreRose = new Gaufre("rose"); // instanciation de la gaufre avec la
couleur rose
var gaufreVerte = new Gaufre("verte"); // instanciation de la gaufre avec la
couleur verte

gaufreRose.estComestible(); // true
gaufreVerte.estComestible(); // false
```

comme vu précédemment la méthode `estComestible` aurait pu s'écrire :

```
  estComestible() {
  return this.couleur == "verte" ? false : true;
  }
```

Notre gaufre peut avoir des relations avec d'autres objets



Notre gaufre peut tout a fait interagir avec d'aurtes objets, notre gaufre peut hériter d'un autre objet qui est patisserie.

Ce que cela donne niveau code :

```
class Patisserie {
  estComestible() {
    return true;
  } // retourne true par défaut
}

class Gaufre extends Patisserie {
  // le mot clé extends précise que l'on hérite de la classe mère Patisserie
  couleur: string;

  constructor(couleur: String) {
    this.couleur = couleur;
  }

  estComestible() {
    // override il y a un comportement spécifique
    if (this.couleur == "verte") {
      return false;
    }

    return true;
  }
}

class Macaron extends Patisserie {
  // pas d'override Macaron.estComestible() retourne true par défaut (valeur de la classe mère)
  // pas de construct car on n'introduit pas de méthode / attibrut particulier
}
```

`super` : mot clé qui permet d'injecter lors de la construction de notre objet les propriétés de la classe mère.

Pour accompagner notre gaufre quoi demieux qu'un café ? (insérer ici une référence au bureau des légendes).
On a une machine à café et plusieurs actions sont possibles :
Certaines actions ici sont réalisables directement par l'utilisateur et d'autres sont faites par la machine elle-même.

- Lancer mon café -> *Utilisateur*

- Chauffer l'eau -> *Machine*
- Allumer la machine -> *Utilisateur*
- Moudre le café -> *Machine*

Action	Visibilité
Lancer mon café	+
Chauffer l'eau	-
Allumer la machine	+
Moudre le café	-

```
class MachineACafe {
  power: boolean;

  constructor() {
    this.power = false;
  }

  public allumerMachine() {
    // public méthode visible : MachineACafe.allumerMachine()
    this.power = true;
  }

  public lancerCafe() {
    if (this.power) {
      this.chaufferLEau();
      this.moudreLesGrains();
      // Code
    }
  }

  private chaufferLEau() {
    // Méthode non visible, mécanique interne à la machine
    // Code
  }
  private moudreLesGrains() {
    // idem
    // Code
  }
}
```

Nos méthodes ou attributs de notre classe ne sont accessibles depuis l'extérieur que si nécessaire.
Ceci afin d'avoir une approche défensive : si ce n'est pas une nécessité d'y accéder depuis l'extérieur on ne le rend pas visible (effets de bords, sécurisation des données, confidentialité...)

Les 4 piliers de la POO :

- Abstraction
- Encapsulation
- Héritage
- Polymorphisme

L'abstraction :

Reprenons l'exemple de la gaufre :

```
abstract class Patisserie {
  abstract estComestible() {}
}
```

En rajoutant absctract, nous obtenons une classe abstraite. Cette classe ne sera plus instanciable, on instinciera des gaufres mais pas patisserie, le corps de la méthode ne possède que des méthodes abstraites (méthode sans corps).
Déclarer une méthode abstraite obligera les enfants à implanter quelque chose dans cette méthode.

Un des risque que l'on peut avoir à définir un comportement par défaut dans patisserie, c'est des effets de bords Par ex, un donut qui instancie patisserie et qui ne serait pas comestible retournerait true pour `Donut.estComestible()` ?

L'encapsulation :

Diagramme UML :

Données:

Machine a café
- température : number
+power : boolean

Méthodes qui manipulent les données:

- chaufferLEau() : void
- moudre() : void
+ allumer(): void
+lancer(): void

C'est ce regroupement qui va être l'encapsulation, notre machine à café contient des données qui seront manipulées par des méthodes qui seront ou non accessibles depuis l'extérieur.

Pourquoi `public` / `private` ? : Notre machine a café va devoir garantir l'intégrité des données, on va faire en sorte que l'on ne puisse pas modifier ces données sans notre accord, il faut en restreindre l'accès.

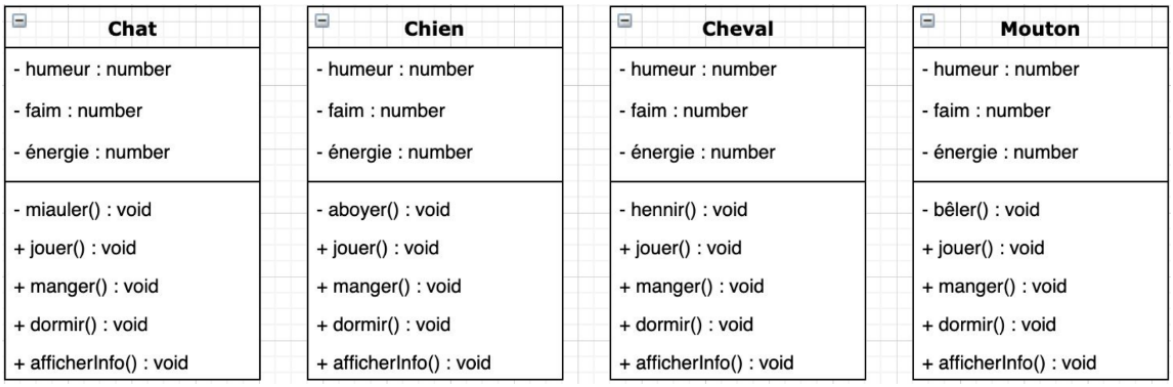
getters & setters : (accesseurs) définit un moyen sûr d'accès et de modifications des données encapsulées.

Protected : Un private mais qui n'est accessible que depuis les classes héritées mais pas depuis l'extérieur.

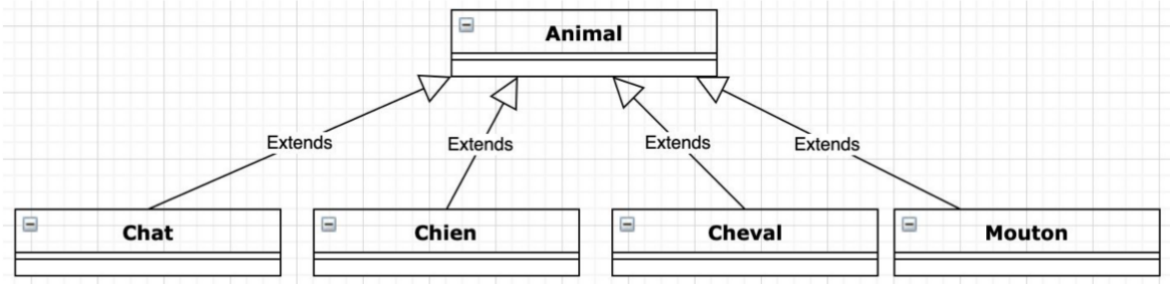
Public : open bar !

L'héritage :

Chat, Chien, Cheval, Mouton sont des...



Avec 4 classes distinctes on se retrouve avec beaucoup de redondance de code (pas canada **DRY**)



On peut organiser les éléments similaire sous la forme d'un héritage depuis une classe mère.

! Attention à l"héritage, lien à étudier :
go

Polymorphisme :

Exemple avec le design pattern DP Factory

- Outils de communication
- Catalogue
- Intelligence collective

On a regoupé au sein d'un catalogue des pratiques efficaces suite à des retours d'expérience de développeurs.

Patron de conception factory

On va avoir une classe *main de poker* a qui on va donner 5 cartes (un tableau de 5 cartes), et à partir de ce tableau de 5 cartes, elle nous ressortira le type (paire d'as ...) et le kicker (quelle carte va départager une égalité).

Si on devait l'implémenter on aurait quelque chose qui ressemble à ça :

```
class PokerHand {
  cards: Array<String>;

  constructor(cards: Array<String>) {
    this.cards = cards;
  }

  type() {
    // Si QFR
    // sinon si QF
    // sinon si Carre
    // ...
  }
```

```
  kicker() {
    // Si type() === brelan
    // Si type() === paire
    // Si type() === double paire
  }
}
```

Code très difficile à maintenir: par ex si on doit rajouter des nouvelles règles.
Une des possiblités pour éviter ce problème est d'utiliser le design pattern factory.

ComboFactory, c'est une classe qui reçoit les 5 cartes et puis a coté de celle-ci on va avoir une classe par combo:

- une classe QFR
- une classe QF
- une classe carre
- etc ...

Notre classe ComboFactory va interroger dans le bon ordre, *Es-tu une QFR?*, *es-tu une QF?*, etc ...
Dès qu'elle tombe sur une réponse positive, elle va instancier la classe *carre* (par ex) et celui-ci va hérité des choses communes à tous les combos.

Je n'ai rien compris au code en RUBY...

Exercices :

Ex1: Figures géométriques

- Une figure a un nombre de côtés.
- Tous les côtés ont la même taille.
- La taille varie d'une instance de Figure à une autre.
- Une figure a un périmètre.
- Une figure a une aire.
- On pourra demander à une figure un rapport retourné dans une chaîne de caractères, contenant son type, ses propriétés, son périmètre, son aire et autre propriétés spécifiques.
- Un triangle est une figure.
- La hauteur d'un triangle de côté A est A x racinecarree(3) / 2
- Un losange est une figure.
- On pourra demander à un losange si c'est un carré
- Selon ces informations, complétez le code TypeScript suivant:

```
let t1 = Triangle(5);
let t2 = Triangle(10);
let l1 = losange(5, 6); // diagonale 1, diagonale 2
let l2 = losange(4, 4);

let figures: Figure[] = [t1, t2, l1, l2];

for (let f of figures) {
  console.log(f.rapport());
}
```

Correction:

```
export abstract class Figure {
```

```
  //notre classe abstraite va contenir tous les éléments communs à nos figures

  private nbCote: number; // A noter le modificateur private, servira uniquement à perimetre()
  protected tailleCote: number; // protected > ne sera accessible qu'a la classe et les classes filles

  constructor(nbCote: number, tailleCote: number) {
    this.nbCote = nbCote;
    this.tailleCote = tailleCote;
  }

  perimetre(): number {
    // methode d'instance
    return this.nbCote * this.tailleCote;
  }

  abstract aire(): number; // méthode abstraite (sans  corps), son implantation sera obligatoire dans les classes dérivées

  rapport(): string {
    return `Aire: ${this.aire()}, Perimetre: ${this.perimetre()}`;
  }
}

class Triangle extends Figure {
  // extends -> dérive de la classe Figure
  constructor(tailleCote: number) {
    super(3, tailleCote); // on appelle le constructeur de la classe mère
  }

  aire() {
    return (Math.pow(this.tailleCote, 2) * Math.pow(3, 0.5)) / 4; // Implantation de la méthode aire() obligatoire
  }

  rapport() {
    return "Triangle, " + super.rapport(); // on précise la méthode mère en l'invoquant via super
  }
}

class Losange extends Figure {
  private diag1: number;
  private diag2: number;

  constructor(diag1: number, diag2: number) {
    super(4, Math.pow(Math.pow(diag1 / 2, 2) + Math.pow(diag2 / 2, 2), 0.5));
    this.diag1 = diag1;
    this.diag2 = diag2;
  }

  estUnCarre() {
    return this.diag1 == this.diag2;
  }

  aire(): number {
    return (this.diag1 * this.diag2) / 2;
```

```
  }

  rapport() {
    return (
      "Losange, " + super.rapport() + " est-il un carré ? " + this.estUnCarre()
    );
  }
}

let t1 = new Triangle(5);
let t2 = new Triangle(10);
let l1 = new Losange(5, 6); // diagonale1, diagonale2
let l2 = new Losange(4, 4);

let figures: Figure[] = [t1, t2, l1, l2];
for (let f of figures) {
  console.log(f.rapport());
}
```

04 - Programmation fonctionnelle

- Webconf du 30/04/2020
    - Une approche impérative anticipe une suite d'instructions à suivre dans l'ordre et qui va modifier l'état du programme.
    - Une approche déclarative, décrit des règles indépendantes qui seront appelées en fonction d'un évènement
- **Impératif**

En reprenant l'exemple de FizzBuzz qui itère sur un tableau :

```
let numbers = [12,15,4];
let newNumbers = [];

for(i = 0; i < numbers.length; i++) {
    let res;

    if(numbers[i] % 3 == 0 && numbers[i] % 5 == 0) {
    res = 'FizzBuzz';
    } else if(numbers[i] % 3 == 0) {
    res = 'Fizz';
    } else if(numbers[i] % 5 == 0) {
    res = 'Buzz';
    } else {
    res = numbers[i]
    }
    newNumbers.push(res);
}

console.log(newNumbers);
```

et nous retourne un nouveau tableau de valeur `newNumbers`

- **Déclaratif**

```
function fizz(x: number): string {
  return x % 3 == 0 ? "Fizz" : "";
}
function buzz(x: number): string {
  return x % 5 == 0 ? "Buzz" : "";
}
function fizzbuzz(x: number, fizz: Function, buzz: Function): string {
  return fizz(x) == buzz(x) ? x.toString() : fizz(x) + buzz(x);
}
console.log([12, 15, 4].map((x) => fizzbuzz(x, fizz, buzz)));
```

La démarche est différente, dans un premier cas on déclare une "brique" dont le but est de faire une opération et ensuite on aura des briques un peu plus grosse qui les assemblera.

En conclusion :

- Impératif vs déclaratif :
  - Impératif : On modifie l'état de notre programme en appliquant une suite d'opérations.
  - Déclaratif : On déclare ce dont on a besoin (des briques de différentes couleurs / tailles, ou des assemblages de briques) Puis on s'en sert pour obtenir le résultat souhaité

La programmation fonctionnelle est une sous catégorie de la programmation déclarative.

Pourquoi faire de la programmation fonctionnelle ?

- La notion de concurrence, exécuter plusieurs tâches en parallèle (meilleure gestion des états au niveau de la concurrence), moins de problèmes de concurrence pour l'accès aux données.
- La "Scalabilité", pour une application qui va être amenée a grossir dans le temps, l'approche fonctionnelle étant issue de plusieurs briques simples se sera plus agréable à gérer qu'une approche impérative trop monolithique.

Think different:

Exemple impératif :

2 fonctions, crédit / débit :

```
function credit(value) { account += value;
}
function debit(value) { account -= value;
}
var account = 0;
// Début de partie
credit(1000);
// Je fais des courses
debit(110);
// Le loyer tombe
debit(700);
// affiche le solde restant
console.log(account);
```

```
const operations = [
  // Début de partie
  { amount: 1000, type: "credit" },
  // Je fais des courses
  { amount: 110, type: "debit" },
```

```
  // Le loyer tombe
  { amount: 700, type: "debit" },
];

const totalCreds: number = operations
  .filter((o) => o.type == "credit") // Filtre, on conserve les opérations
crédit
  .map((c) => c.amount) // On map les valeurs "amount" du filtrage précédent
  .reduce((amount, acc) => amount + acc); // Avec reduce on fait la somme des
valeurs filtrées

const totalDebs: number = operations
  .filter((o) => o.type == "debit")
  .map((d) => d.amount)
  .reduce((amount, acc) => amount + acc);

const account = (totalCredit: number) => (totalDebit: number) =>
  totalCredit - totalDebit;

// Affiche le solde restant
console.log(account(totalCreds)(totalDebs));
```

On peut modifier la façon dont on pense la base de données, plutôt que de stocker la valeur account dans le temps, on ne stocke que les opérations.

Avantage de stocker les opérations plutôt que la valeur account ? :

- Dans le premier cas, on fait évoluer notre compte en banque avec une suite d'instructions
- Dans le second cas, on calcule la valeur à partir des différents évènements qui se sont produits

Notre BDD pourrait alors être un recueil des évènements passés sur lequel on applique des fonctions...

Plus d'erreurs irréversibles ? Si une erreur d'écriture intervient (crédit au lieu de débit, mauvaise affectation de l'opération ...) on peut simplement corriger la base et retrouver le bon amount.

Event sourcing = on stocke les différents évènements et on va déterminer ce dont on a besoin au travers de ses évènements.

Modèle d'approvisionnement en événements

Testabilité :

- Limiter les régressions (si jamais quelqu'un modifiait le code et que le programme ne fonctionnait plus, les tests automatisés informe de la régression)
- Améliorer la qualité de notre code, les tests sont des crash tests pour le code.

Les tests permettent de détecter les erreurs mais aussi de vérifier comment se comporte le programme sur une erreur volontaire.

Par exemple sur ce bout de code :

```
function HelloUser(user: string) {
    console.log(Hello ${user} !);
}
```

Comment doit réagir mon programme si je lui passe un number en argument, ou une chaine de caractère vide etc ?

Faire des tests allant a contre courant du fonctionnement normal du programme améliore la qualité de notre code.

Un des problèmes de la programmation impérative est de devoir mettre le programme dans un état particulier pour réaliser des test plus poussés, souvent l'effort nécessaire à la réalisation de ces tests ou la mise en place volontaire de bugs fait que l'on réalise souvent des "tests de complaisance" .

Avec la programmation fonctionnelle, le fait d'avoir des briques, il est bcp plus facile de tester en injectant des paramètres volontairement biaisés.

Un paradigme qui facilite la mise en place des tests est un paradigme qui facilite la création d'un code de qualité.

- **Les concepts**
  - Immutabilité

    pas de variables !

    ```
    var students = ['Jean'];
    students.push('Nadine'); // Mutation
    console.log(students); // retourne ['Jean', 'Nadine']
    ```

    ```
    const students = ['Jean'];
    const newStudents = [...students, 'Nadine'];
    console.log(newStudents); // retourne ['Jean', 'Nadine']
    ```

    1er cas : version avec mutation, on modifie notre tableau en rajoutant Nadine. on a pas gardé d'historique des évènements.

    2eme cas : On fait une constante students, on fait une nouvelle constante et on rajoute nadine. On prend le contenu du précédent tableau et on en crée un nouveau avec Nadine en plus. On limite les effets de bords en conservant les états antérieurs au prix d'une certaine lourdeur (mis en mémoire de plusieurs éléments).

  - La transparence référentielle

    Une expression est **référentiellement transparente** si elle peut être remplacée par sa valeur sans changer le comportement du programme. Théoriquement une fonction pure retourne toujours la même chose pour un paramètre donné, elle est transparente.

    La fonction pure permet d'atteindre la transparence référentielle.

  - 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.

    Exemples:

    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(); // ->5
add(); // ->6
```

- Fonctions d'ordre supérieur :

  Vérifie au moins une de ces affirmations :

  Prend une ou plusieurs fonctions en paramètre Retourne une fonction:

```
function fizz(x) { return x % 3 == 0 ? 'Fizz' : '';
}
function buzz(x) { return x % 5 == 0 ? 'Buzz' : '';
}
function fizzbuzz(x, fizz, buzz) { return fizz(x) == buzz(x) ? x :
fizz(x) + buzz(x);
}
console.log([12,15,4].map(x => fizzbuzz(x, fizz, buzz)));
```

  fizzbuzz est une fonction d'ordre supérieur, elle prendre 2 fonctions en paramètre (fizz et buzz).

  Cette composition de fonction nous permet de faire des choses plus complexes.

- L'évaluation paresseuse *(lazy evaluation)*

```
function fizz(x) { return x % 3 == 0 ? 'Fizz' : '';
}
function buzz(x) { return x % 5 == 0 ? 'Buzz' : '';
}
function fizzbuzzLazy(x, fizz, buzz) { return fizz(x) == '' ? x :
fizz(x) + buzz(x);
}
function fizzbuzzEager(x, fizz, buzz) { return fizz == '' ? x : fizz +
buzz;
}
// Fizz FizzBuzz ou nombre
console.log([12,15,4].map(x => fizzbuzzLazy(x, fizz, buzz)));
console.log([12,15,4].map(x => fizzbuzzEager(x, fizz(x), buzz(x))));
```

  dans `fizzbuzzLasy` : Si `fizz(x)` est nul ('') alors je retourne `x`, je n'évalue pas `buzz(x)`.

  dans `fizzbuzzEager` : j'évalue d'abord `fizz(x)` et `buzz(x)` et ensuite j'évalue si j'ai une chaine vide ou non.

Faire de l'évaluation paresseuse fait que l'on ne va évaluer une fonction uniquement que lorsque j'ai besoin de l'évaluer et pas en amont.