



FORMATION

JAVA

06/05/2024



m2iinformation.fr

Votre formateur

François Caremoli

Développeur pendant 7 ans sur des applications Java/JavaEE.
Profil full stack (du SQL ↔ HTML ou HTTP)

Lead Developer pendant 8 ans

Architecte SI pendant 2 ans.

Formateur en parallèle.

Et vous ?

- Présentation
 - Les langages et technologies que vous utilisez habituellement
 - Ce que vous connaissez de Java et ses bibliothèques
 - Ce que vous avez appris du cursus
 - Vos attentes vis-à-vis du cursus
 - Vos attentes vis-à-vis de la formation
- Prérequis
 - Droits administrateurs sur votre poste

Ouverture de Slack

- Slack est un outil de chat. Il est utilisé par de nombreux développeurs. Il sera utilisé pour servir d'espace de discussion, de partage de documents, et d'historique à la formation. Il peut être utilisé en tant qu'application téléchargée, ou sur le navigateur.
- Vous avez dû recevoir par mail un lien d'invitation à Slack
- Rejoignez Slack et réagissez au message de bienvenu.

Git (optionnel)

- Le lien du dépôt Git de la formation est sur Slack.
- Vous pourrez fork mon dépôt après la formation.
- Si vous souhaitez refaire l'application après la formation, vous aurez accès à tous les commit pour chaque étape de l'application.
- Un zip du projet sera mis sur Slack à la fin de la formation.

Supports de cours

- Tout est sur Slack.
- Le PPT du cours est épinglé en haut de la discussion.

Horaires, temps de pause et dossier pédagogique

- Les cours ont lieu de 9h à 17h30.
- Une pause de 15mn le matin et une pause de 15mn l'après midi.
- Midi, une pause déjeuner de 1h de 12h30 à 13h30 (attention !).

L'application "SimulationBateau" : fonctionnel

- Un petit jeu en mode texte avec des bateaux.
- Il permet d'utiliser les grands concepts vus pendant la formation.
- Le développement se fera étapes par étapes, via des exercices ou des TPs.
- Extensible pour permettre d'ajouter des fonctionnalités si le temps le permet.

Plan de cours

- Introduction
- Outils de développement
- Un premier programme Java
- Variables
- Classes & objets
- Classpath & Package
- Les classes de java.lang
- Structures de contrôle
- Héritage
- Interfaces
- Tableaux & Collections
- Enumerations
- Lambdas & Streams
- Exceptions



Introduction

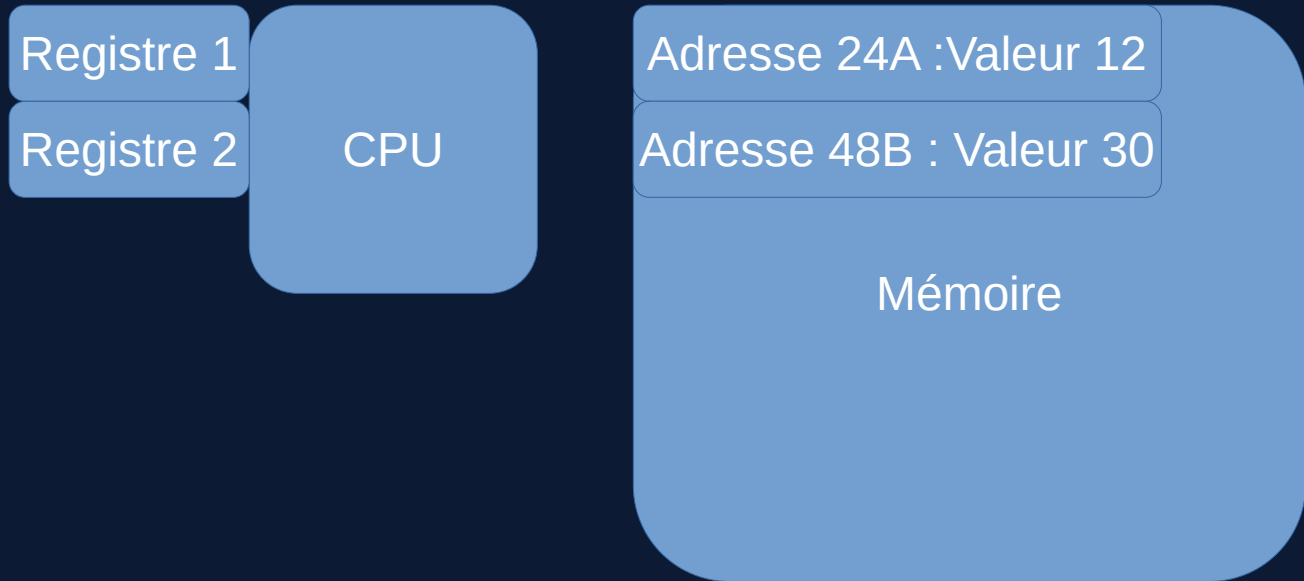
Introduction

Comment programmer un microprocesseur ? En assembleur ...

On veut additionner deux 'valeurs' en mémoire (aux adresses 24A et 48B) et remplacer la première valeur par le résultat de l'addition.

En pseudo code :

- Charger valeur à l'adresse 24A dans registre1
- Charger valeur à l'adresse 48B dans registre2
- Additionne Registre1 + Registre2 dans Registre1
- Charger Registre1 à l'adresse 24A



Comment programmer un microprocesseur ? En assembleur ...

Le pseudo code précédent doit être compilé par un compilateur dépendant du micro-processeur (et du système d'exploitation) pour donner un exécutable. L'exécutable est un fichier que l'on peut lancer (ou exécuter) uniquement sur le même système d'exploitation et le même micro-processeur (ou un micro-processeur utilisant le même jeu d'instructions).

Introduction

Le C

Le C est un langage de plus haut niveau que l'assembleur qui offre des fonctionnalités telles que la création de variables pour gérer la mémoire, les méthodes, les structures de contrôle :

```
| int valeur1 = 12; //On réserve une place en mémoire pour stocker un  
| entier, appelé valeur1  
| int valeur2 = 30; //De même pour valeur2  
| a = a + b; //On met dans a le résultat de l'addition
```

Il est quand même considéré comme un langage de bas niveau. Un code C peut généralement être compilé (avec le bon compilateur) sur des systèmes d'exploitation et des micro-processeurs différents. Il y a néanmoins des adaptations à faire dès que l'on utilise des routines propres au matériel.

La mémoire est gérée directement par le développeur qui alloue et désalloue la mémoire selon ses besoins (avec des problèmes si la mémoire est mal gérée).

C++ et la programmation orientée objets

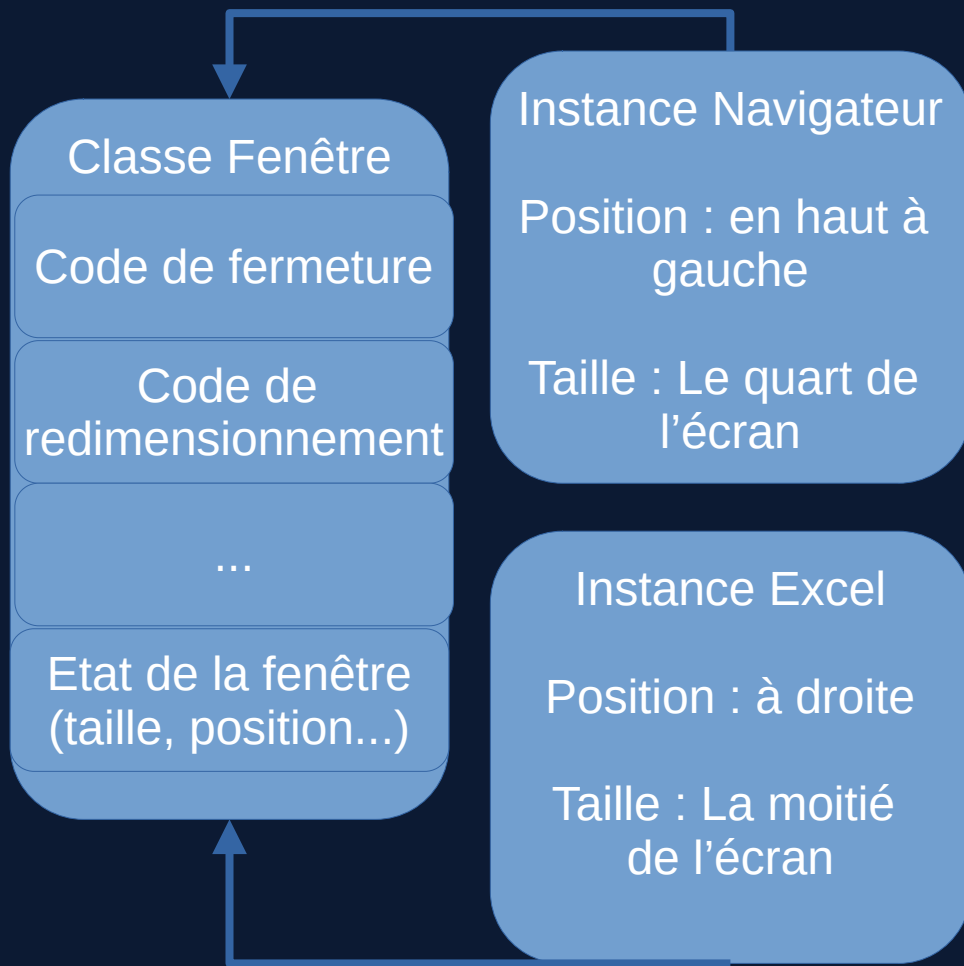
C++ est le successeur de C, qui apporte entre autres les fonctionnalités de la programmation orientée objets. La programmation objets permet de définir des classes : des 'moules' d'objets. A partir de ce moule, on peut créer plusieurs objets. La création d'un objet s'appelle 'instanciation'.

Si l'on simule le fonctionnement d'un système d'exploitation avec fenêtres, on peut imaginer qu'il contient une classe 'Fenêtre' qui crée un objet 'Fenêtre' à chaque fois que l'on double clique sur une icône du bureau.

Dans la classe se trouve le comportement de base de la fenêtre. Par exemple, le code qui stipule que "si on clique sur la croix en haut à droite, l'objet 'Fenêtre' (ou l'instance) se ferme". Ce code est mutualisé dans la classe 'Fenêtre'.

C++ et la programmation orientée objets

- Ci-contre la modélisation d'une classe Fenêtre et ses deux instances, créées lorsque l'on a double cliqué sur un Navigateur puis sur Excel.
- Les codes sont partagés mais chaque instance possède ses propres états, indépendants.
- Ceci permet de savoir où est le code qui permet de gérer les fenêtres (la maintenance est facilitée)
- Mais aussi de s'assurer que seule l'instance Excel peut modifier sa position et sa taille (en tirant sur les bords de la fenêtre par exemple)



La programmation orientée objets

- Les avantages de la programmation orientée objets sont grands pour structurer le code et contrôler comment les données sont lues et modifiées.
- Cette dernière faculté s'appelle 'encapsulation'.
- Un bon développeur doit, à partir d'un domaine fonctionnel ou de spécifications, savoir comment découper ce domaine en objets correctement encapsulés. Il peut s'aider de langages type UML.

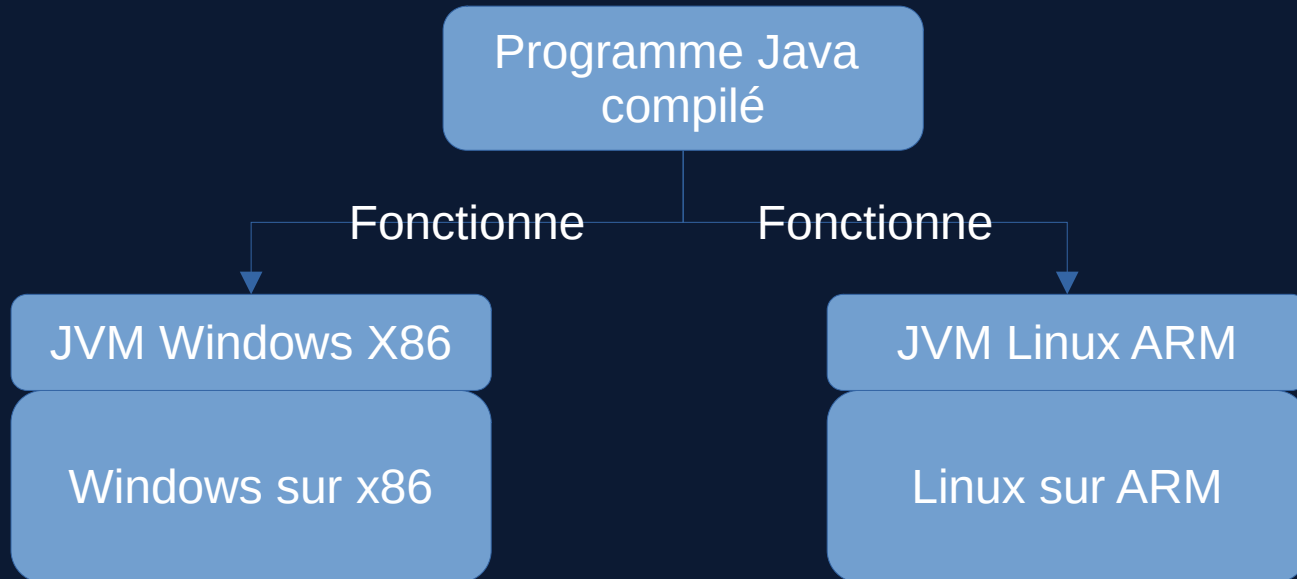
Java

Java est un des successeurs de C++, qui offre :

- Un langage de programmation
- Une machine virtuelle
- Un ensemble de classes disponibles pour offrir des fonctionnalités avancées aux développeurs
- La mémoire y est gérée par un ramasse miettes (ou Garbage Collector), et il se veut très objet (mais pas complètement).
- La machine virtuelle sert à découpler le code compilé de la plate-forme (Système d'exploitation et micro-processeur)

Java

La machine virtuelle sert à découpler le code compilé de la plate-forme (Système d'exploitation et micro-processeur). Un programme compilé en Java est censé tourné n'importe où, pourvu que le système possède une machine virtuelle Java.



Le programme Java compilé sera interprété par les machines virtuelles, donc en théorie plus lent mais ... une compilation à la minute permet de rendre ce code finalement compilé sur l'environnement cible.



Outils de développement

JDK

- Un Java Development Kit (kit de développement Java ou JDK) comprend :
 - Une Machine virtuelle Java, lancée via l'exécutable java.exe (ou java).
 - Un compilateur, lancé via l'exécutable javac.exe (ou javac)
 - Un débbugger, jdb
 - Un archiveur de classes : jar
 - Et d'autres outils ...
- De nombreuses JDK sont disponibles en téléchargement. La version d'OpenJDK (Open Source et libre d'usage) sera utilisée pour cette formation.
- Télécharger la JDK version 19 sur : <https://jdk.java.net/19/>
- Prendre la version correspondant au système d'exploitation actuel.
- L'installer et la dézipper dans un répertoire dédié au développement

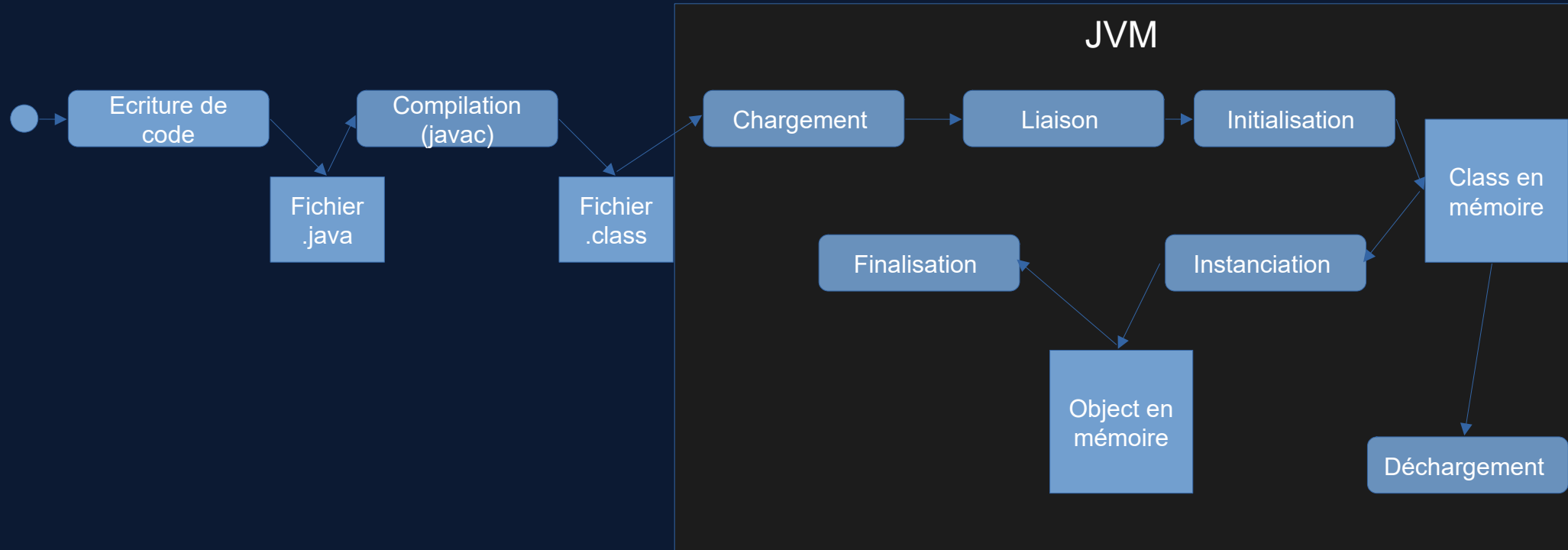
JDK : installation

- Modifier les paramètres du système d'exploitation pour y ajouter les exécutables du JDK dans le path.
- Sous Windows, ouvrir les paramètres (touche windows, taper paramètres et sélectionner Paramètres dans la barre d'outils)
- Cliquer à droite sur "Paramètres avancés du système"
- Cliquer en bas à droite sur "Variables d'environnement"
- Dans "Variables système", ajouter une variable appelée "JAVA_HOME" et dont la valeur vaut le répertoire dans lequel vous avez installé la JVM (ex : "C:\dev\jdk-17.0.1").
- Modifier la variable "Path" pour y ajouter "%JAVA_HOME%\bin\".
- Valider et fermer.
- Lancer une invite de commande (cmd) et vérifier que le lancement de java --version affiche la version du JDK installé

Javac

- Le compilateur Java (appelé javac) transforme les fichiers de classe Java (.java) en classes écrites en bytecode (.class).
- Le bytecode est un langage intermédiaire, interprété non pas par un microprocesseur, mais par une JVM (Java Virtual Machine). Ceci le rend indépendant de la machine allant l'exécuter (d'où le Write Once, Run Everywhere).
- Lors de la compilation du code source en bytecode, le compilateur effectue de nombreuses vérifications (notamment sur la syntaxe) pour garantir que le bytecode produit est valide et qu'il ne risque pas de nuire à la JVM qui va l'exécuter. Vous le voyez très vite dans votre IDE.

Cycle de vie d'une classe/objet



Exercice : compilation et lancement d'une classe

- Ouvrir une invite de commande (type cmd).
- Se placer dans le répertoire src du projet. Ce répertoire contient un fichier Java : MaPremiereClasse.java . Ce fichier contient un code tout simple qui affiche "Salut !".
- Lancer "javac MaPremiereClasse.java" pour compiler.
- Un fichier PremiereClasse.class apparaît. On peut l'afficher avec dir ou ls
- Lancer "java PremiereClasse" et observer le résultat.

Utilité d'un IDE

- Si on veut modifier le fichier .java et voir la modification à l'exécution, il faut modifier le fichier, le sauvegarder, le compiler, et lancer la machine virtuelle.
- Le cycle "développement -> compilation -> exécution" est extrêmement fréquent pour un développeur aujourd'hui, et doit donc être rapide et sans effort.
- Quasiment aucun développeur ne lance aujourd'hui javac (ni même java) directement. Un IDE fait beaucoup mieux ce travail.
- Un Environnement de Développement Intégré (ou IDE en anglais) va (entre autres)
 - Déclencher une compilation d'un fichier source en classe Java dès la modification ou la sauvegarde de ce dernier.
 - Lancer Java avec les bons paramètres pour exécuter un programme (même constitué de milliers de classes).
 - Offrir des outils comme la coloration syntaxique ou l'aide à l'écriture.

Installation d'un IDE (Eclipse)

- Télécharger L'IDE Eclipse à partir de : <https://eclipseide.org/release/>
- L'installer (si possible dans le répertoire de développement).
- Utiliser la version de base
- Le lancer
- Dans les paramètres, utiliser la JDK précédemment installée
- Importer un nouveau projet :
- Aller sur le fichier .. et cliquer sur exécuter . La console en bas affiche le même message que précédemment : notre IDE a lancé javac et java de façon transparente pour l'utilisateur.

Les conseils du formateur

- Il existe d'autres JDKs qui offrent des services différents. Généralement le choix d'une JDK ou d'une autre se fait au niveau d'une entreprise, afin que les développeurs et les administrateurs systèmes travaillent tous avec la même JDK (pour éviter les ennuis dus à des JDKs différentes).
- Il existe une infinité de manière de coder en Java. On peut utiliser des éditeurs de texte (Vim), ou des IDEs. Les IDEs en vogue actuellement sont :
 - Eclipse : pas le meilleur, mais gratuit, OpenSource et très répandu. Son design commence à dater un peu ...
 - NetBeans : équivalent à Eclipse.
 - IntelliJ : sans doute le meilleur, mais payant. Si votre entreprise a les licences, il est à utiliser.
 - Visual Studio Code : le petit dernier. Gratuit, basé sur Electron et une foule de plugins, fonctionne bien et offre une IHM et une ergonomie moderne.
- Si vous avez du temps, essayez en plusieurs de cette liste (ou d'autres ...).



Un premier programme Java

Analyse du premier programme

- Ouvrir src/MaPremiereClasse.java . Identifier :
 - Les commentaires
 - Le package
 - La classe
 - La méthode main
 - Les System.out.println

Les commentaires

- Les commentaires sont des lignes de texte qui ne sont pas compilées. Quelque soit le contenu d'un commentaire, le programme fera exactement la même chose. Ils sont sous deux formes en Java :
 - Les commentaires en ligne : tout ce qui suit les caractères `//` et avant un saut de ligne est commenté.
 - Les blocs de commentaires : tout ce qui se trouve entre `/*` et `*/`
- La Javadoc est un commentaire particulier : il obéit à une structure définie et permet de documenter une classe et des méthodes. Des outils transforment les Javadocs en documentations lisibles sur un navigateur (par exemple). La Javadoc est un commentaire commençant par `/**` . Il est recommandé de faire une Javadoc pour toute méthode ou classe ayant un rôle fonctionnel.

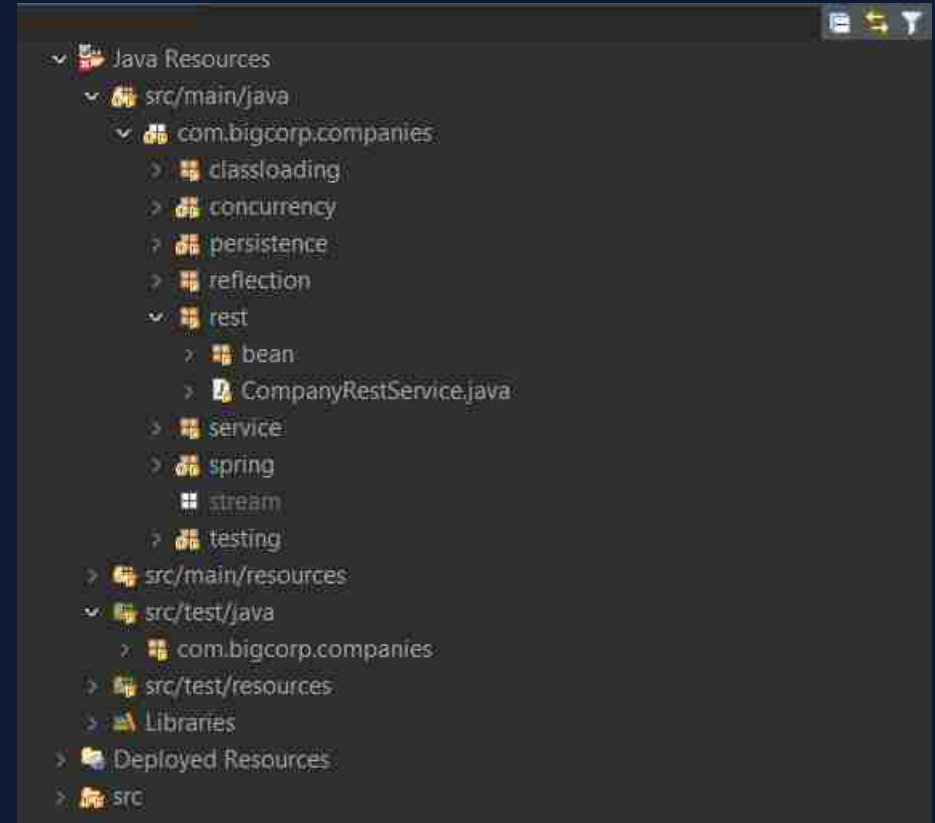
Le package

- Les fichiers sources peuvent être organisés en packages. Les packages définissent une hiérarchie de noms, chaque nom étant séparé par le caractère point. Le nom d'un package est lié au sous-répertoire du même nom que le package : ex : les fichiers sources du package `com.bigcorp.formation.cours` se trouvent dans `"com/bigcorp/formation/cours"`.
- Ceci permet de structurer les sources d'une application car une application peut rapidement contenir plusieurs centaines voire milliers de fichiers source. Les packages permettent aussi d'assurer l'unicité d'une classe grâce à son nom pleinement qualifié (nom du package suivi du caractère «.» suivi du nom de la classe).
- Par exemple , l'API Java est organisée en packages répartis en trois grands ensembles :
 - Packages standards : ce sont les sous-packages du package `java`
 - Packages d'extensions : ce sont les sous-packages du package `javax`
 - Packages tiers : ces packages concernant notamment Corba et XML

Exemple de découpage en packages

A droite, les packages de `com.bigcorp.companies` (lui-même un package) :

- `persistence` : permet de gérer la sauvegarde des objets en base de données
- `rest` permet de présenter les objets sous forme de WebServices REST
- `concurrency` gère les accès multithreads



La classe

- Un fichier .java contient au moins une classe publique ayant le même nom que le fichier.
- Il peut donc contenir d'autres classes, qui ne seront pas publiques, et ayant un nom différent.
- En s'aidant des classes et des packages, on peut "découper" son code de façon à le rendre plus facilement compréhensible, maintenable et réutilisable.

Premier programme Java

La méthode main

- Si on lance le programme java avec les paramètres suivants :

```
java MaPremiereClasse param1 param2
```

- La JVM va exécuter le code qui se trouve dans la méthode main de la classe MaPremiereClasse. La définition des méthodes viendra après, mais, il est possible de résumer une méthode à un "bloc de code".

System.out.println

- Cette méthode sert à afficher une ligne de code. Les méthodes seront vues plus tard, mais la fonctionnalité de

```
System.out.println("Coucou");
```

peut être résumée en : Affiche "Coucou" dans la console, et ajoute un retour chariot.

Premier programme Java

Instruction

- L'instruction suivante :

```
System.out.println("Coucou");
```

exécute quelque chose. En Java, toute instruction se termine par le caractère ";". Tant que ce caractère n'est pas atteint, l'instruction n'est pas terminée. Ceci permet d'écrire des instructions sur plusieurs "lignes de texte".

- L'instruction suivante est équivalente celle au dessus :

```
System.  
out.  
println(  
    "Coucou"  
)  
;
```



Variables

Déclaration, assignation

- On déclare une variable avec la syntaxe suivante. (Ceci réserve de l'espace en mémoire pour la variable) :

```
typeVariable nouvelleVariableDeclaree;
```

- On assigne une valeur à une variable avec la syntaxe suivante :

```
nouvelleVariableDeclaree = 4;
```

- On peut combiner les deux en une seule instruction:

```
typeVariable nouvelleVariableDeclaree = 4;
```

Variables

Portée

En Java, la portée de la variable est définie par :

- sa visibilité si c'est un attribut de classe (sera vu dans le chapitre sur les classes)
- le bloc de code dans lequel elle est pour un argument de méthode, ou une variable locale (déclarée dans un bloc de code).

Dans le cas d'une variable locale, la variable peut être utilisée à partir du moment où elle est définie, jusqu'à la fin du bloc de code dans lequel elle a été instanciée.

Un bloc de code est une suite d'instructions entourée par des accolades.

Les type des base

- On peut manipuler différents types en Java. Ces types sont :
 - short , int et long: des nombres entiers ayant des tailles différentes (signés) (0 par défaut)
 - float et double : des nombres à virgule (ou flottants) ayant aussi des tailles différentes (signés) (0 par défaut)
 - boolean : une valeur booléenne, donc true ou false (vraie ou fausse). (false par défaut)
 - char : un caractère codé en UTF-16 (\u0000 : caractère null par défaut,)
 - byte : un octet (signé) (0 par défaut)
 - Classe : toute instance d'une classe (vu ensuite). (null par défaut)

Nombres et type

- Par défaut, tout nombre à virgule sera un double. Pour le rendre float, il faudra suffixer sa valeur avec f:

```
double monDouble = 3.14;  
float monFloat = 3.14f;
```

- On peut aussi définir des valeurs avec des modificateurs :

```
int miyyaard = 1_000_000_000;  
int hexa = 0xff;  
int binaire = 0b11111111;
```

Les type des base

Type	Désignation	Longueur (!= taille)	Valeurs
boolean	valeur logique : true ou false	1 bit	true ou false
byte	octet signé	8 bits	-128 à 127
short	entier court signé	16 bits	-32768 à 32767
char	caractère Unicode	16 bits	\u0000 à \uFFFF
int	entier signé	32 bits	-2147483648 à 2147483647
float	virgule flottante simple précision (IEEE754)	32 bits	1.401e-045 à 3.40282e+038
double	virgule flottante double précision (IEEE754)	64 bits	2.22507e-308 à 1.79769e+308
long	entier long	64 bits	-9223372036854775808 à 9223372036854775807

Utilité des variables

- Une variable permet d'identifier une valeur avec un nom. Elle peut s'écrire qu'avec les caractères de l'alphabet en minuscule, majuscule, les chiffres et les caractères « _ » et « \$ ».
- Elle ne peut pas commencer par un chiffre.
- Par convention, on écrit les variables en camel case, en commençant par une majuscule : ex `maVariableBienNommee`.
- Remarque : par convention, les constantes s'écrivent en majuscule uniquement. Exemple `MA_CONSTANTE_BIEN_NOMMEE`.
- Utilité des variables :
 - La déclaration d'une variable sert à créer une référence dans un emplacement en mémoire.
 - L'affectation d'une variable sert à lui associer une valeur.

Opérateurs

- Les opérateurs permettent de manipuler ou comparer des valeurs, notamment des variables. En Java, sont disponibles :
- l'opérateur d'affectation `=` : ex `int a = 3; int b = c;`
- Les opérateurs arithmétiques : `+` `-` `*` `/` `%` `^` : ex `int a = 4 * b;`
- Les opérateurs binaire de comparaison : `==` `!=` `>` `<` `>=` `<=` ex `boolean b = 3 < 4`
- Les opérateurs logiques `&` `&&` `|` `||` : ex `boolean b = 3 < 4 && 3 < 2`
- L'opérateur de négation `!` : ex `boolean b = !true;`
- Les opérateurs unitaires : `--` et `++` : ex `i--`; (post décrémentation) `++a` (préincrémentation)
- Les opérateurs avec assignation `+=` `-=` `*=` `/=` `%=` : ex `a+=3;` (éq. à `a = a + 3`)
- D'autres (décalage de bits, etc ...)

Opérateurs de comparaison

Opérateur	Exemple	Signification
>	a > 10	strictement supérieur
<	a < 10	strictement inférieur
>=	a >= 10	supérieur ou égal
<=	a <= 10	inférieur ou égal
Err :520	a == 10	Egalité
!=	a != 10	diffèrent de
&	a & b	ET binaire
^	a ^ b	OU exclusif binaire
	a b	OU binaire
&&	a && b	ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse
	a b	OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie

L'opérateur ternaire

Opérateur	Exemple	Signification
? :	a ? b : c	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type

Priorités des opérateurs (par ordre décroissant)

les opérateurs postfixe	expr++ expr--
les opérateurs unaires	++expr --expr +expr -expr ~ !
multiplication, division et modulo	* / %
les opérateurs d'addition et soustraction	+ -
les opérateurs de décalage	<< >> >>>
les opérateurs de comparaison	< > <= >= instanceof
les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	
l'opérateur ET logique	&&
l'opérateur OU logique	
l'opérateur ternaire	? :
les opérateurs d'assignement	= += -= *= /= %= ^= ;= ...
l'opérateur arrow	->

Exercice

- Compléter l'exercice : ExerciceVariables

Les conversions

- L'opérateur d'affectation = permet de changer le type d'une variable vers un autre type.
- Si aucune information n'est perdue pendant la conversion, rien n'est nécessaire :
 - Par exemple en convertissant un float en double, aucune information n'est perdue parce que double est un nombre flottant (comme float) mais avec une plus grande capacité :
`double monDouble = monFloat;`
- Par contre, si des informations peuvent être perdues pendant la conversion, il faut expliciter au compilateur que l'on peut prendre le risque en faisant un "cast". Un cast explicite le type vers lequel on veut convertir :
 - `int monInt = (int) monDouble;` Ici, on perd toute la partie après la virgule du double, et peut être des nombres significatifs si le double est plus grand que la valeur maximale de monInt.
- Si le cast est impossible, une erreur de compilation apparaît (ex conversion d'un int en boolean). Java est un langage fortement typé qui ne tolère pas qu'on tente de mettre le mauvais type dans une variable.

Les mots réservés

- Certains mots, qui ont un sens en Java : int , new, double, package, class .. ne peuvent être utilisés pour nommer les variables définies par le développeur (ni même les classes ou les méthodes ...).
- Ce sont des mots réservés.
- Le compilateur empêchera l'utilisation des mots réservés, mais la liste complète est ici :
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Exercice

- Compléter l'exercice : ExerciceConversions

Ce qu'il faut retenir

- Java permet de stocker des valeurs dans des variables.
- On définit une variable avec la syntaxe `typeVariable nomVariable`.
- L'opérateur `=` permet d'affecter une valeur à une variable.
- Les types de variables sont nombreux et offrent des tailles différentes.
- Il existe des mots réservés : ils ne peuvent être utilisés pour nommer une variable, une méthode, une classe ...

Les conseils du formateur

- Se référer aux types de variables et connaître leur limite est une bonne chose.
- Privilégier les opérateurs `&&` et `||` (plutôt que `&` et `|`) pour éviter d'évaluer des conditions alors que c'est inutile.
- Il est plus efficace en général d'utiliser `a+=4` que `a = a + 4`, privilégier cette syntaxe (même si les compilateurs modernes optimisent la dernière syntaxe).



Classes et objets

Définition

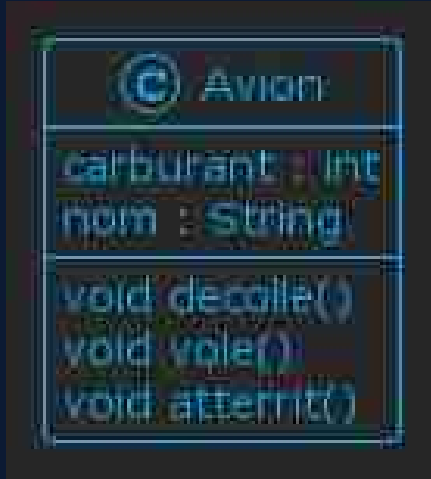
- Comme vu précédemment, la programmation orientée objets vise à grouper le code des applications en classes, afin de favoriser :
 - Le contrôle des données.
 - Le regroupement du code par thème.
 - La réutilisation du code.
- Pour ce faire, le code est regroupé en classes.

Contenu d'une classe

- Une classe est définie par un nom et un package.
- Elle contient :
 - des attributs : des variables typées. Chaque instance de la classe possède et modifie SES propres attributs.
 - des méthodes : une suite d'instructions.

Modélisation

- Soit une classe Avion , avec ses attributs et ses méthodes



- Une fois la classe codée en Java, il est simple de créer des objets de type Avion (des instances d'Avion) à partir de cette classe. Chaque objet aura son propre nom et son propre carburant, et les méthodes decolle() vole() et atterrit() pourront être appelées par l'avion ou d'autres classes.

Déclaration, instantiation, assignation

- Un objet est une instance d'une classe. On déclare une variable d'un objet avec la syntaxe suivante (de la même façon que n'importe quel type)

```
NomClasse nomNouvelleVariable;
```

- On crée une instance d'une classe avec la syntaxe suivante :

```
nomNouvelleVariable = new NomClasse();
```

- On peut combiner les deux en une seule instruction:

```
NomClasse nomNouvelleVariable = new NomClasse();
```

Instantiation

- Le mot clé 'new' crée une instance d'une classe : le 'moule' classe crée un 'objet'.
- A l'instant de l'appel à new, l'espace mémoire nécessaire pour stocker toutes les données de l'instance est réservé.
- Ce qui suit le new s'appelle le constructeur. Il peut contenir des paramètres (ex : new String("coucou")) ou rien (ex : new Object()).
- Une variable d'objet définie mais non instancié vaut 'null'.
- On peut aussi affecter null directement à une variable :

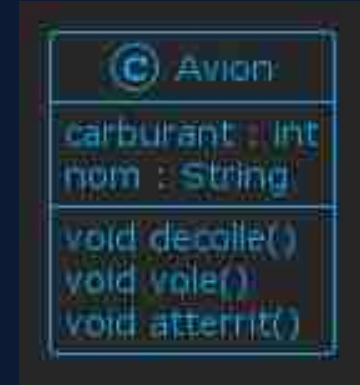
```
NomClasse nomNouvelleVariable = null;
```

Exercice

- Compléter l'exercice : ExerciceInstanciations

Classes et objets

Encapsulation



- En reprenant la classe avion, il s'avère qu'il vaudrait mieux contrôler l'accès aux variables "carburant" et "nom"
Seules les instances d'Avion devraient pouvoir écrire ces données. Ceci est possible en définissant des portées de variables.
- Ci-dessous la correspondance entre la portée d'une variable et les entités qui peuvent y accéder, en lecture ET écriture :
 - public : les instances de n'importe quelle autre classe.
 - protected : les instances des classes du même package, et les classes filles (vues plus tard).
 - package : les instances des classes du même package.
 - private : uniquement les instances de la classe.

Une première classe

- La classe ci-dessous contient quatre variables avec des visibilités différentes :

```
package monpackage;  
  
public class Avion{ //<-- Déclaration d'une classe  
  
    private int carburant; // <-- Déclaration des variables  
    protected String nom;  
    double cap;  
    public String nomCompagnie;  
  
}
```

Méthodes

- Une méthode est un ensemble d'instructions que l'on peut appeler, en fournissant 0, 1 ou n paramètres, et qui renvoie un type (ou rien). Elles permettent de regrouper du code, ou algorithmes, dans un bloc nommé.
- Une méthode appartient à une classe et dispose aussi des portées vues précédemment.
- Une méthode s'écrit ainsi :

```
| visibilité typeRetour nomMethode(TypeParametre1 parametre1){  
|     instructions Java ici;  
| }
```

Exemple de méthodes

```
public class Avion{ //<-- Déclaration d'une classe
    ... // <-- Déclaration des variables

    public void decolle(){
        System.out.println("Je décolle!");
    }

    public int afficheCarburantRestant(){
        return carburant;
    }

    public void vole(int nombreHeuresVol){
        int consommation = 40 * nombreHeuresVol;
        carburant -= consommation;
    }
}
```


Exercice

- Créer une classe Bateau dans le package simulateur.bateau . (donc un fichier Bateau.java dans le répertoire simulateur/bateau du projet).
- Cette classe correspond au diagramme UML suivant (le carré bleu signifie private, le rond vert public) :



- Ajouter les attributs et les méthodes nécessaires. Les méthodes affichent juste une chaîne de caractère avec `System.out.println()` pour le moment.

this

- Le mot clé this est une référence vers l'instance de la classe.
- Il peut servir pour discriminer une variable.

```
public class SuperClasse{  
    private int longueur;  
    public boolean plusGrandQue(int longueur){  
        return this.longueur > longueur; //Ici la JVM sait que la première  
        //variable est celle de this, la seconde est le paramètre de la méthode  
    }  
}
```

Encapsulation avec des méthodes

- En reprenant la classe avion, des attributs ne sont plus modifiables par des objets qui ne sont pas du type "Avion". Pourtant, il est parfois désirable que ceux-ci le fassent.
- Une pratique extrêmement courante est de mettre le maximum d'attributs de classe en private. Puis de créer des méthodes "accesseurs" pour autoriser l'accès à ces attributs.
- Un accesseur peut être :
 - Une méthode get . Par exemple `public int getCarburant(){return this.carburant;}` pour accéder en lecture à l'attribut carburant.
 - Une méthode set . Par exemple `public void setCarburant(int carburant){this.carburant=carburant;}`, pour accéder en écriture à l'attribut carburant.
- Si l'on veut tout permettre, on peut générer des méthodes set et get pour chaque attribut. Notre classe est alors un POJO (ou Plain Old Java Object), ou JavaBean : une classe qui permet de transporter des données. Ce genre de classes est très utilisé dans les frameworks transférant des données d'un système à un autre (vers une base de données, des WebServices).
- Si l'on ne laisse que les getter, et que les attributs ne changent jamais, notre classe génère des objets immutables. Ils peuvent être intéressants dans des contextes de multithread, où les changements de données amènent des comportements étranges.

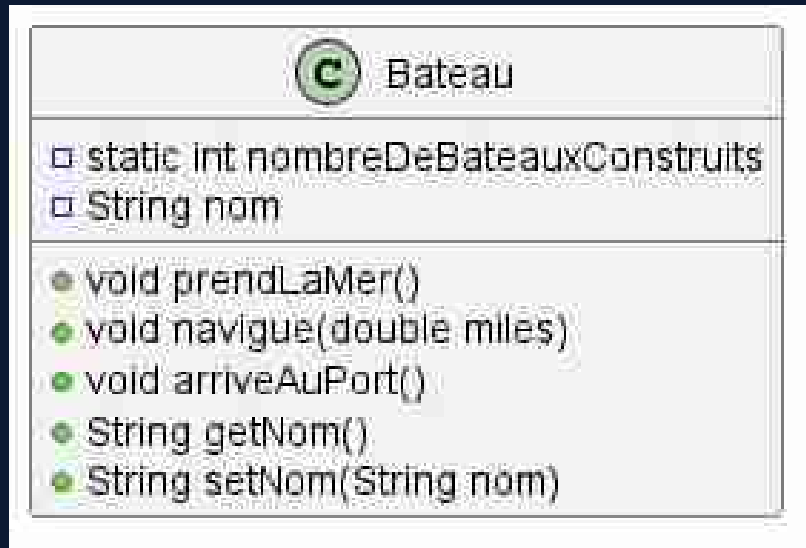
static

- Le mot clé static est une référence vers la classe ET NON l'instance.
- Avec static on peut déclarer des membres (attributs et méthodes) qui ne dépendent pas d'une instance, mais d'une classe.
- Dans le cas d'attributs, ces attributs seront partagés par toutes les instances.

```
public class Voiture {  
    private static int nombreDeVoituresProduites;  
}
```

Exercice

- Dans la classe Bateau précédente, ajouter l'attribut static `nombreDeBateauxConstruits` et les méthodes `getXXX` ajoutées ci-dessous (utiliser `this` et `static`)



final

- Le mot clé final permet de spécifier qu'une variable ne changera jamais de valeur. Elle sera constante.

```
public class PieceMecanique {  
    private final int numeroDeSerie;  
}
```

Constructeur d'objets

- Un constructeur est une méthode spéciale d'une classe. On définit une visibilité (souvent public), mais elle n'a pas de type de retour, et son nom est exactement le même que celui de la classe. C'est la méthode qui sera appelée quand on va instancier un objet avec new.
- Si aucun constructeur n'est présent dans une classe, le compilateur en crée un de façon transparente : le constructeur par défaut, qui est sans argument.
- Dès lors que l'on crée un constructeur, le constructeur par défaut n'est plus créé par le compilateur.
- Un constructeur peut en appeler un autre en utilisant le mot clé `this(...)`. Ceci doit être la première instruction du constructeur.

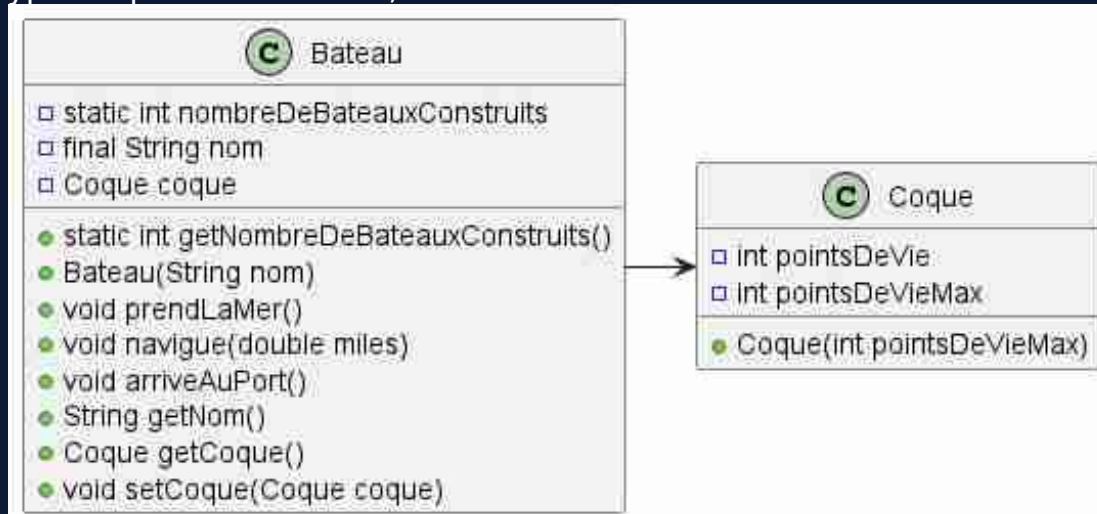
Exemples de constructeurs

```
public class Avion {  
    private int carburant;  
  
    public Avion(){  
        //équivalent à : return this;  
    }  
  
    public Avion(int carburant){  
        this.carburant = carburant;  
    }  
}  
  
// On peut alors appeler (ailleurs dans notre programme) :  
Avion monAvionDeBase = new Avion();  
Avion monAvionPlein = new Avion(780_000);
```


Classes et objets

TP

- Rajouter un constructeur à Bateau : ce constructeur incrémente nombreDeBateauxConstruits à chaque fois qu'il est appelé. De plus, ce constructeur prend une chaîne de caractères en paramètres pour la mettre dans le nom. L'attribut nom devient final et le setter doit disparaître. (le nom devient immutable).
- Rajouter une méthode public static void main(String[] args) dans Bateau.
- Dans cette méthode, instancier deux ou trois Bateaux, et afficher via System.out.println le nombre de bateaux construits .
- Créer une classe Coque, dans le même package que Bateau. Cette classe contient deux attributs de type int : pointsDeVie et pointsDeVieMax. Le constructeur de Coque prend en argument un int qui sera attribué à pointsDeVie et pointsDeVieMax.
- Créer un attribut de type Coque dans Bateau, avec les accesseurs associés.



- Créer une classe Coque, dans le même package que Bateau. Cette classe contient deux attributs de type int : pointsDeVie et pointsDeVieMax. Le constructeur de Coque prend en argument un int qui sera attribué à pointsDeVie et pointsDeVieMax.
- Créer un attribut de type Coque dans Bateau, avec les accesseurs associés.
- Dans le main, à la création des bateaux, créer une coque et la rattacher au bateau, puis afficher les points de vie de la coque.

La destruction d'un objet

- new réserve de l'espace mémoire. Or cet espace est limité sur tout ordinateur.
- Contrairement à C, ce n'est pas le développeur qui va taper un code qui libère la mémoire : le programme va s'en charger.
- Une machine virtuelle Java (JVM) contient un processus qui scrute périodiquement tous les objets. Tout objet qui n'est plus référencé par une variable est éligible à la destruction, et à la libération de l'espace mémoire. Cet espace pourra être ré-utilisé par d'autres objets.
- Ce processus s'appelle ramasse-miettes ou Garbage Collector.

Ce qu'il faut retenir

- La syntaxe `NomClasse maVariable = new NomClasse();` ou `NomClasse maVariable = new NomClasse(param1, param2...);` est à connaître sur le bout des doigts.
- Le fait qu'une variable de type Classe soit null à l'instanciation doit être toujours pris en compte. Le cas null n'est pas forcément celui auquel on pense quand on voit du code et pourtant, il amène de nombreuses erreurs.
- La JVM gère la mémoire au fur et à mesure des appels à `new` et des fins de vie des instances.
- Construire une classe avec des attributs et des méthodes, chacun ayant une visibilité propre est assez complexe, mais aide à structurer et maintenir le code.

Les conseils du formateur

- Pour générer des classes, des getters, des setters, des constructeurs, utilisez au maximum les fonctionnalités de l'IDE.
- De base, mettez tous les attributs en private et augmentez la visibilité des attributs en fonction des besoins. Ceci évitera que des utilisateurs de vos classes n'altèrent les données des classes que vous avez développées.
- Les méthodes aussi doivent avoir la plus petite visibilité possible : n'exposez (en public) que les méthodes utiles aux autres utilisateurs.
- Si vous n'arrivez pas à organiser votre code en classes, écartez vous de l'écran et du clavier, prenez un crayon et un cahier, et essayez de modéliser votre application en UML (ou toute autre méthodologie).
- Un bon concept pour découper correctement un modèle est : "Une classe, une responsabilité". Si votre classe a plusieurs responsabilités, il vaut mieux la découper. Exemple une classe qui gère les accès au réseau et les accès disques devrait être découpée en deux.



Classpath & Package

Utilisation du package

- Une classe est définie par un nom et un package.
- Si un package monpackage contient deux classes A et B. A peut faire des références à la classe B de façon implicite.
- Si la classe A du package monpackage veut utiliser la classe B, elle va devoir l'importer. L'import se fait grâce à l'instruction import. Cette instruction se trouve après la déclaration du package, mais avant la définition des classes.

```
package monpackage;  
  
import autrepackage.B;  
  
public class A {  
    private B instanceDeB;  
}
```

Utilité du package

- Le package permet de différencier de façon explicite deux classes portant le même nom.
- Si (par malheur) la classe A a besoin de deux classes B avec le même nom, mais des packages différents, elle peut importer l'un et explicitement définir l'autre quand elle utilise son type :

```
package monpackage;  
  
import autrepackage.B;  
  
public class A {  
    private B instanceDeB;  
    private encoreunautrepackage.B instanceDeB;  
}
```


Le package en pratique (1/2)

- On peut ne pas utiliser de directive import et expliciter tous les noms de classes (ou de types) mais cela alourdit le code.
- Tout IDE permet d'automatiser la gestion des imports avec des fonctionnalités comme "source" -> "Organize imports" sous Eclipse, (shift+ctrl+o) ou le fait de lancer un "Organize imports" à chaque sauvegarde de classe.
- On peut importer toutes les classes d'un package avec import monpackage.* . Toutes les classes du package java.lang (comme String) sont implicitement importées.
- On peut aussi importer des attributs statiques d'une classe avec import static.
- Par convention, les noms de package sont en minuscule et au singulier.

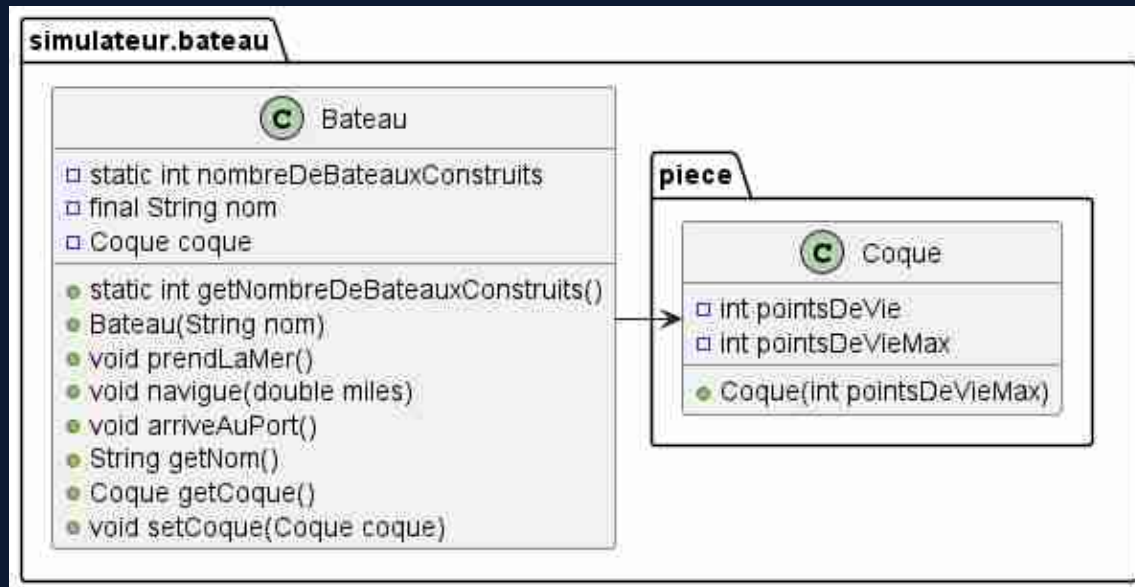
Le package en pratique (2/2)

- Pour avoir des noms de package uniques, les noms des packages professionnels sont formatés comme une URL inversée avec d'abord des informations sur l'entreprise, puis sur le projet, puis sur le rôle du package dans le projet : ex `fr.edf.fusion.model.reacteur.InjecteurHydrogene` :
 - `fr.edf` est le nom de l'entreprise
 - `fusion` le nom du projet
 - `model.reacteur` un regroupement de classes, concernant le réacteur dans le modèle de connées.
- Cela évite les collisions de nom de package entre projets et entre entreprises. Un projet peut contenir des milliers de classe et une entreprise peut gérer des centaines de projets ...

Classpath & Package

Exercice

- Avec l'IDE, dans la vue "Workspace" créer un nouveau package simulateur.bateau.piece (avec un clic droit sur le package simulateur.bateau, par exemple).
- Déplacer Coque.java dans ce package, l'IDE propose une refonte ou refactoring. Accepter et voir le résultat dans la classe Moteur et Bateau.
- Bonus : créer une classe Moteur dans le package piece et le rattacher au bateau. (Moteur contient deux attributs final et int : puissance et consommation.

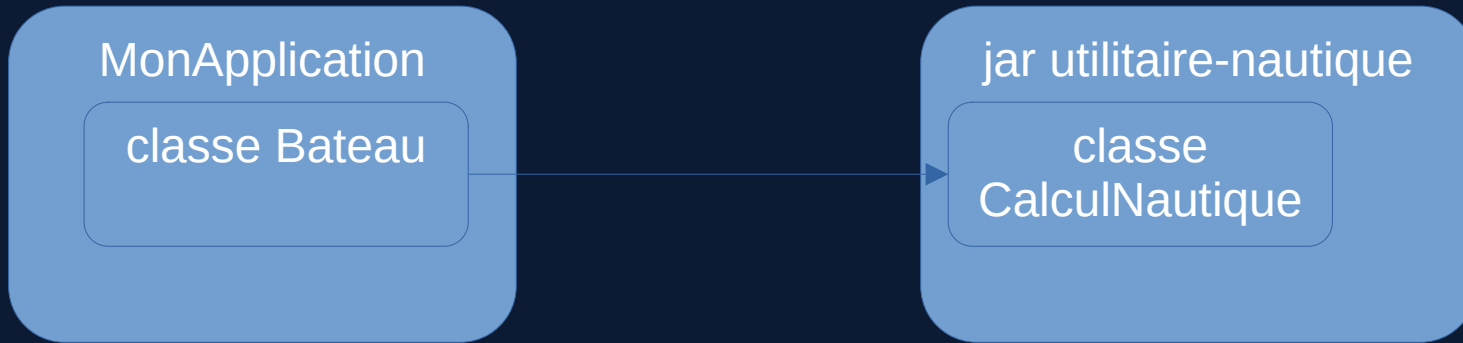


Le classpath

- Quelles classes peut-on utiliser dans un projet ?
 - Celles fournies par java de base (java.lang.String par exemple)
 - Celles fournies par les extensions de java (ex : javax.xml)
 - Celles ajoutées dans le classpath
- Le classpath permet de préciser au compilateur et à la JVM où se trouvent les classes nécessaires à la compilation et l'exécution d'une application.
- Le classpath est constitué de chemins vers des répertoires et/ou des archives sous la forme de fichiers .jar ou .zip. Chaque élément du classpath peut donc être :
 - Pour des fichiers .class : le répertoire qui contient l'arborescence des sous-répertoires des packages ou les fichiers .class (si ceux-ci sont dans le package par défaut)
 - Pour des fichiers .jar ou .zip : le chemin vers chacun des fichiers
 - Le répertoire à partir duquel on lance javac ou java est dans le classpath

Usage du classpath

- Si notre projet a besoin de classes contenues dans un JAR (un zip contenant des classes). Il faut ajouter à la compilation et à l'exécution ce JAR dans le classpath.



```
javac -cp utilitaire-nautique.jar simulateur.bateau.Bateau.java  
java -cp utilitaire-nautique.jar simulateur.bateau.Bateau.java
```

Usage du classpath

- On peut ajouter les différentes ressources du classpath en les séparant par un ":" sous Unix ou ";" sous Windows.

```
javac -cp utilitaire-nautique.jar;autre.jar simulateur.bateau.Bateau.java  
javac -cp utilitaire-nautique.jar:autre.jar simulateur.bateau.Bateau.java
```

- Si la même classe avec le même nom de package est présente dans plusieurs jars utilisés, celle du premier jar sera utilisée.
- Un JAR peut aussi contenir dans son fichier META-INF/MANIFEST.MF les classpath dont il a besoin (souvent sous la forme d'autres JARs), ainsi que la classe contenant la méthode main. Ainsi on peut déployer ce JAR et le lancer avec un simple : `java -jar nom-du-jar`

Avec un IDE et Maven

- L'utilisation du Classpath est incontournable pour développer et déployer une application utilisant des composants externes ...
- ... mais ceci est tellement fastidieux que presque plus aucun développeur ne le fait à la main.
- Les IDEs et des outils de build comme Maven permettent de définir à haut niveau des dépendances. L'outil de build télécharge ensuite ces dépendances, les ajoute dans le classpath à l'exécution, et même à la livraison.
- Néanmoins, en cas de souci, il est absolument nécessaire de comprendre comment le classpath fonctionne.

Ce qu'il faut retenir

- Un programme Java a besoin de nombreuses classes pour fonctionner.
- Il est recommandé de réutiliser les classes déjà existantes, développées par des collègues ou d'autres entreprises, via des JARs.
- Il faut définir au lancement de la JVM les JARs à utiliser, c'est ce à quoi sert le classpath : le classpath est une liste de JARs (ou de répertoires) qui contiennent les classes nécessaires à la compilation, ou l'exécution du projet.
- Aujourd'hui, les développeurs modifient rarement ce classpath à la main, mais il faut tout de même avoir conscience de ce mécanisme, en cas de problème.

Les conseils du formateur

- Si une `NoClassDefFoundError` (ou une `ClassNotFoundException`) survient, c'est généralement qu'une classe dont vous avez besoin n'est pas dans le classpath (le jar manque, ou ce n'est pas le bon, ou ce n'est pas la bonne version...)
- Les applications Web Java gèrent le classpath différemment des jars, il vaut mieux s'y intéresser quand on développe une application Java Web. Consulter la documentation du serveur Web (ou du conteneur de Servlets).
- Utilisez Maven (ou un autre outil de build).

<https://docs.oracle.com/javase/tutorial/essential/environment/paths.html>

<https://maven.apache.org/>



Les classes de java.lang

Les chaînes de caractères

- Le compilateur simplifie l'instanciation d'objets de type chaîne de caractères. On peut par exemple taper :

```
String salut = "Salut !!";
```

Ce qui va instancier une nouvelle chaîne de caractères. Une instance est bien créée, mais cela est 'caché' par le compilateur.

- Le compilateur permet d'utiliser l'opérateur + pour concaténer des chaînes de caractères.

```
String lesGens = "Les gens";  
String salutLesGens = salut + lesGens;  
salutLesGens += salut;
```

- Les exemples de cette page ont créé 4 instances de chaînes de caractères en mémoire

Les classes de java.lang

Exercice

- Compléter l'exercice : ExerciceChainesDeCaracteres

Les Wrappers de types primitifs

- Chaque type primitif de Java (int, double, boolean ...) a une classe "Wrapper" associée.

```
int monEntier = 4;  
Integer monEntierInstance = Integer.valueOf(4);
```

- Une instance de Integer peut :
 - être null
 - être intégrée dans des collections d'objets
 - ...
- Integer propose aussi des fonctions intéressantes

```
int fromString = Integer.parseInt("4");  
monEntierInstance.byteValue();
```

AutoBoxing / Unboxing

- Il est possible de convertir automatiquement des types primitifs en Wrappers (autoboxing) et vice-versa (unboxing).


```
int monEntierInstance = Integer.valueOf(4); //autoboxing  
int monEntier = monEntierInstance; //unboxing
```

- Le compilateur “retouche” le code afin de faciliter la tâche du développeur.
- Par contre, les types primitifs ne pouvant être null , tenter de faire un unboxing sur un null va renvoyer une erreur.

Les classes de java.lang

Exercice

- Compléter l'exercice : ExerciceWrappers



Structures de contrôle

Structures de contrôle

Un bloc de code

- Un bloc de code regroupe des instructions :

```
{  
  Des instructions à exécuter  
}
```

- Le bloc de code est encadré par des accolades.
- Les blocs de codes sont très utilisés pour les structures de contrôles, afin de grouper des instructions, pour les exécuter en boucle, ou optionnellement.

La boucle for

- Si l'on veut répéter des instructions, on peut utiliser la boucle for

```
for (instructionA ; conditionB; instructionC ) {  
    Des instructions à exécuter  
}
```

- L'instructionA est exécutée avant que la boucle ne commence
- La conditionB renvoie un booléen. Avant chaque itération, la conditionB est évaluée. Si instructionB == true, la boucle continue. Sinon, elle s'arrête.
- L'instructionC est exécutée à chaque fin de boucle
- La boucle exécute toutes les instructions du bloc de code.

La boucle for

- La boucle for suivante va afficher tous les nombres de 0 à 9

```
for (int i=0 ; i < 10; i++ ) {  
    System.out.println(i);  
}
```

- On peut ne rien mettre dans les instructions. Exemple : une boucle infinie :

```
for ( ; ; ) {  
    System.out.println("Coucou");  
}
```

La boucle while

- Si l'on veut répéter des instructions d'un bloc de code, on peut utiliser la boucle while

```
while (condition) {  
    des instructions ici  
}
```

- Tant que condition est vraie, la boucle s'exécute.

La boucle do while

- Si l'on veut répéter des instructions, on peut utiliser la boucle do while

```
do {  
    des instructions ici  
}while (condition)
```

- Tant que condition est vraie, la boucle s'exécute. La différence avec la boucle précédente est que la boucle est exécutée au moins une fois.

Instructions de contrôle

- L'instruction "break;" permet d'arrêter une boucle : le programme quitte alors cette boucle. Le programme ci-dessous n'affichera qu'une fois "coucou".

```
while(true){  
    System.out.println("coucou");  
    break;  
}
```

- L'instruction "continue;" permet de stopper une itération, mais tout en continuant la boucle. Le programme ci-dessous n'affichera jamais au revoir.

```
while(true){  
    System.out.println("coucou");  
    continue;  
    System.out.println("au revoir");  
}
```

Exercice

- Compléter l'exercice : ExerciceBoucles

If then else

- Si l'on veut exécuter des instructions de manière optionnelle, il faut utiliser la syntaxe avec if, then (optionnellement else).

```
if(condition){  
    System.out.println("ok");  
}else{  
    System.out.println("ko");  
}
```

- Le bloc de code juste après if est exécuté si condition est vraie.
- Le bloc de code juste après else est exécuté si condition est fausse.

If then else avec opérateur ternaire

- Si l'on récupérer une variable en une seule instruction, on peut utiliser l'opérateur ternaire condition?valeur1:valeur2 .

```
System.out.println(isOk?"true":"false");  
String valeurAAfficher = isOk?"true":"false";
```

- Ci-dessus, si isOk est vrai, on affiche "true". Du plus, la variable valeurAAfficher vaudra "true". Si isOk est false, on affiche "false" et valeurAAfficher vaudra "false"

Switch case, version ancienne

- L'instruction switch permet d'exécuter du code selon l'évaluation de la valeur d'une expression.

```
switch (monEntier) {  
    case 1 :  
        instr11;  
        instr12;  
        break;  
  
    case 2 :  
        break;  
    default :  
        ...  
}
```

- Si on oublie les breaks, le programme continue d'exécuter les instructions de haut en bas.
- Les instructions en bas de default sont exécutées si aucun "case" ne correspond à l'expression.

Switch case, version nouvelle (JDK 17 et plus)

- L'instruction switch évolue :

```
switch (monEntier) {  
    case 1, 2 -> {  
        instr11;  
        instr12;  
    }  
    case 3 -> {  
        instr31;  
    }  
    default ->{  
    }  
}
```

- La syntaxe n'est plus la même que celle du C, mais la compréhension du code est facilitée.

Exercice

- Compléter l'exercice : ExerciceConditions

Ce qu'il faut retenir

- Répéter des instructions de façon contrôlée est possible grâce aux boucles.
- Exécuter des instructions conditionnellement est possible grâce aux if, then , else et switch case.
- Avec tout ceci, il est possible de créer des algorithmes simples ou complexes.

Les conseils du formateur

- Il est possible d'utiliser les boucles et structures conditionnelles sans bloc de code : la première instruction après le for (dans le cas d'une boucle) sera répétée. Pour des raisons de lisibilité du code, cela est déconseillé.
- Il faut toujours réfléchir à l'endroit où les objets sont déclarés et instanciés, pour des raisons de lisibilité et de performance. Par exemple l'instanciation d'un même objet dans une boucle n'a pas de sens (il vaut mieux le sortir de la boucle si l'objet est réutilisable). En revanche, instancier un objet qui n'a de sens que dans un if, hors de ce if, n'a pas de sens non plus.
- Attention aux switch/case ancienne version. L'utilisation de break est contre intuitive.

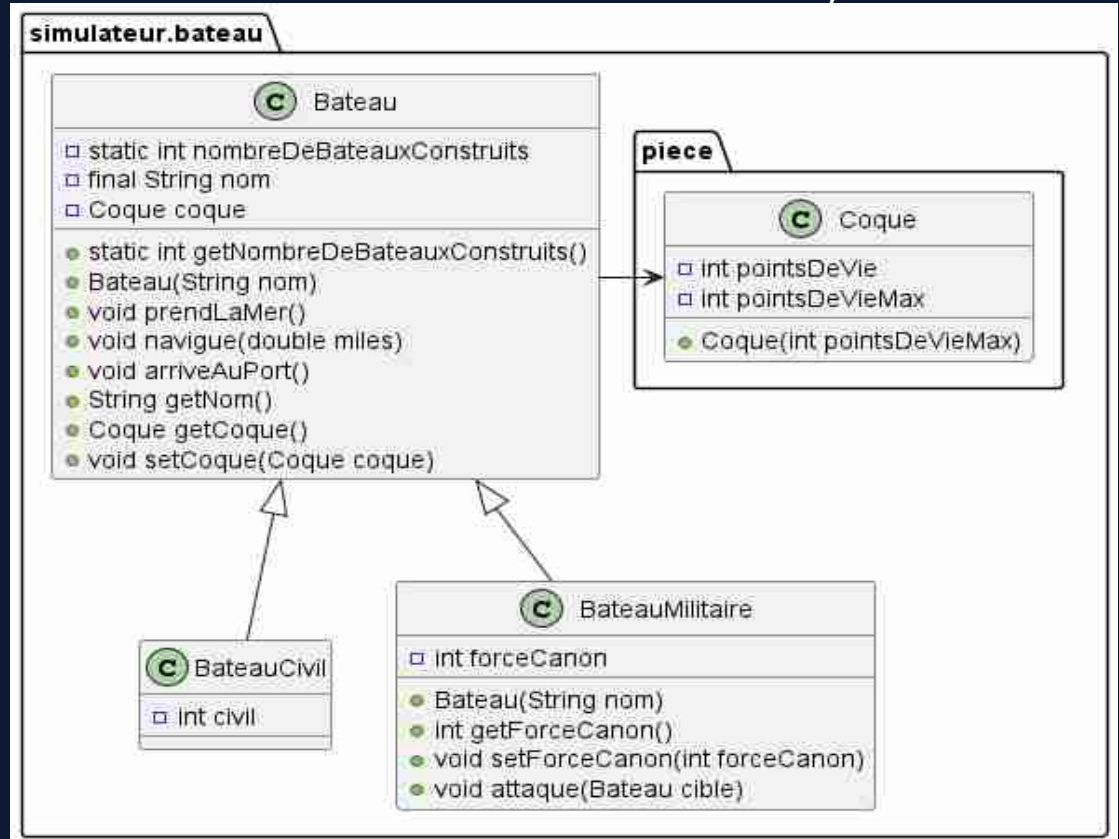


Héritage

Utilité de l'héritage

- Sur l'exemple du Bateau, si l'on veut modéliser différents types de bateaux, tout en réutilisant les méthodes et les attributs de Bateau, l'héritage peut être une solution.

Exemple : Le modèle de données doit évoluer pour pouvoir gérer des bateaux civils et militaires. Les bateaux civils transportent des civils. Les bateaux militaires peuvent attaquer un autre bateau. Une modélisation possible de ceci est :



L'héritage en Java

- Dans l'exemple précédent, les classes BateauCivil et BateauMilitaire héritent des attributs et des méthodes de Bateau. Un BateauCivil aura un attribut Moteur (venant de la classe Bateau), et un attribut civils (venant de sa classe).
- Il est courant d'appeler "Classe mère" ou "Classe parente" la classe dont on hérite et "Classe fille" la classe qui hérite.
- En Java, définir une relation d'héritage se fait avec le mot-clé "extends".
- Une classe ne peut avoir qu'un seul parent (un parent peut avoir plusieurs enfants). L'héritage peut se faire sur autant de niveaux que désiré.

```
public class BateauCivil extends Bateau{  
    private int civils;  
}
```

Visibilité des variables et redéfinition

- Une classe fille a accès à tous les attributs public et protected de la classe mère.
- Si l'on veut remplacer une méthode de la classe mère par une autre, on peut. Il suffit de la redéfinir : ré écrire une méthode avec la même signature (même nom, mêmes attributs, même valeur de retour) dans la classe fille. Dans ce cas, utiliser l'annotation `@Override` est recommandée, car elle peut éviter des erreurs de conception.

```
public class BateauCivil extends Bateau{  
    private int civils;  
    @Override  
    public boolean prendLaMer(){  
        System.out.println("Tuut Tuut j'embarque des gens");  
    }  
    ....}  
}
```

Héritage

super

- super permet d'accéder aux méthodes de la classe mère, à partir d'une classe fille.
- L'exemple ci-dessous affichera deux lignes de code (une due à la méthode prendLaMer() de Bateau et une autre due à la méthode prendLaMer() de BateauCivil).

```
public class BateauCivil extends Bateau{  
    private int civils;  
    public boolean prendLaMer(){  
        super.prendLaMer();  
        System.out.println("Tuut Tuut j'embarque des gens");  
    }  
    ....}  
}
```

Héritage

super

- super permet aussi d'appeler un constructeur de la classe parente.
- L'exemple ci-dessous affichera deux lignes de code (une due à la méthode prendLaMer() de Bateau et une autre due à la méthode prendLaMer() de BateauCivil).

```
public class BateauCivil extends Bateau{  
    public BateauCivil(String nom, int civils){  
        super(nom);  
        this.civils = civils;  
    }  
}
```

- De façon implicite, tout constructeur appelle super() si aucun autre appel à super() n'est fait. L'appel à super dans un constructeur doit être la première instruction.

Héritage final

- Le mot clé final peut être utilisé pour empêcher tout héritage.
- Il peut se placer sur une classe pour empêcher un héritage.
- Ou sur une méthode, pour empêcher une surcharge.
- Cela permet de protéger les informations à l'intérieur d'une classe.

La classe Object

- En Java, toute classe hérite soit d'une autre classe de façon explicite, soit de la classe Object.
- Ceci permet à toute classe d'avoir des fonctionnalités de base, comme `toString()`, `hashCode()`, `equals()` ...
- `equals` permet de comparer l'état interne de deux objets , alors que `==` compare uniquement des références.

instanceof

- instanceof permet de valider si une instance d'une classe peut être considérée comme une instance d'une autre classe.

```
| BateauMilitaire bateauMilitaire = new BateauMilitaire();  
| boolean isBateau = bateauMilitaire instanceof Bateau; //renvoie true  
| boolean isBateauMili = bateauMilitaire instanceof BateauMilitaire;  
| //renvoie true  
| boolean isBateauCivil = bateauMilitaire instanceof BateauCivil; //renvoie  
| false
```

Héritage

cast

- Comme vu dans les types, un cast est toujours possible pour forcer le type d'une variable vers un autre type.

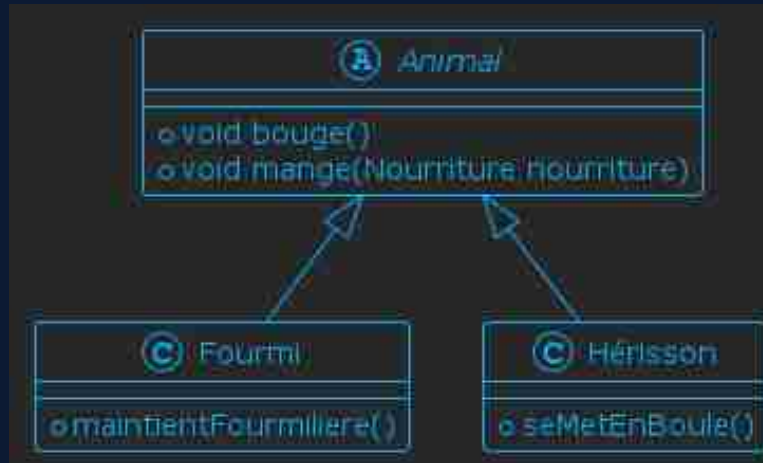
```
//Si on est sûr que derrière la variable bateau se cache un BateauCivil :  
BateauCivil bateauCivil = (BateauCivil) bateau;  
  
//Pour être sûr, mieux vaut tester tout d'abord avec instanceof  
if(bateau instanceof BateauCivil){  
    BateauCivil bateauCivil = (BateauCivil) bateau;  
}
```

Attention toutefois, si votre code contient trop de instanceof, cela peut vouloir dire que votre modèle n'est pas bien défini, puisque le polymorphisme n'est pas assez utilisé.

Héritage

abstract

- Une classe peut être définie comme étant abstraite. On utilise alors le mot clé `abstract`.
- Une classe `abstract` n'est pas "complète". Elle ne peut être instanciée car il lui manque généralement le code de certaines méthodes déclarées, mais non implémentées.
- Ces méthodes déclarées, mais non implémentées, sont aussi abstraites. Dès qu'une classe contient une méthode abstraite, elle n'est pas complète et doit être abstraite pour être compilée.
- Une classe abstraite peut être utile pour définir une hiérarchie de classes, sans pouvoir instancier la tête de la hiérarchie, car cette tête représente un concept qui n'est pas instanciable.



Polymorphisme

- Le polymorphisme est l'atout le plus intéressant de l'héritage. Il permet de considérer toute instance d'une classe fille comme étant aussi instance de la classe mère.
- L'héritage est une relation "est un". Le BateauMilitaire est un Bateau. Toute instruction qui fonctionne avec une instance de Bateau fonctionnera avec une instance de BateauMilitaire.
- A contrario, on dit que la composition est une relation "a un"

```
//Polymorphisme car on affecte un BateauMilitaire à un Bateau
Bateau bateau = new BateauMilitaire();

public void controleBateau(Bateau bateau){....}

//Polymorphisme car on appelle une méthode avec un argument de type
BateauMilitaire à la place de Bateau
BateauMilitaire bateauMilitaire = new BateauMilitaire();
controleBateau(bateauMilitaire);
```

- Créer les classes : BateauCivil et BateauMilitaire
- Surcharger les méthodes prendLaMer, navigue et arriveAuPort. Chacune de ces méthodes appelle la méthode de la classe parente puis affiche une ligne différente (par exemple : j'embarque n civils).
- Créer une classe Controleur, qui contient une méthode controle(Bateau bateau).
- Dans un main (quelconque), créer un Controleur, des bateaux civils et militaires et vérifier qu'ils puissent être contrôlés : donc passé en paramètre à Controleur.controle(Bateau bateau). Cette méthode fait prendre la mer, naviguer les navires et les fait revenir au port.

Ce qu'il faut retenir

- L'héritage permet de réutiliser des attributs et méthodes en définissant une relation de parenté entre les classes.
- Combiné avec les modificateur de visibilité, et le mot clé abstract, ceci permet de mutualiser du code si bien utilisé.

Les conseils du formateur

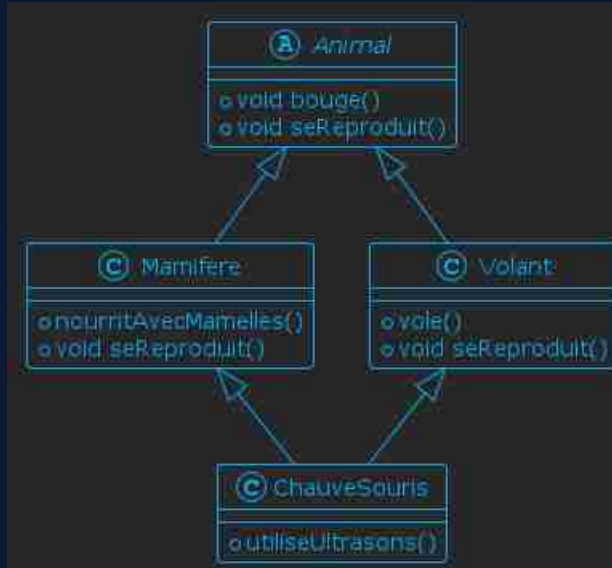
- La relation “est un” d’héritage est beaucoup plus puissante, mais contraignante que la relation “a un” de composition.
- Il faut toujours peser le pour et le contre avant de trancher entre ces deux relations quand on veut mutualiser du code (ce qui est une très bonne chose).
- La manière la plus “maline” d’utiliser l’héritage est aussi via les Design Patterns. Ceci est complètement hors du cadre de ce cours, mais peut vous servir quelque soit le langage (orienté objet) que vous utiliserez.



Interfaces

Le diamant de la mort

- En C++, une classe peut avoir deux parents. Ceci pose des problèmes notamment dans le cas du “diamant de la mort”.



Ici, si on appelle `chauveSouris.seReproduit`, la méthode réellement appelée n'est pas claire, car `Mammifere` et `Volant` ont tous les deux surchargé cette méthode. C'est pour cela que Java n'a pas implémenté d'héritage multiple.

Définition

- Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement à l'héritage multiple. On peut donc dire qu'une classe fille "est un" TypeA mais aussi "est un" TypeB ...
- Une interface est un ensemble déclarations de méthodes abstraites. Elle peut aussi contenir des constantes. C'est une sorte de "contrat" auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci. Les classes qui implémentent l'interface s'engagent à implémenter les méthodes abstraites (ou être elles-mêmes abstraites).
- Les interfaces se déclarent avec le mot clé interface et sont intégrées aux autres classes avec le mot clé implements. Une interface est implicitement déclarée avec le modificateur abstract.


```
| public interface Volant{  
|     public abstract void vole(); //<-- public abstract est facultatif ici  
| car les méthodes d'une interface sont forcément public ET abstract  
| }  
|
```

```
| public ChauveSouris implements Volant{  
|     ....  
|     public void vole(){  
|         ... code de vol  
|     } ...  
|
```

```
| public Pigeon implements Volant{  
|     ....  
|     public void vole(){  
|         ... autre code de vol  
|     } ...  
|
```

Polymorphisme

- L'implémentation d'une interface est une relation “est un”. Pigeon et ChauveSouris sont bien des “volant”.
- A contrario, on dit que la composition est une relation “a un”

```
//Polymorphisme car on affecte une ChauveSouris à un Volant
Volant volant = new ChauveSouris();

public void testVole(Volant volant){....}

//Polymorphisme car on appelle une méthode avec un argument de type Pigeon
à la place de Volant
Pigeon pigeon = new Pigeon();
testVole(pigeon);
```

Implémentation multiple

- Une classe peut implémenter de multiples interfaces.
- Une classe peut aussi hériter d'une autre classe et hériter de multiples interfaces.

```
public ChauveSouris extends Animal implements Volant, ChasseurUltraSons{  
    ...  
    public void vole(){  
        ... code de vol  
    } ...  
}
```

- Créer une interface Cargo, avec deux méthodes :
 - void chargeTonnage(int tonnage)
 - int dechargeTonnage()
- Créer une classe PorteContainer, et une classe Ravitailleur. PorteContainer hérite de BateauCivil et Ravitailleur de BateauMilitaire. Ces deux classes implémentent Cargo.
- Faire en sorte que les méthodes soient implémentées.
- Créer une interface Submersible avec les méthodes :
 - void plonge()
 - et void faitSurface()
- Créer une classe SousMarinAttaque qui hérite de BateauMilitaire et implémente Submersible
- Créer une classe SousMarinRavitailleur qui hérite de BateauMilitaire et implémente Submersible et Cargo.

Ce qu'il faut retenir

- Une interface est un ensemble de méthodes abstraites.
- Une classe qui implémente une ou des interfaces doit implémenter ces méthodes.
- Ainsi, une classe peut être éligible au polymorphisme de nombreuses interfaces.

Les conseils du formateur

- Si des classes qui ont peu de rapport conceptuel partagent une ou des fonctionnalités, il peut être intéressant de leur donner une interface commune, afin qu'un client de ces classes puisse utiliser de façon transparente l'une ou l'autre de ces classes.



Tableaux & Collections

Création de tableaux

- Un tableau est une suite ordonnée d'éléments du même type (objets ou types primitifs).
- Un tableau est considéré comme une instance d'un objet.
- On peut instancier un tableau de deux manières :
 - en spécifiant sa taille :

```
int monTableauDeDouzeEntiers[] = new int[12];
```

- en le remplissant (le compilateur en déduit la taille) :

```
String monTableauDeStrings[] = {"chaine1", "chaine2", "chaine3"};
```


Accès aux éléments du tableau

- La propriété `length` du tableau est égale à la taille, qui ne pourra être changée.
- Pour accéder (en lecture ou écriture) à un élément du tableau, on utilise un index :

```
monTableauDeStrings[0] = "hey!";  
System.out.println(monTableauDeStrings[1]);  
System.out.println(monTableauDeStrings[monTableauDeStrings.length]);  
//La dernière ligne renvoie une erreur !
```

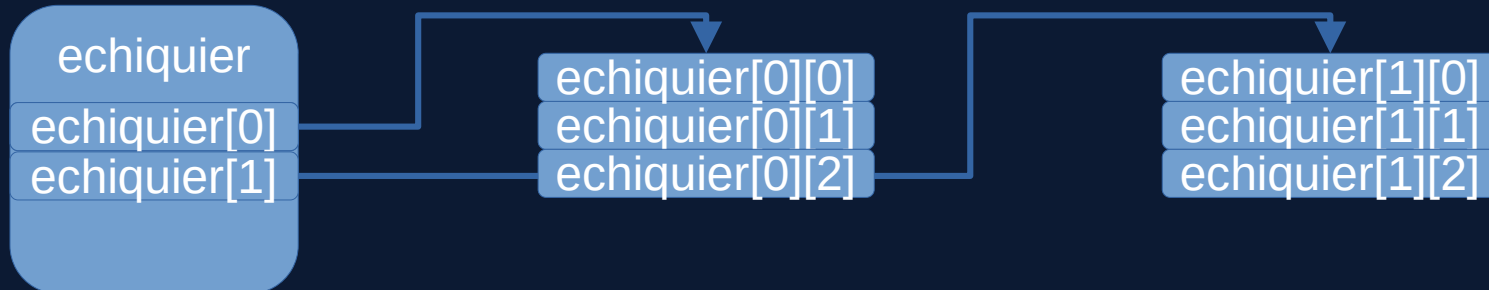
- Si un tableau a une longueur de `l`, les éléments accessibles sont ceux de 0 à `l-1`

Tableau multidimensionnel

- On peut créer un tableau de n'importe quelle dimension :

```
Case[][] echiquier = new Case[8][8];  
for(int i=0; i < 8; i++){  
    for(int j=0; j < 8; j++){  
        echiquier[i][j] = new Case();  
    }  
}
```

- Ceci peut être considéré comme un tableau à une dimension, dont chaque élément est lui aussi un tableau à une dimension.



Exercice

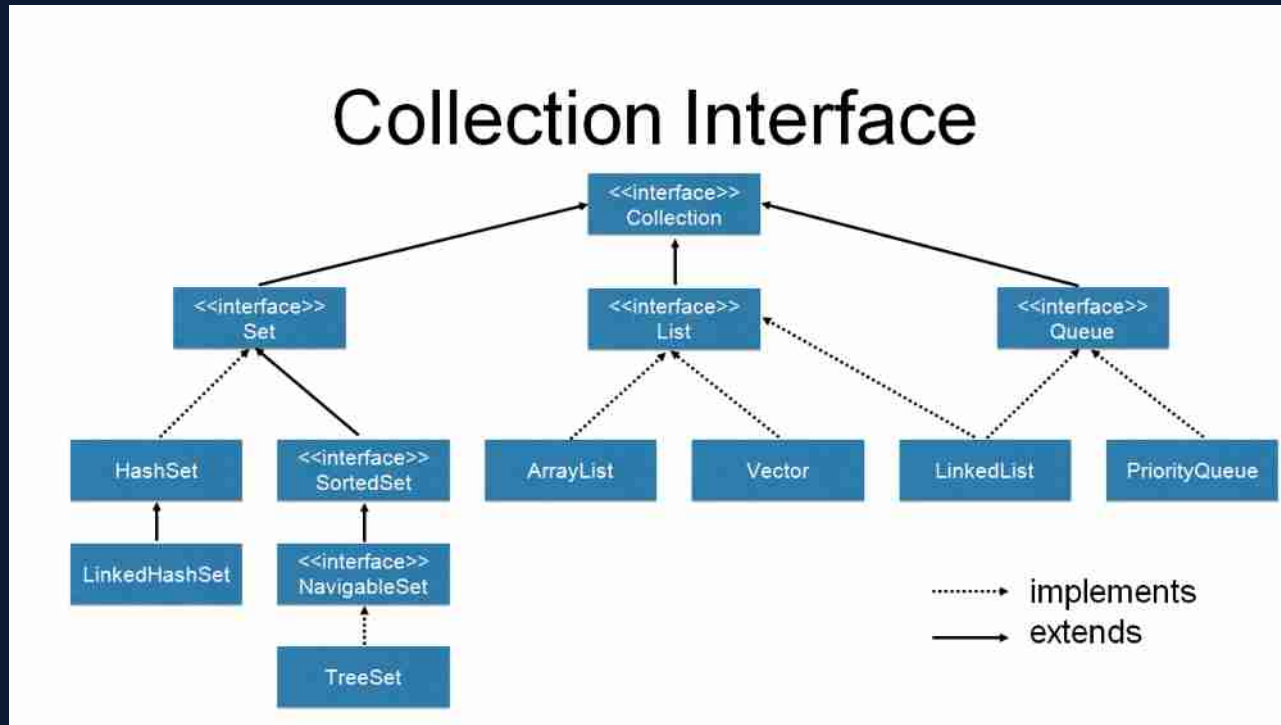
- Compléter l'exercice : ExerciceTableaux

Collection et Map

- Java offre de nombreuses classes et interfaces pour gérer des ensembles.
- Elles dérivent toutes de :
 - Map<K,V> un ensemble de clés-valeurs
 - Collection<E> un ensemble d'éléments

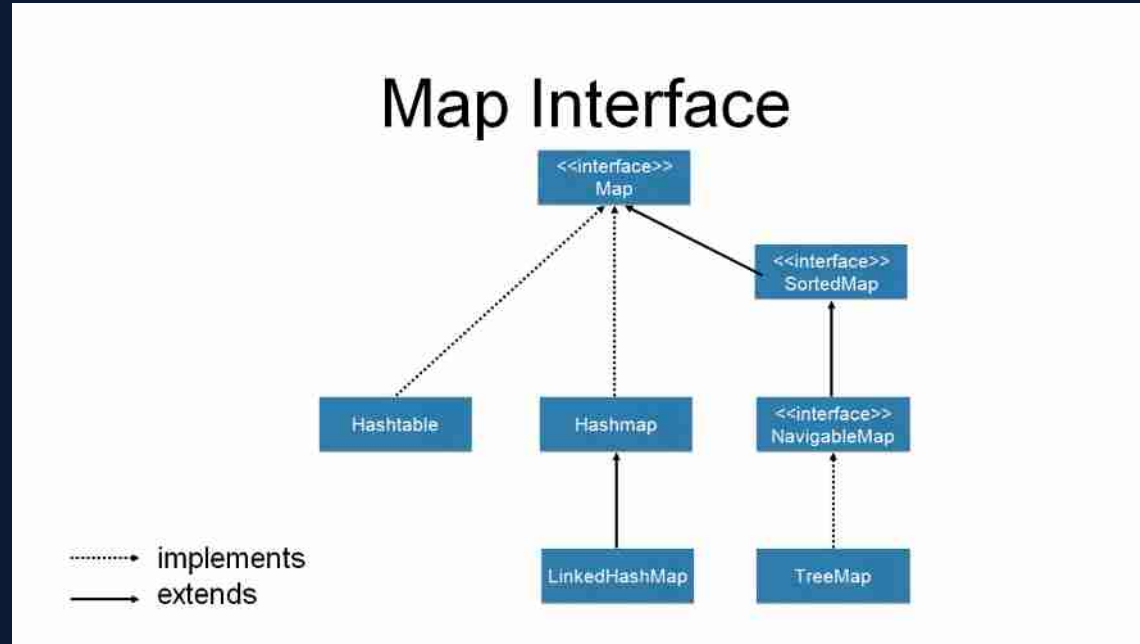
Utilité des collections

- Plus intelligentes qu'un tableau, les collections peuvent être redimensionnées, triées, dupliquées ... , en appelant des méthodes.



Utilité des maps

- Les Maps gèrent une collection de clés : valeurs. On affecte à chaque clé une valeur. On peut comparer ceci à un tableau dont l'index pourrait être de n'importe quel type.



Les listes

- Les List sont des collections d'éléments ordonnés. Par ordonnés, on entend qu'un ordre leur est affecté, mais ils ne sont pas forcément triés. Une liste
- List est une interface, pour créer une nouvelle liste, il faut une classe qui implémente cette interface. La plus commune est ArrayList : une classe qui implémente List et qui utilise (en interne) un tableau.
- Comme toutes les Collection, on peut (et il est recommandé de le faire) définir le type d'objets qui sera dans la collection avec la syntaxe <TypeElement>.
- Ainsi, on peut créer une ArrayList<> contenant des Integer de cette façon :

```
List<Integer> maListe = new ArrayList<>();
```

Les listes

- La syntaxe précédente est usuelle. Le polymorphisme est utilisé pour se forcer à n'utiliser que les méthodes de l'interface List, afin que tout le reste du code ne dépende pas de l'implémentation ArrayList. Ainsi, si le développeur veut changer d'implémentation, il peut le faire de manière transparente.
- Une liste peut contenir des éléments null, ou des doublons.
- List offre de nombreuses méthodes pour gérer la collection. Les méthodes se trouvent ici :
<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

```
List<Integer> maListe = new ArrayList<>();
```


Les listes

- Un exemple d'utilisation de liste :

```
List<String> maListe = new ArrayList<>();
maListe.add("tout ");
maListe.add("le ");
maListe.add(0,"Salut ");
System.out.println(maListe.size());
for(int i=0; i<maListe.size(); i++){
    System.out.println(maListe.get(i));
}

List<String> autreListe = new ArrayList<>(maListe);
autreListe.add(" Monde !!! ");
System.out.println(autreListe.size()); //affiche 4
```

Les sets

- Les sets sont des collections d'éléments distincts (ils peuvent contenir un null).
- Les sets ne sont pas ordonnés.
- Set est une interface, la classe implémentant Set la plus commune est HashSet.

```
Set<String> maListe = new HashSet<>();
```

Les sets

- Un exemple d'utilisation de set :

```
Set<String> monSet = new HashSet<>();
monSet.add("salut ");
monSet.add("le ");
monSet.add("salut ");
monSet.add("monde ");
System.out.println(monSet.size());
for(Iterator<String> iterator = monSet.iterator(); iterator.hasNext();){
    System.out.println(iterator.next());
    //Va afficher trois lignes, dans un ordre non garanti
}
```

Le pattern Iterator et la boucle foreach

- Il est possible d'itérer sur une liste avec un index, car la liste est ordonnée.
- Pour un Set, ce n'est pas possible.
- Le pattern Iterator est alors utile.
- Depuis Java5, une amélioration syntaxique permet de cacher l'usage de l'Iterator :

```
for(Iterator<String> iterator = monSet.iterator(); iterator.hasNext();){  
    System.out.println(iterator.next());  
}  
  
//Les deux syntaxes sont équivalentes :  
for(String string : monSet){  
    System.out.println(string);  
}
```

Le pattern Iterator et la boucle foreach

- Il n'est pas possible de boucler sur une collection avec un for each, et de la modifier en même temps (une ConcurrentModificationException est lancée).
- Par contre, utiliser explicitement Iterator permet de le faire.

```
for(Iterator<String> iterator = monSet.iterator(); iterator.hasNext();){  
    String value = iterator.next();  
    if(value.equals("coucou")){  
        iterator.remove();  
    }  
}
```

La boucle foreach

- La boucle foreach est utilisable pour les tableaux, et pour tout objet implémentant Iterable (une interface).
- Or Collection hérite d'Iterable : toutes les collections sont donc Iterable

```
for(String string : maCollection){  
    System.out.println(string);  
}
```

Les maps

- Map<K,V> est une interface proposant des méthodes pour gérer un ensemble de clés valeurs. A chaque clé est associée une valeur dans la map.
- Les deux implémentations les plus utilisées sont :
 - HashMap<> pour une Map non triée, mais avec accès rapide.
 - TreeMap<> pour une Map avec des clés triées.

```
Map<Integer, String> maMap = new HashMap<>();  
maMap.put(1,"Ccucou");  
maMap.put(1,"Coucou");  
maMap.put(2,"le monde");  
String valeur = maMap.get(2);  
maMap.remove(2);  
...
```

La classe utilitaire Collections

- Collections est une classe utilitaire contenant de nombreuses méthodes statiques. Elle propose :
 - `sort()` pour trier une collection
 - `emptySet()`, `emptyList()` pour créer des collections vides.
 - `shuffle()` pour mélanger une liste ...

Tris

- Trier une collection ordonnée est possible en Java (avec par exemple Collections.sort).
- Il existe des collections triées : chaque ajout d'élément à la collection le fait à la bonne position.
- Les éléments sont triés selon deux possibilités:
 - Leur ordre "naturel". Celui-ci est défini si le type de l'élément implémente l'interface Comparable. Comparable a une seule méthode : `compareTo(other)` qui renvoie un nombre négatif si `this` est plus petit que `other`, 0 s'ils sont à la même position, et un nombre positif si `this` est plus grand.
 - On fournit une instance qui implémente `Comparator`. Cette instance définit la règle triant les éléments. `Comparator` contient la méthode `compare(objet1,objet2)` qui renvoie un entier négatif si `objet1` est plus petit `objet2`, positif si `objet2` est plus grand qu'`objet1`, et 0 dans les autres cas.

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

Exercice

- Compléter l'exercice : ExerciceSet
- Compléter l'exercice : ExerciceList
- Compléter l'exercice : ExerciceMap
- Compléter l'exercice : ExerciceCollections

Ce qu'il faut retenir

- Les implémentations de Collection et Map sont riches et nombreuses.
- Utiliser un tableau ne doit se faire qu'en dernier recours (uniquement si une bibliothèque qu'on utilise fournit des tableaux).
- La boucle foreach est plus agréable à lire que l'Iterator. Toutefois, Iterator peut être utile si l'on veut supprimer des éléments d'une collection en itérant dessus.

Les conseils du formateur

- Prendre quelques minutes pour lire la documentation des classes et interfaces est recommandé pour comprendre comment doit être utilisée une collection (ou une map).
- De même, avant de coder une classe de collection, prendre un peu de temps pour voir si la classe dont on a besoin n'existe pas (dans les classes Java, ou dans d'autres bibliothèques, comme Apache Collections, ou Guava ...)
- Les collections font un usage intensif des classes, des interfaces dans le but d'aider les développeurs à avoir une API (Interface de Programmation Avancée) la plus utile possible. Peut-être cela vous convaincra-t-il de l'utilité des interfaces et de l'héritage :)



Enumerations

Enumérations

Utilité

- En Java, si on veut modéliser un ensemble de valeurs discrètes, on peut utiliser les énumérations. Une énumération utilise le mot clé “enum” pour être déclarée. Les énumérations sont souvent appelées enum.

```
public enum StatutEntreprise {  
    ACTIVE, EN_PAUSE, RADIEE;  
}
```

- On utilise ensuite les enums comme une classe contenant uniquement des constantes (ou des variables statiques) :

```
StatutEntreprise statut = StatutEntreprise.EN_PAUSE;  
...  
if(statut == StatutEntreprise.ACTIVE){  
    //déclencher appel annuel de cotisation  
}
```

Enumérations

Utilisation

- Les énumérations proposent des méthodes statiques :
 - `values` permet de récupérer un tableau de toutes les valeurs de l'enum
 - `valueOf(String)` permet de transformer une chaîne de caractère en valeur de l'énumérée (ou lancera une `Exception` si aucune valeur énumérée ne correspond).
- Les énumérations s'utilisent bien avec un `switch case` : le compilateur peut détecter si des valeurs ont été oubliées.

Enrichissement de l'enum

- On peut aussi modifier une enum comme une classe, pour ajouter des méthodes et attributs, afin d'enrichir fonctionnellement l'enum :

```
public enum StatutEntreprise {  
    ACTIVE(true),  
    EN_PAUSE(false),  
    RADIEE(false);  
  
    private boolean cotisable;  
  
    private StatutEntreprise(boolean cotisable){  
        this.cotisable = cotisable  
    }  
  
    public isCotisable(){  
        return this.cotisable;  
    }  
}
```


Exercice

- Dans le package où se trouve Bateau, créer une énumération EtatBateau.
- Lui ajouter les valeurs OPERATIONNEL, EN_PANNE, COULE.
- Rajouter un attribut etat de type EtatBateau à Bateau. La valeur initiale est : OPERATIONNEL
- Ajouter les getter et setter associés .

Ce qu'il faut retenir


- Les enums sont utilisées pour stocker un ensemble de valeurs discrètes.
- Elles sont utiles dans le cas de contrôle avec if ou un switch.
- Il est possible d'ajouter des attributs et méthodes aux énumérations au besoin.

Les conseils du formateur

- Avant Java 5, les énumérations n'existaient pas. Pour pallier à ce manque, les développeurs utilisaient des ensembles de constantes dans les classes.
- Par exemple :

```
public class CommeUneEnum {  
    public static final VALEUR_1 = "valeur_1";  
    public static final VALEUR_2 = "valeur_2";  
}
```

- Ce pattern peut encore être présent pour des raisons historiques, mais il ne doit plus être utilisé : l'utilisation d'enums apporte beaucoup plus de contrôle au niveau de la compilation. Par exemple, le développeur de CommeUneEnum peut modifier valeur_1 en valeur1 sans problème de compilation. Mais si un autre utilisateur de la classe utilisait : `if(maValeur.equals(VALEUR_1))` son code ne fonctionnera plus .



Lambdas & Streams

Lambdas

- Java 8 intègre les lambdas, des blocs de code courts représentant des méthodes.
- Ces méthodes prennent 0 ou plusieurs paramètres et renvoient une valeur ou void.
- Ci-dessous un exemple de lambda avec un bloc

```
(parameter1, parameter2) -> { return parameter1 };
```

- On peut stocker ces méthode dans une variable (de type Consumer, BiConsumer), ou les utiliser telles quelles.

```
Consumer<String> method = ( n ) ->  
{ System.out.println(n); };
```

- L'intérêt des lambdas est qu'elles peuvent être utilisées en paramètre de méthodes, pour paramétrer un traitement. Ceci est appelé programmation fonctionnelle.

Lambdas : corps d'une expression

Le corps d'une expression lambda est défini à droite de l'opérateur `->`. Il peut être :

- une expression unique
- un bloc de code composé d'une ou plusieurs instructions entourées par des accolades

Le corps de l'expression doit respecter certaines règles :

- il peut n'avoir aucune, une seule ou plusieurs instructions
- lorsqu'il ne contient qu'une seule instruction, les accolades ne sont pas obligatoires et la valeur de retour est celle de l'instruction si elle en possède une
- lorsqu'il y a plusieurs instructions, elles doivent obligatoirement être entourées d'accolades
- la valeur de retour est celle de la dernière expression ou `void` si rien n'est retourné

Lambdas : les variables

Dans le corps d'une expression lambda, il est possible d'utiliser :

- Les variables passées en paramètre de l'expression
- Les variables définies dans le corps de l'expression
- Les variables **final** définies dans le contexte englobant
- Les variables effectivement final définies dans le contexte englobant : ces variables ne sont pas déclarées final mais une valeur leur est assignée et celle-ci n'est jamais modifiée. Il serait donc possible de les déclarer final sans que cela n'engendre de problème de compilation. Le concept de variables effectivement final a été introduit dans Java 8. Ce sont par exemple les arguments de la méthode qui contient la lambda.

Utilisation des lambdas : les streams

L'API Stream permet de réaliser des opérations fonctionnelles sur un ensemble d'éléments. De nombreuses opérations de l'API Stream attendent en paramètre une interface fonctionnelle ce qui conduit naturellement à utiliser les expressions lambdas et les références de méthodes dans la définition des Streams. Un Stream permet donc d'exécuter des opérations standards dont les traitements sont exprimés grâce à des expressions lambdas ou des références de méthodes.

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

Streams: première méthode

- En passant une lambda à `forEach`, on peut exécuter une méthode pour chaque élément d'un stream:

```
List<Integer> entiers = new ArrayList<>();  
entiers.add(1);  
entiers.add(2);  
entiers.stream().forEach(i ->  
    System.out.println(i));
```

- En chaînant les streams, et en ajoutant deux lambdas, on peut afficher un message sur les entiers dont la valeur vaut 1:

```
entiers.stream()  
    .filter(v -> v == 1)  
    .forEach(i ->  
        System.out.println(i));
```

Streams: d'autres méthodes

- On peut mapper (avec les méthodes map...) pour transformer les éléments du stream, puis utiliser une méthode terminale pour agréger des résultats :

```
long somme = entiers.stream()
                    .filter(v -> v
<10)
                    .mapToInt(i -> i)
                    .sum();
```

- On peut aussi reconstruire une collection avec l'aide la classe utilitaire Collectors :

```
List<Integer> newList =
entiers.stream()
    .filter(v -> v < 10)
    .collect(Collectors.toList());
```

Finalement, on peut simplement lancer le calcul en parallèle :

```
List<Integer> newList =
entiers.parallelStream()
    .filter(v -> v < 10)
    .collect(Collectors.toList());
```

Streams: opérations de filtre

- `filter(Predicate)` : renvoie un Stream qui contient les éléments pour lesquels l'évaluation du Predicate passé en paramètre vaut true
- `distinct()` : renvoie un Stream qui ne contient que les éléments uniques (elle retire les doublons). La comparaison se fait grâce à l'implémentation de la méthode `equals()`
- `limit(n)` : renvoie un Stream qui ne contient comme éléments que le nombre fourni en paramètre
- `skip(n)` : renvoie un Stream dont les n premiers éléments sont ignorés

Streams: opérations de transformation

- `map(Function)` : applique la Function fournie en paramètre pour transformer l'élément en créant un nouveau
- `flatMap(Function)` : applique la Function fournie en paramètre pour transformer l'élément en créant zéro, un ou plusieurs éléments

Streams: opérations de recherche

- `anyMatch(Predicate)` : renvoie un booléen qui précise si l'évaluation du Predicate sur au moins un élément vaut true
- `allMatch(Predicate)` : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut true
- `noneMatch(Predicate)` : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut false
- `findAny()` : renvoie un objet de type `Optional` qui encapsule un élément du Stream s'il existe
- `findFirst()` : renvoie un objet de type `Optional` qui encapsule le premier élément du Stream s'il existe

Streams: opérations de réduction

- `reduce()` : applique une Function pour combiner les éléments afin de produire le résultat
- `collect()` : permet de transformer un Stream qui contiendra le résultat des traitements de réduction dans un conteneur mutable



Exceptions

Utilité des exceptions

- Une méthode en Java peut renvoyer un objet en retour.
- Si toutefois la méthode rencontre un événement qu'elle ne peut gérer, que doit-elle renvoyer ? Un objet spécial qui correspond à l'erreur (comme une valeur négative pour un int qui doit être positif) ? Ce n'est pas toujours possible (et pas correct au niveau programmation orientée objet).
- Il faut néanmoins que le code appelant la méthode soit au courant qu'un événement exceptionnel a eu lieu, et qu'il faut le traiter.
- En Java, ceci est possible grâce aux Exceptions.
- Une méthode peut lancer une exception, au même titre qu'elle peut renvoyer un résultat :

```
public void ajouteAuPanier(String referenceProduit) throws ReferenceInconnueException
```


Syntaxe

Une méthode doit déclarer la ou les exceptions qu'elle lance (ce n'est pas toujours vrai, cela sera éclairci plus tard) avec le mot clé throws :

```
public int methode() throws ExceptionA, ExceptionB ...
```

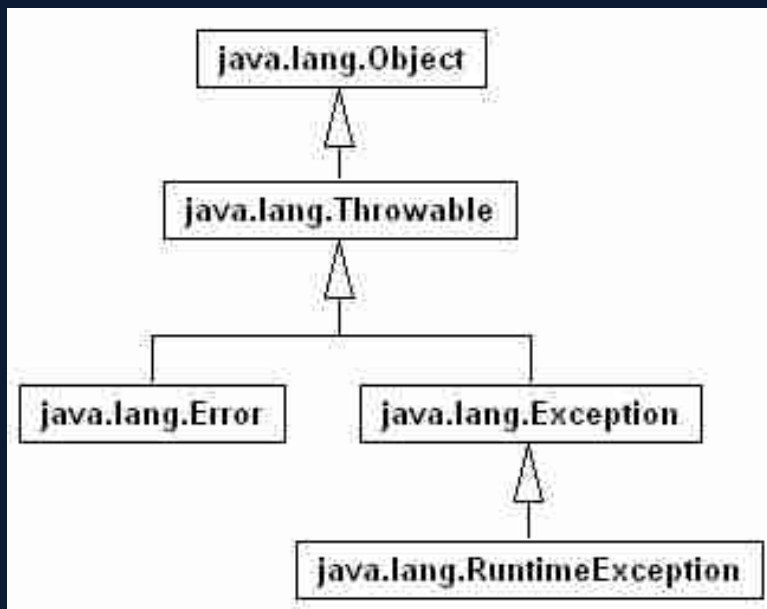
Une instruction doit lancer l'exception au sein de la méthode :

```
public int calcule() throws ExceptionA{  
    ...  
    if(erreurSurvenue){  
        throw new ExceptionA("Une erreur est survenue");  
    }  
    return int,  
}
```

La syntaxe pour lancer une exception est "throw instanceOfThrowable"

Throwable

Throwable est une classe, au sommet de la hiérarchie des classes qui peuvent être jetées (donc être placées après throw). Dès lors qu'une classe hérite de Throwable, ses instances peuvent être jetées. Ceci concerne bien sûr Exception (et ses classes filles), ainsi qu'Error.



Lancement avec throw

```
public int calcule() throws ExceptionA{  
    ...  
    if(erreurSurvenue){  
        throw new ExceptionA("Une erreur est survenue");  
    }  
    System.out.println("Tout va bien");  
    return int;  
}
```

Dans le code précédent, si erreurSurvenue vaut true :

- Une instance de ExceptionA est créée avec new (c'est une construction "classique" d'instance d'une classe).
- Elle est jetée : la méthode appelante doit maintenant la gérer (sinon la compilation est impossible).
- Les lignes de code qui suivent ne sont pas exécutées. Ici "Tout va bien" ne sera pas affiché et la méthode ne renverra pas d'int.

Récupération avec catch

```
...  
try{  
    int resultat = calcule();  
    System.out.println("Le résultat vaut : " + resultat);  
} catch(ExceptionA exception){  
    System.out.println("Erreur : le résultat n'est pas calculable.");  
}
```

- Le bloc de code ci-dessus représente un appel à la méthode vue précédemment : `calcule()`.
- Pour que le code compile, étant donné que `calcule()` lance une exception de type `ExceptionA`, il faut que compile soit dans un bloc de méthode `try{} catch(ExceptionA exception){}`
- Et il faut que ce bloc soit suivi d'un bloc `catch(ExceptionA exception){}`
- Si aucune exception n'est lancée , tout le bloc `try` sera exécuté, mais pas le bloc `catch`.
- Si par contre `calcule()` lance une exception, les lignes suivantes du bloc `try` ne sont pas exécutées, mais le bloc `catch` est exécuté.

Utilisation de l'exception dans le catch

```
try{
    int resultat = calcule();
    System.out.println("Le résultat vaut : " + resultat);
} catch(ExceptionA exception){
    System.out.println("Erreur : le résultat n'est pas calculable.");
    System.out.println("La cause est : " + exception.getMessage());
}
```

- Le bloc catch déclare une variable, qui contient l'exception lancée.
- On peut donc se servir de cette variable pour en récupérer des informations.
- Ces informations peuvent être utiles pour :
 - afficher les raisons de l'erreur,
 - exécuter un code différent ...
- Libre au développeur de créer sa propre implémentation de Exception, afin d'y rajouter des attributs et des méthodes pour traiter au mieux les erreurs.

Règles de récupération avec catch

```
try{
    int resultat = calcule();
}catch(ExceptionCalcul exception){
    System.out.println("Erreur : le résultat n'est pas calculable.");
}catch(ExceptionHeure exception){
    System.out.println("Erreur : il n'est pas possible de lancer un
calcul à cette heure.");
}catch(Exception exception){
    System.out.println("Erreur inconnue.");
}
```

- On peut chaîner les blocs catch, de façon à traiter les erreurs différemment en fonction de l'exception renvoyée.
- Dans tous les cas, un seul bloc catch sera exécuté.
- A l'exécution, c'est toujours l'Exception la plus spécifique qui sera exécutée.

Héritage et récupération

```
try{
    int resultat = calcule();
}catch(Exception exception){
    System.out.println("Erreur inconnue.");
}
```

- Dans le code ci-dessus, si `calcule()` lance n'importe quelle `Exception` qui hérite de la classe `Exception`, elle sera attrapée par le `catch(Exception)`

```
try{
    int resultat = calcule();
}catch(ExceptionA exception){
    System.out.println("Erreur de type A.");
}catch(Exception exception){
    System.out.println("Erreur inconnue.");
}
```

- A l'exécution, c'est le `catch` avec l'exception du niveau le plus bas, et dont hérite l'exception lancée, qui va être exécuté. Dans le code ci-dessus, si `calcule()` lance une `ExceptionA`, c'est le premier bloc `catch()` qui sera exécuté, même si `ExceptionA` hérite de `Exception`.

Multi récupération

```
try{
    int resultat = calcule();
}catch(ExceptionA | ExceptionB | ExceptionC exception){
    System.out.println("Erreur A B ou C : " + exception.getMessage());
}
```

- Depuis Java7, la multi récupération (ou multi catch) est possible, en séparant les types d'Exceptions dans un catch par le caractère |
- Ceci permet de mutualiser du code de gestion d'exceptions

finally

finally est un bloc exécutant du code, à la fin d'un bloc catch, qu'une exception ait été lancée ou non. Il peut être utilisé notamment pour nettoyer des ressources après une opération (que celle-ci ait réussi ou non).

```
try{
    int resultat = calcule();
}catch(ExceptionA | ExceptionB | ExceptionC exception){
    System.out.println("Erreur A B ou C : " + exception.getMessage());
}finally{
    System.out.println("Cette ligne s'affichera toujours");
    monSystemeDeCalcul.libereRessources();
}
```

Problème de la libération de ressources lançant des exceptions

Si une ressource lance une exception en se lançant, cela peut amener le développeur à écrire un code comme suit :

```
FileInputStream fileInputStream = null;
File monFichier = new File("mon-fichier.txt");
try {
    fileInputStream = new FileInputStream(monFichier);
    byte[] fileBytes = fileInputStream.readAllBytes();
} catch (IOException e) {
    e.printStackTrace();
}finally {
    if(fileInputStream != null) {
        try {
            fileInputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

try with resources

Depuis Java7, de nombreuses ressources que l'on peut ouvrir et fermer implémentent l'interface `AutoCloseable`. Or tout `AutoCloseable` peut être utilisé dans un `try with resources` :

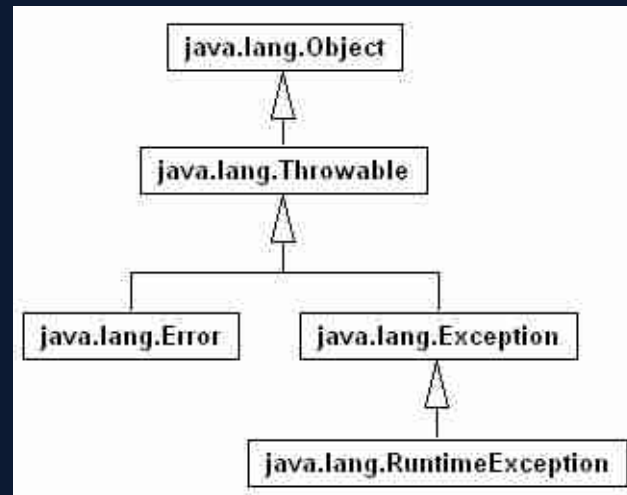
```
File monFichier = new File("mon-fichier.txt");
try (FileInputStream fileInputStream = new FileInputStream(monFichier))
{
    byte[] fileBytes = fileInputStream.readAllBytes();
} catch (IOException e) {
    e.printStackTrace();
}
```

Le code ci-dessus s'assure que `fileInputStream` sera fermé (s'il n'est pas null) qu'une exception survienne ou pas.

Rethrow de l'exception

- Soit methodeA qui appelle methodeB qui elle même lance une exception.
- methodeA peut ne pas attraper l'exception de methodeB et déclarer qu'elle lance elle-même l'exception.
- L'exception devra alors être traitée par la méthode appelant methodeA.
- Et ainsi de suite ... l'exception peut ainsi remonter jusqu'à la première méthode appelante (ce peut être la méthode main appelée par la JVM)

Les classes filles de Error



- La hiérarchie vue précédemment a montré Error : une classe qui hérite aussi de Throwable.
- Cette classe a de nombreuses classes filles, qui représentent une erreur système : ex OutOfMemoryError lorsque la JVM ne peut plus travailler car elle n'a plus de mémoire.
- On peut attraper ces Error, mais cela ne sert pas à grand chose puisque le système est déjà dans un état critique (logger l'erreur peut quand même servir ...). Après une Error, la JVM s'arrête.
- Il vaut mieux ne pas redéfinir de classe fille d'Error.

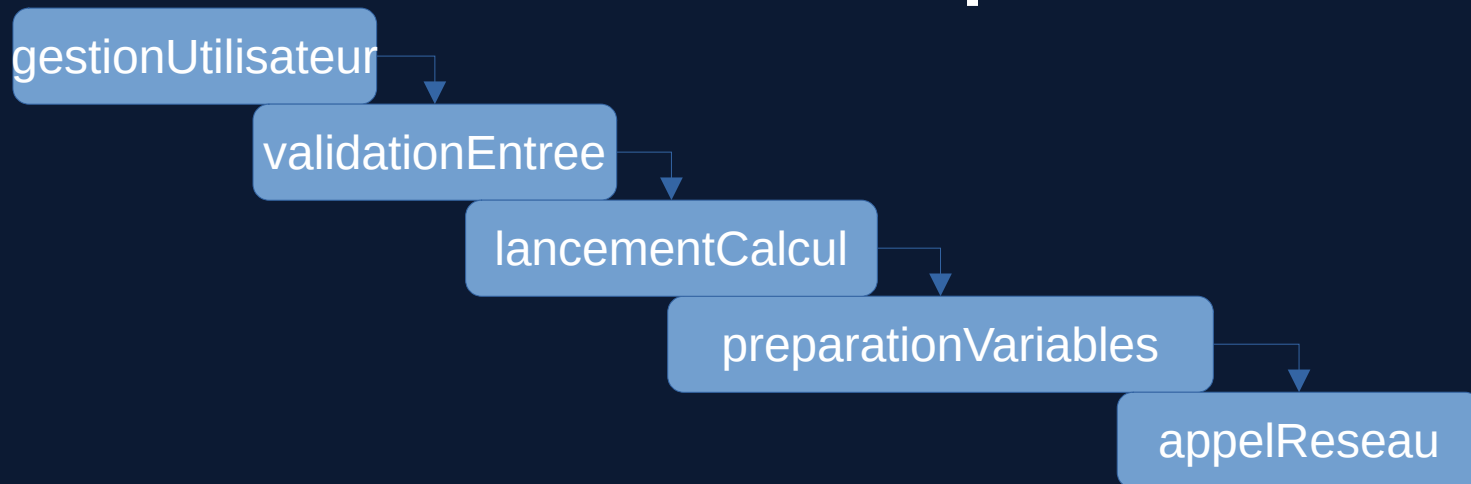
Les classes filles de RuntimeException

- Toutes les classes qui héritent de RuntimeException sont des exceptions normales mais ...
- ... on peut les lancer sans rien déclarer dans la signature de la méthode (pas de throws) ...
- ... et le développeur n'est pas obligé de les attraper (pas d'obligation de coder un catch).
- Elles peuvent remonter ainsi jusqu'à la première méthode appelante (la méthode main ou celle du Thread).
- Si elles ne sont pas attrapées, elles arrêtent le Thread ou la JVM.
- On ne vérifie pas si elles sont attrapées, et sont donc appelées `UncheckedExceptions` (les erreurs aussi sont `UncheckedException` puisque rien n'oblige à les attraper).

Checked ou UncheckedExceptions ?

- Si l'on crée une nouvelle classe d'Exception, faut-il hériter de Exception et forcer les développeurs à attraper l'Exception,
- ou faut-il hériter de RuntimeException et laisser le choix au développeur d'attraper ou non l'Exception ?
- La tendance historique était d'hériter d'Exception, par conséquence tous les frameworks obligeaient les développeurs à coder des catch partout, ou à relancer les Exceptions, ce qui n'était pas pratique.
- La tendance recommandée est :
 - Si l'Exception peut être traitée de façon à ce que l'opération puisse être continuée sans problème, lancer une Exception normale.
 - Sinon lancer une Exception de type RuntimeException, qui sera sans doute récupérée pour afficher un message avertissant qu'un problème a eu lieu.

Checked ou UncheckedExceptions ?



- Le développeur de appelReseau doit gérer le cas où le réseau ne répond pas.
- Dans ce cas là, toutes les méthodes appelantes ne peuvent rien faire que de remonter l'erreur à gestionUtilisateur, pour afficher un message d'erreur (la méthode preparationVariables ne peut faire refonctionner le réseau).
- Il vaut mieux lancer une RuntimeException et laisser gestionUtilisateur attraper toutes ces erreurs, et en tirer un message à afficher à l'utilisateur.
- Sinon toutes les méthodes intermédiaires (et il peut y en avoir des dizaines en entreprise) vont devoir déclarer des catch ou des throws pour chaque type d'exception.

Exercice

- Compléter l'exercice : ExerciceExceptions
- Compléter l'exercice : ExerciceExceptions2 (la consigne est dans la méthode : lanceCalculPourDeVrai

Ce qu'il faut retenir

- La gestion des exceptions ajoute une nouvelle dimension au flux des appels de méthode.
- Les exceptions sont des instances de classes qui héritent de Throwable (mais très généralement de Exception ou RuntimeException).
- Elles sont lancées avec throw.
- Une méthode déclare qu'elle lance une Exception avec throws
- Si une méthode déclare un lancement d'Exception, la méthode appelante doit :
 - L'attraper avec un bloc try/catch (optionnellement finally)
 - La rejeter (avec la clause throws)

Les conseils du formateur

- Assurez-vous que les méthodes de plus haut niveau attrapent toujours les Exception, pour au moins les logger si elles surviennent.
- Si vous devez créer des classes Exceptions, servez-vous des règles Checked ou Unchecked pour connaître quel genre d'Exception créer.
- Try with resources est un pattern très utile pour toutes les ressources que l'on doit fermer.