JAVA Avancé

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA Avancé

POO - Génériques – Collections - Exceptions - Lambdas - Stream

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Programmation Orientée Objet

Qu’est-ce que la Programmation Orientée Objet ?

- La **POO** est est un paradigme de programmation informatique. Elle consiste en la **définition** et l'**interaction** de briques logicielles appelées **objets**. Un **objet** représente un **concept**, une **idée** ou toute **entité** du monde physique (personne, voiture, dinosaure).
- Lorsque que l'on programme avec cette méthode, la première question que l’on se pose est :
  « qu'est-ce que je manipule ? »
- Alors qu'en programmation **Procédurale**, c'est plutôt :
  « qu'est-ce que je fait ? »

Qu'est-ce que la Programmation Orientée Objet ?

- Elle permet de **découper** une grosse **application**, généralement floue, en une multitude d'**objets** interagissant entre eux
- La POO améliore également la **maintenabilité**. Elle facilite les **mises à jour** et l'ajout de **nouvelles fonctionnalités**.
- Elle permet de faire de la **factorisation** et évite ainsi un bon nombre de lignes de code
- La réutilisation du code fut un argument déterminant pour venter les avantages des langages orientés objets.

Les paradigmes de la POO

La POO repose sur plusieurs concepts importants

- **Accessibilité** (ou **Visibilité**)
- **Encapsulation**
- **Polymorphisme**
- **Héritage**
- **Abstraction**
- **Interfaces**
- **Fonctions Anonymes**
- **Généricité**

Nous les aborderons tous par la suite.

Qu’est-ce qu’un objet en programmation?

Commençons par définir les objets dans le mode réel:

- Ils possèdent des **propriétés propres** : Une chaise a 4 pieds, une couleur, un matériaux précis...
- Certains objets peuvent **faire des actions** : la voiture peut rouler, klaxonner...
- Ils peuvent également **interagir entre eux** : l’objet roue tourne et fait avancer la voiture, l’objet cric monte et permet de soulever la voiture...

Le concept d’objet en programmation s’appuie sur ce fonctionnement.

Qu’est-ce qu’un objet en programmation?

Il faut distinguer ce qu’est l’objet et ce qu'est la définition d’un objet

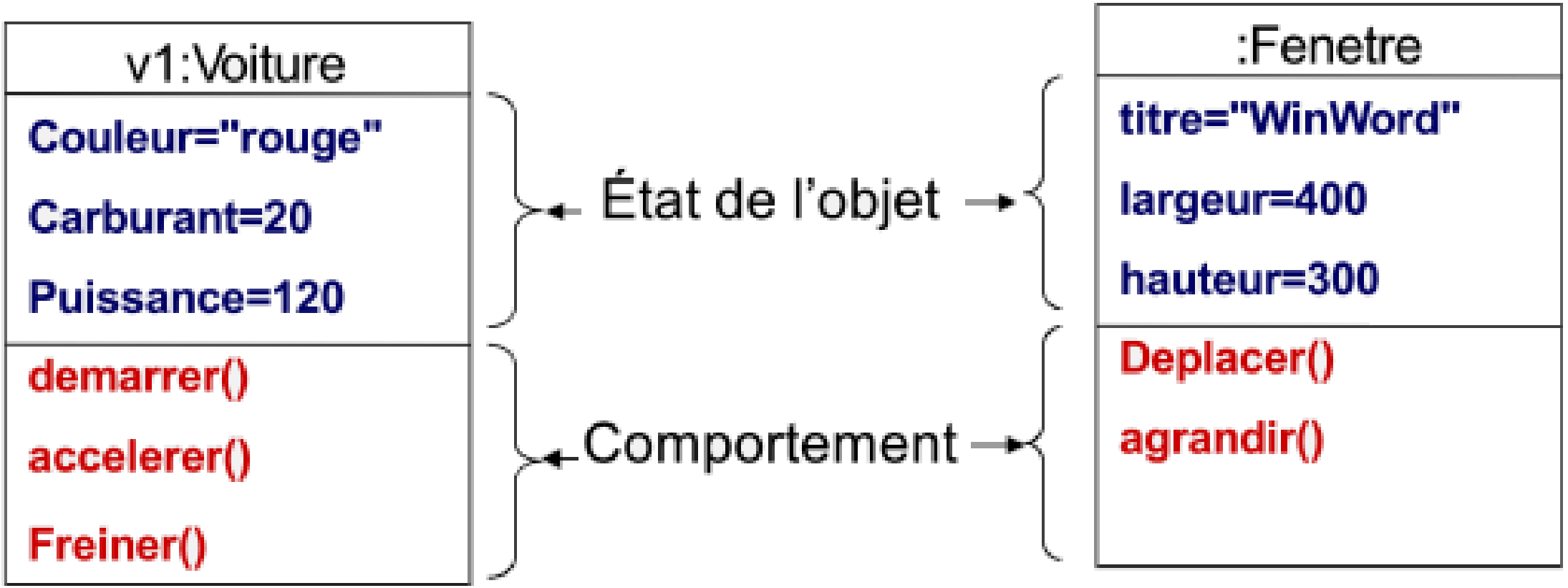
- **Le concept de l’objet** (ou définition/structure)
  - Permet d’indiquer ce qui compose un objet, c'est-à dire quelles sont ses propriétés, ses actions…
- **L’instance d’un objet**
  - C’est la création réelle de l’objet : *Objet Chaise*
  - En fonction de sa définition : *4 pieds, bleu…*
  - Il peut y avoir **plusieurs instances** : *Plusieurs chaises, de couleurs différentes, matériaux différents…*

Qu'est-ce qu'un objet en programmation?

- Un objet est une structure informatique définie par un **état** et un **comportement**.
  - L'**état** regroupe les **valeurs instantanées** de tous les **attributs de l'objet**. Il peut changer dans le temps.
  - Le **comportement** décrit les **actions** et les réactions de l'objet. Autrement dit le comportement est défini par **les opérations que l'objet peut effectuer**. Généralement, c'est le comportement qui modifie l'état de l'objet.

Exemple

Objet = état + comportement



Il s'agit ici d'un diagramme d'objet

Identité d'un objet

- En plus de son état, un objet possède une **identité** qui caractérise son existence propre.
- Cette identité s'appelle également **référence** ou **handle** de l'objet
- En terme informatique de bas niveau, l'identité d'un objet représente son **adresse mémoire**.
- Deux objets **ne peuvent pas avoir la même identité**:
  c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire

Résumé

- La **POO** est un **paradigme de programmation** basé sur la manipulation d'**objets**, représentant des entités ou concepts du monde réel.
- Elle **découpe les applications** complexes en **objets**, améliorant ainsi la maintenabilité et favorisant la réutilisation du code.
- **Concepts clés** : Accessibilité, Encapsulation, Héritage, Polymorphisme, Abstraction, Interfaces, Fonctions Anonymes, Généricité.
- Un objet combine **état** (propriétés actuelles) et **comportement** (actions possibles), avec une **identité unique** (adresse mémoire).

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Définition de Classes

Qu’est-ce qu’une Classe ?

Un **Classe** (`class`) permet de regrouper tous les éléments qui représenteront un Objet : ses **attributs** et ses **méthodes**

On dit qu'une classe représente le **concept** de l'objet.

Dans les langages fortement typés, **la création d'une classe** aboutira à la création d'un **nouveau Type** qu'on utilisera pour référencer les objets avec des variables.

Instanciation

- Les objets qui sont **définis à partir** d'une classe **appartiennent à celle-ci**.
- Ce processus s'appelle l'**Instanciation**
- On passe du **concept** (classe) à l'objet **réel** (instance/objet)
- La **classe est unique** mais les **objets** qui en **dérivent** peuvent être nombreux

Main

Nous avons déjà pu voir une Classe dans le code que nous avons utilisé précédemment qui a été généré par IntelliJ, la classe **Main**.

```
package org.example;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Syntaxe de Main

La class **Main** est une classe particulière car elle contient la méthode « **main( )** » qui est le **point d'entrée de notre application**

- Elle fonctionne comme toutes les classes
- La classe Program peut **faire des actions**, par exemple la **méthode main()** en est une
- Notez la présence des **accolades {}** qui **délimitent la classe** (le bloc d'instructions de celle-ci)
- Les **noms des classes** comme des méthodes s'écrivent en **PascalCase**.
  Exemple: `MaNouvelleClasse`

Arborescence de Projet Java

- Un **projet Java** est organisé **hiérarchiquement** dans **un système de fichiers**. Cette organisation standard facilite la gestion et la compilation du code.
- Les **dossiers** créés dans certaines correspondront à des **Packages** Java.
- Un **Package Java** regroupera un ensemble de fichier `.java` dans lequel des **types javas** (Classes, Enums, Interfaces, ...) seront **définis**.
  Il faudra ajouter dans chaque fichier le nom du package associé.

```
package org.exemple.monpackage;
```

Arborescence de Projet Java

```
projet-java/                          -> Dossier du projet
├── src/                              -> Code source
│   ├── main/                         -> Partie principale, l'application en elle-même
│   │   └── java/                     -> Pur les sources Java, autrement: scala, groovy, kotlin
│   │       └── org.exemple/          -> Package principal de la solution, ex: net.google, org.openai
│   │           ├── monpackage/       -> Sous-package
│   │           │   ├── MonEnum.java  -> Un Enum
│   │           │   └── MaClasse.java -> Une Classe
│   │           └── Main.java         -> La Classe contenant le "psvm"
│   └── test/                         -> Partie tests unitaires
├── target/                           -> Fichiers Compilés
└── pom.xml                           -> Fichier projet Maven
```

Si on veut utiliser MaClasse dans le Main, il faudra l'importer

```
import org.exemple.monpackage.MaClasse;
```

Création de Classe en Java

```java
package org.exemple.monpackage;

// Définition de la classe
public class MaClasse {
    // Bloc de classe où seront définit les Membres de la classe
    // Exemple :
    // - Attribut
    private int attribut;
    // - Constructeur
    public MaClasse(...) { ... }
    // - Méthodes
    public void methode() { ... }
}
```

La notion de visibilité/accessibilité

L'**indicateur de visibilité** est un **mot clé** qui sert à indiquer **depuis où** on peut **accéder** à l'**élément** qui le suit.

Visibilité	Description	Classe	Membres de classes
public	Accès non restreint	✓	✓
private	Accès uniquement depuis la même classe	✗	✓
protected	Accès depuis la même classe ou depuis une classe dérivée (cf héritage)	✗	✓
pas de mot clé	Accès restreint au même package (aussi appelé package-private)	✓	✓

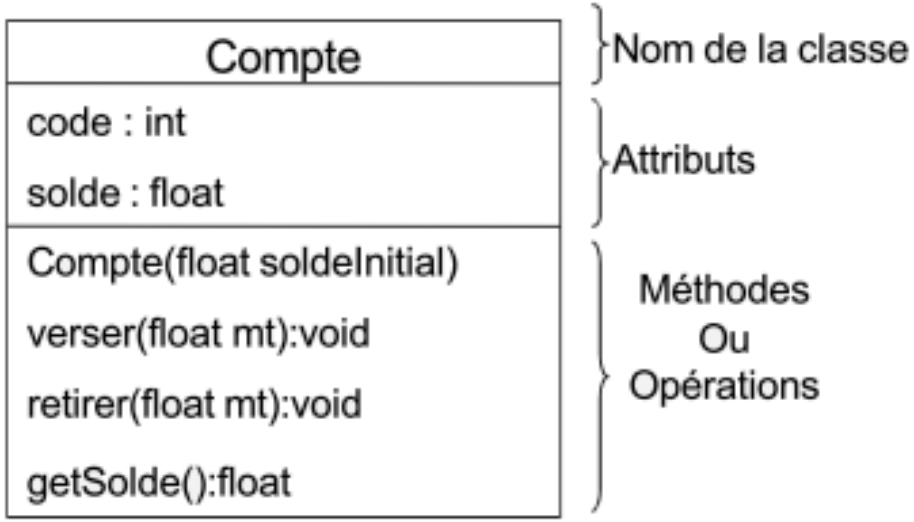
Les membres de la classe sont les attributs, constructeurs et méthodes

Elements d'une classe

Élément	Caractéristiques	Détails
Attributs : Variables d'instance	- Nom - Type - Valeur initiale (optionnelle)	**État** de l'objet
Méthodes : Fonctions liées à l'instance	**Signature** : - Nom - Type de retour - Paramètres	**Comportement** de l'objet
Constructeurs	- Pas de type de retour - Même nom que la classe - Paramètres	Appelés à la **création** de l'objet
Destructeur	Rarement utilisé, varie selon les langages	Méthode particulière appelée par le **Garbage Collector** à la **suppression**

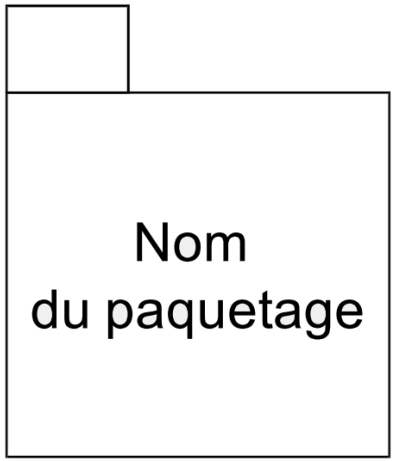
Représentation UML d'une classe

- Une classe est représenté par un rectangle à 3 compartiments :
  - Un compartiment qui contient le nom de la classe
  - Un compartiment qui contient la déclaration des attributs
  - Un compartiment qui contient les méthodes



Les classes sont stockées dans des packages

- Les packages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de la modélisation
- Chaque package est représenté graphiquement par un dossier
- Les packages divisent et organisent les modèles de la même manière que les dossier organisent le système de fichier



Les Attributs

Les **attributs** sont un **ensemble de variables** permettant de définir les caractéristiques de notre objet (aussi appelés **variables d'instance**). Ils doivent être déclarés par convention **au début de notre classe**.

- **Tous les types de variables sont utilisables pour la déclaration des attributs y compris des objets** (int, float, String, List<>, Voiture, Personne, etc.)
- Ils se déclarent comme suit et **peuvent être initialisés** ou non en fonction des besoins de votre application

```
private String model;                    private String model = "Tesla";
```

Le Constructeur

Maintenant que notre **concept de Voiture** (**class**) a des **attributs**, il nous faut un outil pour pouvoir créer **des nouvelles voitures spécifiques** (**instances/objets**), on parle de **construction**.

- Cet outil s'appelle donc le **constructeur**, il définit la manière de **créer une nouvelle instance**.
- Il est **similaire à une fonction** et **prend des paramètres** en entrée.
- Lors de son **appel**, il faut utiliser le mot-clé `new` (instanciation/construction d'un **nouvel** objet/instance).

Le Constructeur

Voici la syntaxe d'un constructeur en Java pour notre classe Voiture :

```
public class Voiture {
    private String model;
    private String couleur;
    private int reservoir;
    private int autonomie;

    public Voiture(String model, String couleur, int reservoir, int autonomie) {
        this.model = model;
        this.couleur = couleur;
        this.reservoir = reservoir;
        this.autonomie = autonomie;
    }
}
```

Mot-clé this

Lorsque l'on génère le constructeur, on utilise souvent le **mot-clé this** pour référencer l'instance courante.

```
public Voiture(String param_model, String couleur, int reservoir, int autonomie) {
    model = param_model;
    setCouleur(couleur); // setter
    this.reservoir = reservoir;
    this.autonomie = autonomie;
}
```

Ce mot-clé représente **l'instance sur laquelle on travaille**, dans le constructeur il s'agit donc de **celle que l'on construit**. Il est le plus souvent **facultatif** si l'on respecte les conventions de nommage.

Constructeur par défaut (sans paramètres)

Lorsque l'on crée une **nouvelle classe vide**, on **pourrait penser** qu'il est **impossible de l'instancier si aucun constructeur n'est défini**.

- En réalité, **il existe un constructeur vide par défaut** (implicite/invisible) dans toute classe qui **n'a pas encore de constructeur**. Voici à quoi il correspond :

```
public class Voiture {
    public Voiture() { }
}
```

Dès que l'on ajoute un constructeur, ce constructeur **disparaît**.

Vue d'ensemble de notre classe Voiture à présent

```java
public class Voiture {
    // Attributs
    private String model;
    private String couleur;
    private int reservoir;
    private int autonomie;

    // Constructeurs
    public Voiture() { }
    public Voiture(String model, String couleur, int reservoir, int autonomie) {
        this.model = model;
        this.couleur = couleur;
        this.reservoir = reservoir;
        this.autonomie = autonomie;
    }
```

L’instanciation d’un objet

Maintenant que notre **classe Voiture** a des **attributs** et des **constructeurs**, nous allons pouvoir créer des voitures. Voici la syntaxe pour **l’instanciation d’un objet** en Java (utilisation du constructeur sans paramètres) :

```
// type nomVariable = new Classe();
Voiture autoDeGuillaume = new Voiture();
```

- Attention, la classe Voiture n’est **pas reconnue** tant que nous n'avons pas fait l'**import du package**.

```
import org.example.Voiture;
```

L’instanciation d’un objet avec paramètres

Instantiation avec l'autre constructeur que nous avons défini :

```
// Voiture(String model, String couleur, int reservoir, int autonomie)
Voiture autoDeGuillaume = new Voiture("Fiat Multipla", "Rouge", 63, 733);
```

Ici, les attributs auront les valeurs définies à l'appel du constructeur (arguments).

Cependant si vous avez défini un comportement spécifique dans le constructeur ou les méthodes, ces valeurs peuvent changer.

La modification d'un objet instancié

Maintenant que nous avons instancié notre objet Voiture, nous pouvons **accéder à ses attributs** via l'**auto complétion** de l'IDE (Ctrl+Espace ou Alt+Enter le plus souvent).
Il suffit ensuite de les **assigner** pour les modifier pour notre instance depuis la variable autoDeGuillaume.

```
autoDeGuillaume.model = "Clio";
autoDeGuillaume.couleur = "Noir";
autoDeGuillaume.reservoir = 45;
autoDeGuillaume.autonomie = 900;
```

Affichage de notre objet Voiture dans la console

Maintenant que nous avons instancié notre objet Voiture, nous pouvons l'utiliser. Essayons de l'**afficher dans la console** :

```
System.out.println(autoDeGuillaume); // résultat : Voiture@hashcode
```

Ce résultat est la **représentation textuelle de l'objet**. Nous verrons comment le changer par la suite (cf `toString()`).
Voici comment nous pourrions l'afficher :

```
System.out.println("Notre première voiture est une " + autoDeGuillaume.model
+ " de couleur " + autoDeGuillaume.couleur);
System.out.println("Elle a un réservoir de " + autoDeGuillaume.reservoir
+ " litres pour une autonomie de " + autoDeGuillaume.autonomie + " km.");
```

Pour les attributs non définis, ils auront leur valeur par défaut.

Les Méthodes d'une classe

Une **méthode** est une **fonction associée à une classe**. Elle est définie à l'intérieur du bloc de la classe et peut accéder aux **attributs**, **propriétés** et autres **méthodes** de la classe.

Pour faciliter l'affichage de nos objets **Voiture**, nous pouvons encapsuler le code précédent dans une **méthode**, ce qui permet de le réutiliser plus facilement.

```
public void afficher() {
    System.out.println("Notre première voiture est une " + this.model + " de couleur " + this.couleur);
    System.out.println("Elle a un réservoir de " + this.reservoir
        + " litres pour une autonomie de " + this.autonomie + " km.");
}
```

Les Méthodes d'une classe

Il est possible d'ajouter autant de méthodes que nécessaire, leur **nom** indiquant **leur utilité** pour la classe.

Créons ensemble une méthode `demarrer()`.

- Ajoutons une propriété booléenne `demaree` pour indiquer si le moteur est en marche. Nous utiliserons celle-ci pour vérifier si le moteur tourne avant de le démarrer.
- **Si** la voiture est **éteinte**, nous afficherons un message dans la console pour informer l'utilisateur que **la voiture démarre**.
- **Sinon**, nous indiquerons que **le moteur tourne déjà**.

Les Méthodes d'une classe

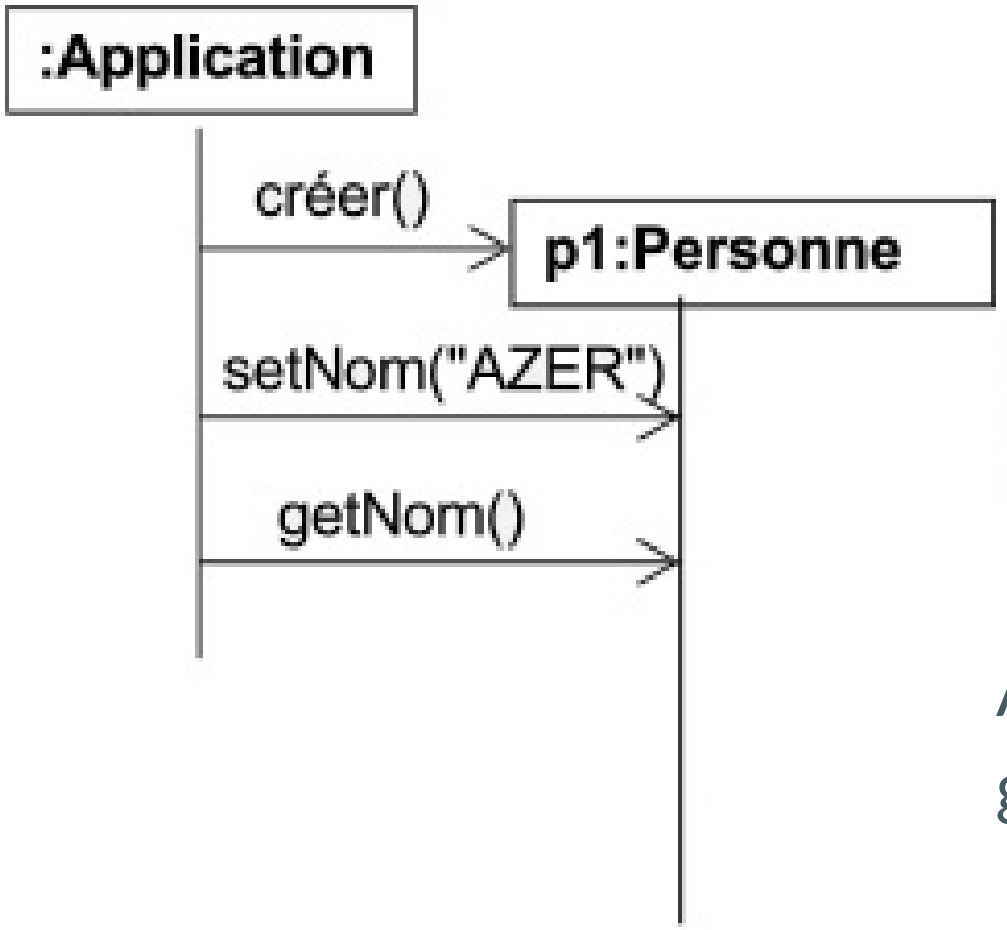
Voici la méthode `demarrer()` :

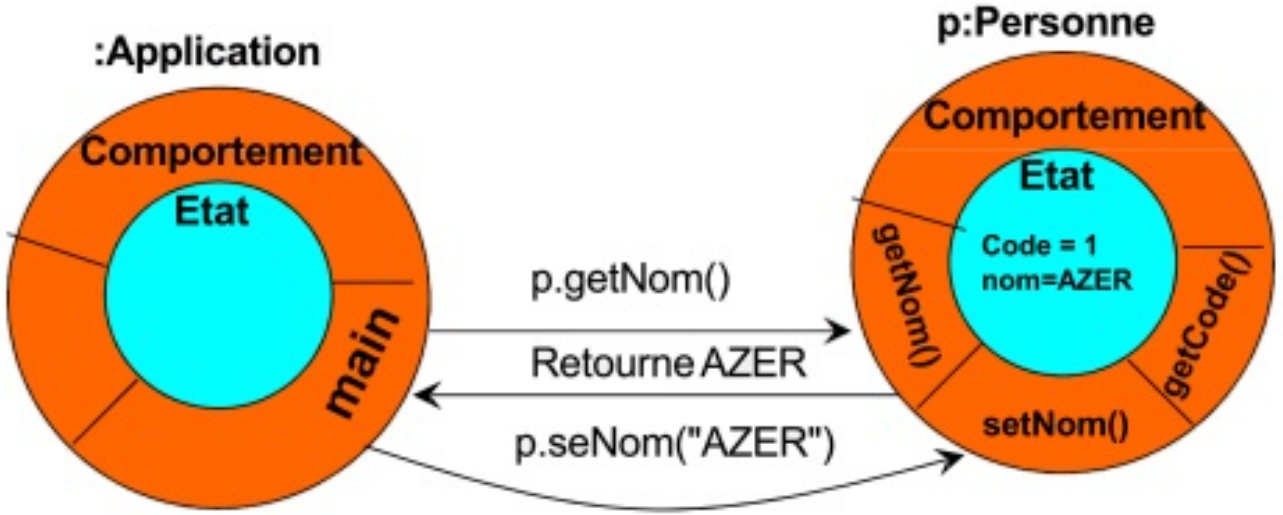
```
public boolean demarrer() {
    if (!demaree) {
        demaree = true;
        System.out.println("La voiture est démarrée... le moteur tourne !");
    } else {
        System.out.println("La voiture est déjà démarrée !");
    }
    return demaree;
}
```

Les Getters et Setters

- Le principe de l'**encapsulation** en POO recommande de laisser les **attributs** en **privé** (**private**), ce qui signifie qu'ils sont **uniquement accessibles depuis l'intérieur de cette classe**.
- En Java, on y accède par des **méthodes publiques** toujours nommées `getAttribut()` et `setAttribut(valeur)`.
- Les **getters** et **setters** permettent de **lire** et **modifier** les attributs tout en conservant le contrôle sur ces opérations.

Encapsulation





Avec ce schéma, on peut voir que via les getters/setters, l'état est "**protégé**"

Les Getters et Setters

Voici la syntaxe pour un **getter** et un **setter** en Java :

```java
public class Voiture {
    private String model;

    // Getter -> Aucun paramètre + type de retour de l'attribut
    public String getModel() {
        return model;
    }

    // Setter -> Un seul paramètre et aucun type de retour
    public void setModel(String model) {
        this.model = model;
    }
}
```

Les Getters et Setters

Si l'on veut **définir un comportement spécifique** à la **modification** (**setter**) ou à la **récupération** (**getter**) d'un attribut, il faudra donc **changer le bloc d'instruction** du set ou du get en fonction de nos besoins.

```
public class Voiture {
    private double poids;

    public double getPoids() {
        System.out.println("Poids récupéré: " + poids);
        return poids;
    }

    public void setPoids(double poids) {
        if (poids <= 0) {
            System.out.println("Valeur invalide! "
                + "Poids fixé à 100 kg.");
            this.poids = 100;
        } else {
            this.poids = poids;
        }
    }
}
```

Attribut en lecture seule

Si l'on veut **empêcher la modification d'un attribut**, on peut décider de **ne pas définir de setter**, rendant ainsi l'attribut **en lecture seule**.

```
public class Voiture {
    private String model;

    // Getter uniquement, pas de setter
    public String getModel() {
        return model;
    }
}
```

Getters composés (lecture seule)

Lorsque l'on veut créer un getter qui **dépend d'autres attributs**, on fera en sorte qu'il **retourne une valeur calculée** à partir de ces attributs.

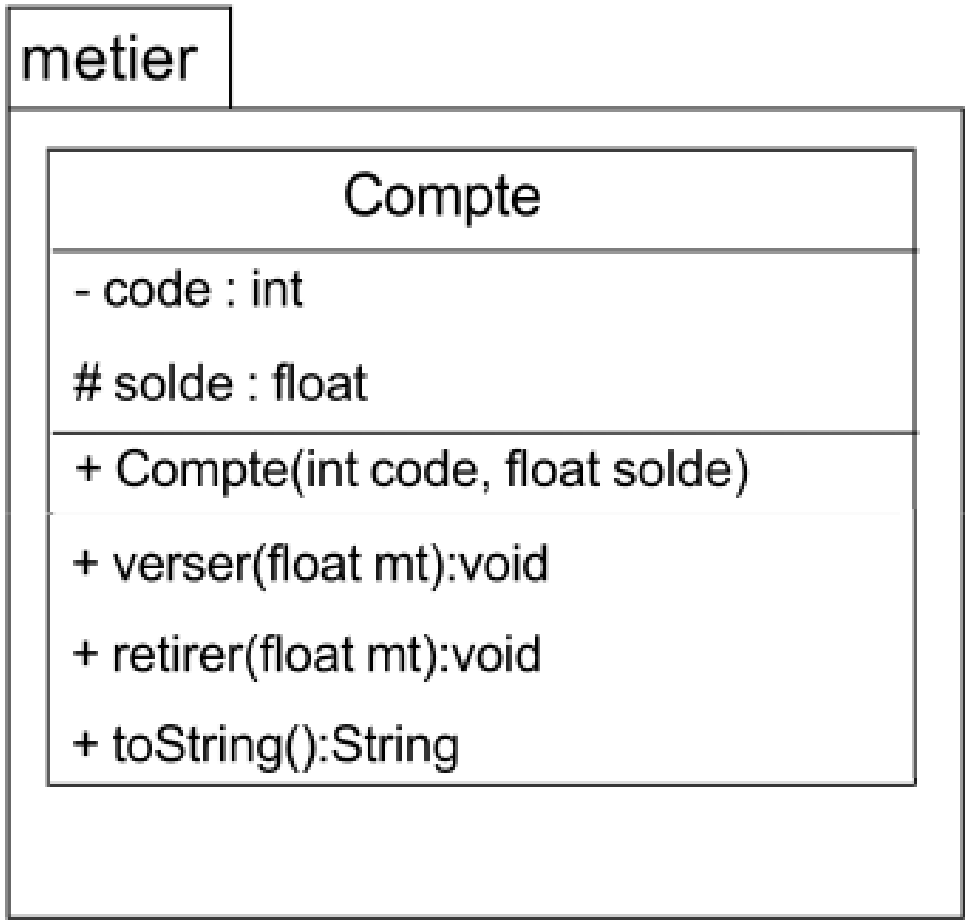
```
public class Personne {
    private String nom;
    private String prenom;

    public String getNomComplet() {
        return nom + " " + prenom;
    }
}
```

Commentaires de documentation dans une classe

Le **commentaire de documentation** en Java se fait avant **un membre d'une classe**, **une classe** ou d'autres éléments. Il permet d'**expliquer l'élément en question** et cette explication peut être affichée par les IDEs lors du survol de l'élément.

```
/**
 * Effectue l'addition de deux entiers.
 *
 * @param a Premier entier
 * @param b Deuxième entier
 * @return La somme des deux entiers
 */
public int add(int a, int b) {
    return a + b;
}
```

Exemple d'implémentation d'une classe




```java
package metier;

public class Compte {
  // Attributs
  private int code;
  protected float solde;
  // Constructeur
  public Compte( int c,float s){
    code=c;
    solde=s;
  }
  // Méthode pour verser un montant
  public void verser(float mt){
    solde+=mt;
  }
  // Méthode pour retirer un montant
  public void retirer(float mt){
    solde-=mt;
  }
  // Une méthode qui retourne l'état du compte
  public String toString(){
    return(" Code = "+code+" Solde = "+solde)
  }
}
```

Notion de `static`

Il est possible via l'utilisation du mot clé `static` de **créer des membres** (attributs et méthodes) qui seront **liés à la classe** et non aux instances.

```
private static int nombreDeVoitures = 0;
public static int getNombreDeVoitures() { return nombreDeVoitures; }
```

Ici, nous avons un **attribut de classe** et non d'instance, il est **partagé entre toutes les instances** et accessible directement depuis la classe avec cette syntaxe :

```
System.out.println("Total : " + Voiture.getNombreDeVoitures());
```

Notion de `static`

Un autre exemple avec des **méthodes** `static` (méthode de classe) :

```
public static void afficherTotalVoitures() {
    System.out.println("Voitures créées avec le constructeur : " + nombreDeVoitures);
}

public static void afficherVoituresParlantes() {
    System.out.println("Les voitures qui parlent, ça n'existe pas...");
}
```

Elles serviront en général à travailler avec des notions relatives à toutes les voitures et non à une en particulier.

Notion de `static`

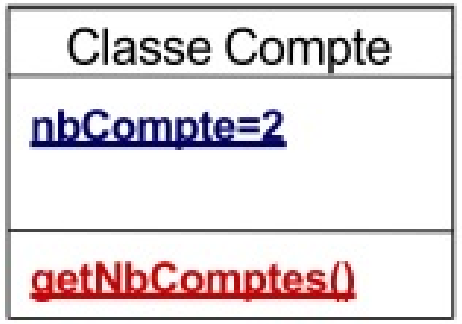
Utilisations des `static` dans un constructeur :

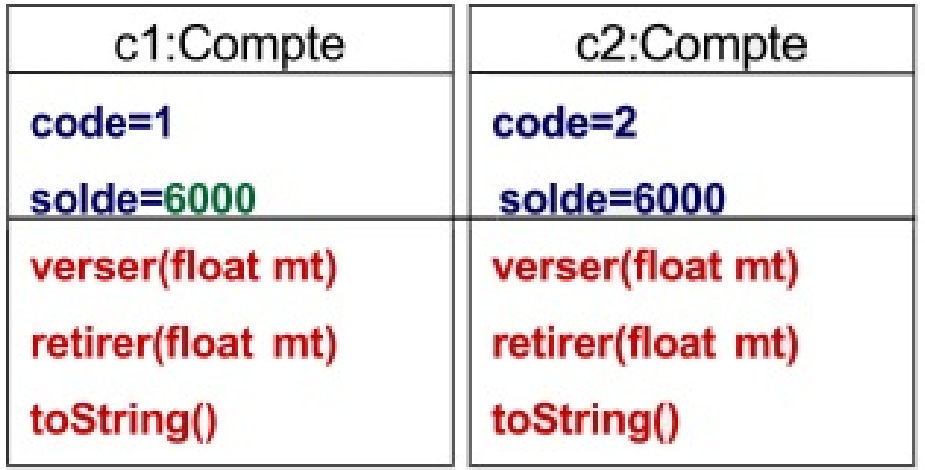
```
public Voiture() {
    nombreDeVoitures++;
    afficherTotalVoitures();
}
```

/!\ Attention, pour des raisons évidentes, un constructeur **ne peut pas être** `static`, il permet de créer **une** instance

Autre Exemple

```
package test;
import metier.Compte;

public class Application  {

  public static void main(String[] args) {
    Compte c1=new  Compte(5000);
    Compte c2=new Compte(6000);
    c1.verser(3000);
    c1.retirer(2000);
    System.out.println(c1.toString());
    System.out.println(Compte.nbComptes)
    System.out.println(c1.getNbComptes())
  }
}
```





Constructeur dépendant d'un autre constructeur

À l'aide de `this(params)`, on peut préciser qu'à l'appel d'un constructeur, on en appelle un autre au début, cela permet d'éviter les répétitions :

```
public Voiture() {
    nombreDeVoitures++;
    afficherTotalVoitures();
}

public Voiture(String model, String couleur, int reservoir, int autonomie) {
    this(); // réutilise le premier constructeur
    this.model = model;
    this.couleur = couleur;
    this.reservoir = reservoir;
    this.autonomie = autonomie;
}

public Voiture(int reservoir, int autonomie) {
    this("Fiat Multipla", "Rouge", reservoir, autonomie); // réutilise le deuxième constructeur avec des valeurs prédéfinies
} // attention aux conflits (2 constructeurs avec le même nombre de paramètres)
```

03 Le Polymorphisme

Rappel sur les signatures

```
public boolean ajouterVoiture(Voiture voiture)
```

- La **signature** de la méthode nous renseigne sur le **nom**, les **paramètres**, et le **type de retour**.
- Lorsque l'on parle de méthodes, le mot **clé de visibilité / accessibilité** vient s'ajouter.
- **2 méthodes** portant le même nom mais avec des **paramètres** et **types de retour différents** sont bien **2 éléments distincts**. C'est **un premier cas de polymorphisme** (polymorphisme paramétrique).

Le concept de Polymorphisme

- Le mot **polymorphisme** suggère qu'un élément **défini par son nom** (**identificateur/symbol**) possède **plusieurs formes**.
- Il a la capacité de faire **une même action** avec **différents types d'intervenants**.
- En POO, ce concept s'applique principalement aux **méthodes**, mais aussi aux **attributs** et aux **constructeurs**.

Les types de Polymorphisme

Il y a plusieurs types de **polymorphisme** en **POO**:

- Le polymorphisme **avec signatures différentes**:
  - par **Surcharge / Overload** (aussi nommé « **ad hoc** »)
  - **Paramétrique**
- Le polymorphisme de l'**Héritage**:
  - par **Masquage / Shadowing**
  - par **Substitution / Override**

Le polymorphisme par surcharge / overloading (ad hoc)

C'est le cas où l'on utilise le même nom de méthode mais avec **un nombre de paramètres différents**.

Prenons le cas d'une classe `Concessionnaire` possédant une `List<Voiture>` dans laquelle **on ajoutera des voitures**:

- Ici notre méthode prend un objet en paramètre:

```
public boolean ajouterVoiture(Voiture voiture)
```

- Ici notre méthode prend 3 paramètres (on instanciera la voiture):

```
public boolean ajouterVoiture(String model, String couleur, int reservoir, int autonomie)
```

Le polymorphisme paramétrique

C'est le cas où l'on utilise le même nom de méthode, le même nombre de paramètres mais avec **une signature différente au niveau des types**.

- Ici notre méthode est signée `int`:

```
public static int additionner(int a, int b)
```

- Ici notre méthode est signée `String`:

```
public static String additionner(String a, String b)
```

Le polymorphisme de l'Héritage

Les polymorphismes par **Masquage** et par **Substitution / Override** interviennent **dans la notion d'Héritage** *(chapitre suivant)*.

Ils permettent de faire de la **spécialisation** sur nos **méthodes**.

04 Définition de l’héritage

Le concept de l'héritage

L'héritage est un mécanisme central en POO

- Une classe peut **hériter** d'une **autre classe**, acquérant ainsi **les membres** (méthodes, attributs, constructeurs) de cette dernière, on dit aussi qu'elle **dérive** de l'autre classe.
- On parle alors de **classe fille/enfant** (spécialisée) et de **classe mère/parent** (générale).
- Pour **réaliser un héritage** en Java, on utilise le mot-clé `extends` après le nom de la classe que l'on crée, suivi de la classe dont on souhaite hériter:

```
public class Homme extends Mammifere {...}
```

Exemples réels

Pour comprendre la notion d'héritage, rien de tel que quelques exemples basés sur le réel:

- `Chien` est une classe dérivée de `Mammifere`
- `Mammifere` est une classe dérivée de `Animal`
- `Animal` est une classe dérivée de `ÊtreVivant`

Chaque **parent** est plus **général** que son **enfant**.
Et inversement, chaque **enfant** est plus **spécialisé** que son **parent**.
L'**enfant** aura donc **les caractéristiques du parent** auxquelles s'ajoutent ses **spécificités**.

Non-multiplicité de l'héritage

Il est possible pour un **parent** d'avoir **plusieurs enfants**. Par contre, **l'inverse est impossible**, un **enfant ne peut pas** avoir **plusieurs parents** -> **L'héritage multiple est interdit en Java**.

On peut définir une sorte de **hiérarchie** entre les objets, un peu comme un **arbre généalogique**.

Mot clé super

- Lors d'un **héritage**, il est possible d'**accéder aux attributs et aux méthodes de la classe mère**.
  Si l'on souhaite **accéder à un membre** de la **classe mère** pour **s'en servir dans la classe enfant**, on doit utiliser le mot-clé `super`.
  Exemples: `super.attribut`, `super.methode()`.
- Le mot-clé `super()` est également utilisé au niveau d'un **constructeur** pour faire **appel au constructeur de la classe parent**:

```
public Mammifere(String nom, int age, String genre) {
    super(nom, age);
```

Les polymorphismes de l'Héritage

Les polymorphismes par **Masquage** et par **Substitution / Override** permettent de faire de la **spécialisation** sur nos **méthodes**.

En effet, si on veut **modifier** ou **remplacer** le **comportement de méthodes** d'une **classe mère** dans une **classe fille**, cela sera possible avec ces concepts.

Ainsi, ces méthodes auront **plusieurs formes** en fonction du **type de l'instance** que l'on utilisera.

Savoir quand utiliser les mots-clés override et new

Les polymorphismes de l'Héritage

Exemple:

Prenons une classe `Mammifere` qui a la méthode `seDeplacer()`.

Tous les **Mammifères** se déplacent, mais de manière **spécifique** (nager, voler, marcher, sauter, ...).

Ce type de polymorphisme permet de définir des **formes différentes** pour `seDeplacer()` en fonction du mammifère.

Un `Dauphin` se déplace **différemment** d'un `Humain`, pourtant ce sont tous les deux des `Mammifères`.

Masquage / Shadowing (polymorphisme d'héritage)

Lors du **Masquage**, on aura des **méthodes** dans les classes **mère et fille** de **même nom**, mais celle de la fille viendra **remplacer** celle de la mère. En Java, il n'est **PAS RECOMMANDÉ** dans une majorité des cas.

```java
public class Animal {
    private String nom;
    private boolean estVivant;

    public Animal(String nom, boolean estVivant) {
        this.nom = nom;
        this.estVivant = estVivant;
    }

    public void respirer() {
        System.out.println("L'animal respire");
    }
}
```

```java
public class Mammifere extends Animal {
    private String genre;

    public Mammifere(String nom, boolean estVivant, String genre) {
        super(nom, estVivant);
        this.genre = genre;
    }

    @Override
    public void respirer() {
        System.out.println("Le mammifere respire");
    }
}
```

Substitution / Override (polymorphisme d'héritage)

Lors de la **Substitution**, on aura des **méthodes** dans les classes **mère et fille** de **même nom**, mais celle de la fille viendra **redéfinir** celle de la mère tout en ayant la possibilité de la réutiliser. On utilise les mots-clés `@Override`.

```java
public class Animal {
    private String nom;
    private boolean estVivant;

    public Animal(String nom, boolean estVivant) {
        this.nom = nom;
        this.estVivant = estVivant;
    }

    public void respirer() {
        System.out.println("L'animal respire");
    }
}
```

```java
public class Mammifere extends Animal {
    private String genre;

    public Mammifere(String nom, boolean estVivant, String genre) {
        super(nom, estVivant);
        this.genre = genre;
    }

    @Override
    public void respirer() {
        super.respirer(); // appeler une méthode du parent
        System.out.println("Le mammifere respire");
    }
}
```

La classe Object

Chaque classe en Java **hérite automatiquement** d'une classe appelée `Object`.

Cette classe comporte **une série de méthodes** qui seront ainsi automatiquement héritées par les classes enfants.

- **toString** = représentation textuelle de l'objet
- **equals** = comparaison d'égalité
- **getClass** = récupération du type
- **hashCode** = Hash de l'objet
- **clone** = clone de l'objet avec attributs identiques

La méthode toString()

L'exemple le plus courant est sans doute celui de l'héritage de la méthode `toString()` qui est utilisée pour récupérer la représentation textuelle de l'objet.

```java
// Animal
@Override
public String toString() {
    return this.getClass().getSimpleName()
            + " : Nom = " + nom
            + ", EstVivant = " + estVivant;
}
```

```java
// Mammifere
@Override
public String toString() {
    return super.toString()
            + ", Genre = " + genre;
}
```

`getClass()` est une des méthodes de Object, elle permet de récupérer les informations de la classe actuelle

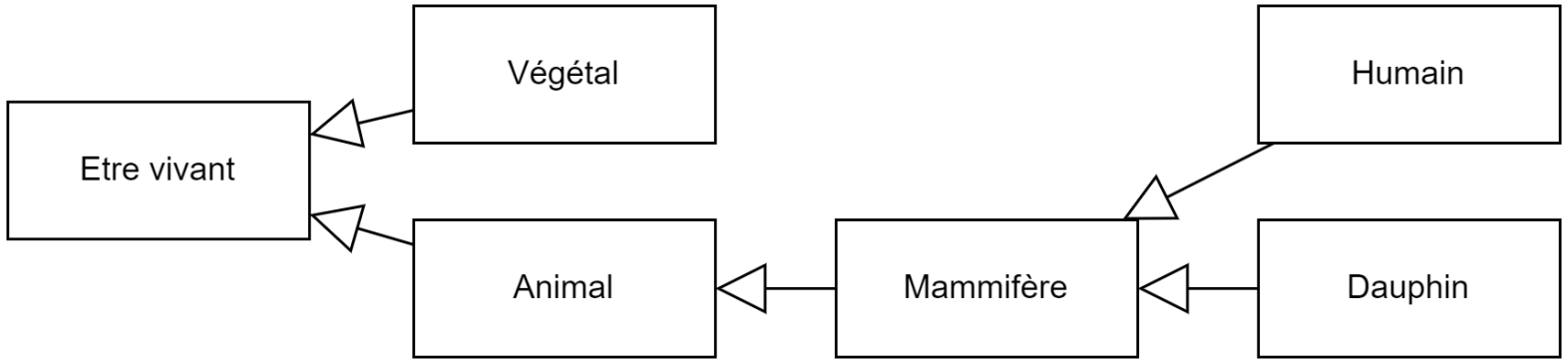
Les classes abstraites (abstract)

- Une classe `abstract` est une classe particulière qui **ne peut pas être instanciée**.
- **Impossible d'utiliser les constructeurs et l'opérateur new**.
- Pour être **utilisables**, les **classes abstraites** doivent être **héritées** et leurs méthodes abstraites redéfinies.

- Elles servent à **représenter** un **concept** ou un **objet** qui **n'a pas de sens tel quel car trop général**, ce seront ses **spécialisations / enfants** qui seront **instanciées** (possiblement indirectement).

Les classes abstraites (abstract)

Dans notre exemple précédent, nous pourrions avoir `EtreVivant`, `Vegetal`, `Animal` et `Mammifere` en abstrait car par la présence de leurs spécialisations, leur **instanciation devient incohérente**, ***abstraite***.



Autre exemple, si nous avions supprimé les classes `Humain` et `Dauphin` et ajouté un attribut `Especе` à `Mammifere`, celui-ci pourrait ne plus être abstrait.

Les classes et les méthodes abstraites (abstract)

De la même façon, une **méthode abstraite** est une méthode qui ne contient **pas d'implémentation**.

- Elle n'a **pas de corps** (pas de bloc de code).
- Une méthode `abstract` sera toujours dans une classe `abstract`.
- Pour être utilisables, les méthodes `abstract` doivent être redéfinies avec un `override`.

```java
public abstract class Mammifere {
    // Méthode abstraite sans implémentation
    public abstract void seDeplacer();
}
```

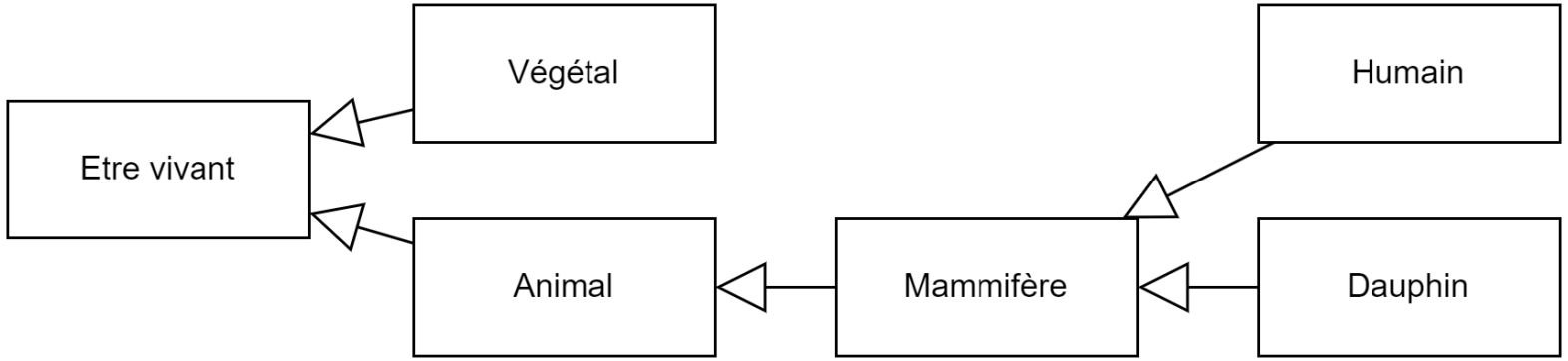
```java
public class Dauphin extends Mammifere {
    @Override
    public void seDeplacer() {
        System.out.println("Le dauphin nage.");
    }
}
```

Les classes et les méthodes finales (final)

Nous utilisons le mot-clé `final` quand une **classe** ne doit **plus être héritée** ou qu'une **méthode** ne doit **plus être substituée/override**.

- La classe sera la **dernière** de la lignée.
- La méthode ne pourra **plus être substituée**.

Dans notre exemple, on pourrait avoir `Humain` et `Dauphin` en `final`.



Sealed

A partir de Java 17, le mot-clé `sealed` a été introduit.

- Il est utilisé pour **restreindre quelles autres classes ou interfaces peuvent étendre ou implémenter une classe ou une interface**.
- Cela permet de **contrôler** plus précisément la **hiérarchie** des classes et d'augmenter la sécurité et la maintenabilité du code.

Sealed

```
public sealed class Shape permits Circle, Square, Rectangle {
    // Classe scellée Shape
}
public final class Circle extends Shape {
    // Classe Circle doit être final car elle ne peut pas être étendue davantage
}
public final class Square extends Shape {
    // Classe Square doit être final car elle ne peut pas être étendue davantage
}
public non-sealed class Rectangle extends Shape {
    // Classe Rectangle peut être étendue car elle est marquée comme non-sealed
}
public class RoundedRectangle extends Rectangle {
    // Cela est permis car Rectangle est marqué comme non-sealed
}
```

Voici quelques points clés concernant l'utilisation de `sealed` :

1. **`sealed` classes** : Une classe scellée utilise le mot-clé `sealed` et doit indiquer les sous-classes autorisées via le mot-clé `permits`.
2. **`non-sealed` classes** : Une sous-classe d'une classe scellée peut être marquée comme `non-sealed` pour permettre à d'autres classes de l'étendre.
3. **`final` classes** : Une sous-classe d'une classe scellée peut être marquée comme `final`, empêchant ainsi toute autre extension.
4. **`permits` clause** : La clause `permits` est utilisée pour énumérer les classes spécifiques qui sont autorisées à étendre la classe scellée.

Type de variables et type d'instance

Si on reprend notre exemple, **un Dauphin EST un Animal**, on pourra alors faire :

```
Animal dph = new Dauphin();
```

Ici, la **variable** sera de **type** `Animal` mais pourra aussi **référencer** des **instances** de **classes dérivées** de `Animal`.

Autre exemple :

```
Animal[] animaux = {
    new Baleine(), new Dauphin(), new ChauveSouris(), new Pigeon(), new Humain()
};
```

Cast et Opérateur instanceof dans l'héritage

Précédemment, nous avons vu les **casts implicites** et **explicites** et l'opérateur `instanceof`. Ils prennent tout leur sens dans le cadre de l'héritage.

Si nous prenons l'exemple précédent avec le tableau, nous pourrons ainsi itérer sur le tableau puis **convertir chaque élément si besoin**.

```
for (Animal animal : animaux) {
    if (animal instanceof Dauphin) {
        Dauphin dauphin = (Dauphin) animal;
        System.out.println(dauphin.getClass().getSimpleName()
            + " => Cet Animal est bien un dauphin.");
        dauphin.nager();
    }
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Interfaces

Pourquoi les interfaces existent-elles ?

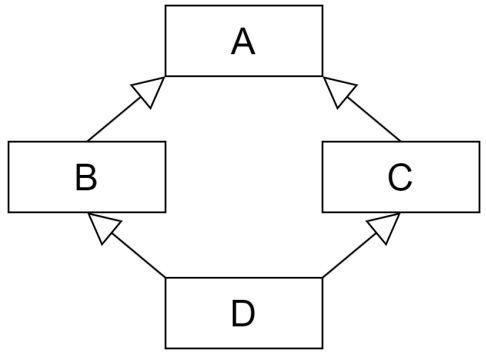
- Imaginons que nous cherchions à **regrouper** plusieurs **classes** qui ont un **comportement commun** sous **un même type**
- Nous pourrions être tenté d'utiliser l'**héritage**
- Cependant celui-ci n'est **valide** que lorsque l'**on peut dire**: "`ClasseB` **est un cas spécifique de** `ClasseA`"
- Il existe certains cas qui **ne correspondront pas** et où cette affirmation sera **invalide**

Exemple Concret

- Un `Avion` peut voler, il aura les méthodes `décoller()` et `atterrir()`
- Un `Oiseau` peut voler, il aura les **mêmes méthodes**
- Si nous voulions créer une liste d'**objets volants** pour les faire **décoller** successivement, il nous faudrait un type `Volant`
- Cependant, l'`Oiseau` est un `Animal` et l'`Avion` est une `Machine`
- On aurait donc besoin de pouvoir **hériter de plusieurs classes simultanément**, par **Héritage Multiple**
- Or, en Java, c'est **Impossible**
- Le Python le permet mais un problème complexe en résulte

Héritage en diamant impossible

- Java ne supporte pas l'**héritage multiple** pour éviter le problème de l'**héritage en diamant**.
- Si `B` et `C` **héritent** de `A` et `D` **hérite** de `B` et `C`, **quelle version** de `A` **doit être utilisée** par `D` ?



- Les interfaces offrent une **solution**, permettant à une classe d'**implémenter plusieurs interfaces**.

Pourquoi les interfaces existent-elles ?

- Les **interfaces** permettent de définir des **contrats** que les classes doivent respecter.
- Elles facilitent la **modularité** et la **réutilisabilité** du code.
- Elles permettent d'implémenter une forme de **polymorphisme** sans héritage multiple.

Qu'est-ce qu'une interface ?

- Une interface est un **type** en Java qui ne contient que des **méthodes abstraites**.
- Les classes qui implémentent une interface doivent **redéfinir** toutes ses méthodes.
- Les interfaces permettent de définir des **comportements communs** sans imposer une hiérarchie d'héritage.

Interfaces comme contrat

- Une interface définit un **contrat** : une classe qui l'implémente s'engage à fournir des implémentations pour toutes ses méthodes.
- Exemple : une interface `Volant` peut définir des méthodes `decoller` et `atterrir`.

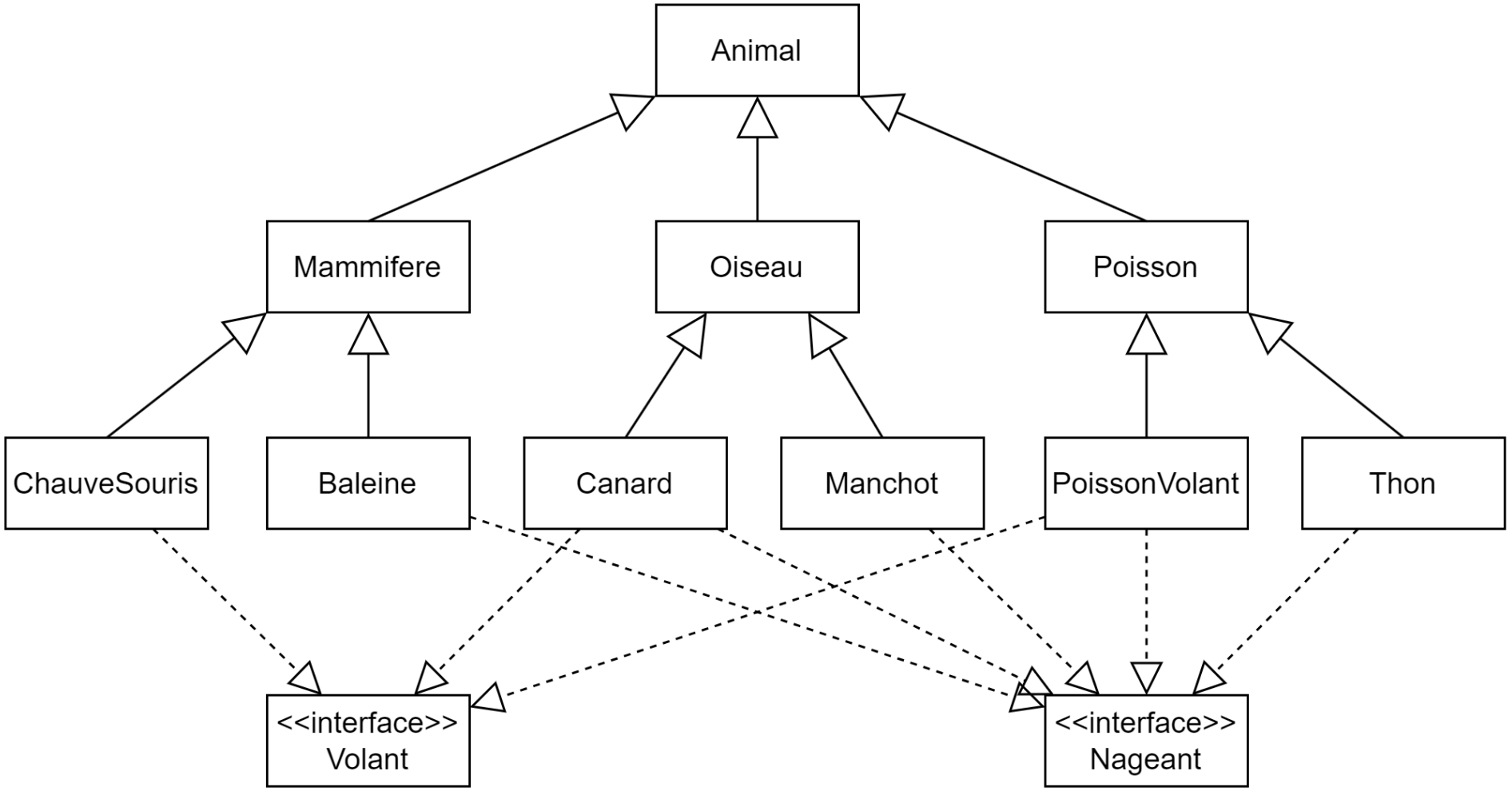
```
public interface Volant {
    void decoller();
    void atterrir();
}
```

Exemple de classes et interfaces

```java
public abstract class Animal {
    // Attributs et méthodes communs
}

public abstract class Mammifere extends Animal {
    // Attributs et méthodes spécifiques aux mammifères
}

public class Baleine extends Mammifere
                     implements Nageant {
    @Override
    public void nager() {
        System.out.println("La baleine nage.");
    }
}
```

```java
// Interfaces
public interface Volant {
    void decoller();
    void atterrir();
}

public interface Nageant {
    void nager();
}
```

Démonstration avec diverses classes et interfaces



Implémentation de 2 classes

```java
public class Canard extends Mammifere
                    implements Volant, Nageant {
    @Override
    public void decoller() {
        System.out.println("Le canard décolle.");
    }

    @Override
    public void atterrir() {
        System.out.println("Le canard atterrit.");
    }

    @Override
    public void nager() {
        System.out.println("Le canard nage.");
    }
}
```

```java
public class PoissonVolant extends Poisson
                           implements Volant, Nageant {
    @Override
    public void decoller() {
        System.out.println("Le poisson volant décolle.");
    }

    @Override
    public void atterrir() {
        System.out.println("Le poisson volant atterrit.");
    }

    @Override
    public void nager() {
        System.out.println("Le poisson volant nage.");
    }
}
```

Exemple d'utilisation

```
Animal[] zooDeLille = new Animal[] {
    new Baleine(),
    new Canard(),
    new Thon(),
    new PoissonVolant(),
    new ChauveSouris(),
    new PoissonVolant(),
    new Manchot(),
    new Manchot(),
};
```

```
for (Animal animal : zooDeLille) {
    System.out.println(animal);

    if (animal instanceof Poisson) {
        System.out.println("C'est un poisson!");
    }

    if (animal instanceof Volant animalVolant) {
        animalVolant.decoller();
        animalVolant.atterrir();
    }

    if (animal instanceof Nageant animalNageant) {
        animalNageant.nager();
    }
}
```

Résumé

- Les interfaces en Java offrent une manière de définir des **contrats** de **comportement** sans les contraintes de l'héritage multiple.
- Elles permettent d'implémenter le **polymorphisme** de manière **flexible et modulaire**.
- Utilisées correctement, elles améliorent la **maintenabilité** et la **réutilisabilité** du code.

Génériques

Introduction

La **généricité** en Java permet de définir des **classes**, des **interfaces** et des **méthodes** avec des **types paramétrés**.
Cela permet d'écrire du code plus flexible et réutilisable tout en garantissant la sécurité des types à la compilation.

On parle en général de **classes "Moule"** faites pour **accueillir** et **s'adapter** à d'**autres classes**.

L'exemple de plus connu est celui des **collections**, dans `List<T>` la **liste s'adapte au type interne** pour faire des opérations avec ce type.

1. Classes et Interfaces Génériques

Les **classes** et **interfaces** peuvent être définies avec des **paramètres de type**, ce qui permet de travailler avec des **types spécifiques** tout en maintenant une **structure générale**.

Ce **type ne pourra pas changer** dans le temps **lors de l'utilisation** de la classe, le **typage** reste **Fort**.

Exemple 1

Classe générique :

```java
public class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

Utilisation :

```java
Box<Integer> integerBox = new Box<>();
integerBox.set(123);
Integer value = integerBox.get();
System.out.println(value); // Affiche 123

Box<String> stringBox = new Box<>();
stringBox.set("Hello");
String text = stringBox.get();
System.out.println(text); // Affiche Hello
```

2. Méthodes Génériques

Les **méthodes** peuvent également être **génériques**, ce qui permet de définir des méthodes avec des **paramètres de type**.

Méthode générique :

```
public class Util {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Utilisation :

```
Integer[] intArray = {1, 2, 3, 4, 5};
String[] strArray = {"one", "two", "three"};

Util.printArray(intArray); // Affiche 1 2 3 4 5
Util.printArray(strArray); // Affiche one two three
```

3. Bornes de Types

Vous pouvez **restreindre les types** qui peuvent être **utilisés** avec des **paramètres de type** en utilisant des **bornes**

- `extends` pour les **classes** et les **interfaces**
- `super` pour les **types inférieurs** (parent ou interfaces d'un type T)

Exemple de bornes de types :

```
public class Util {
    public static <T extends Number> void printNumbers(T[] array) {
        for (T number : array) {
            System.out.print(number + " ");
        }
        System.out.println();
    }
}
```

Utilisation :

```
Integer[] intArray = {1, 2, 3, 4, 5};
Double[] doubleArray = {1.1, 2.2, 3.3};

Util.printNumbers(intArray);
// Affiche 1 2 3 4 5

Util.printNumbers(doubleArray);
// Affiche 1.1 2.2 3.3

// Util.printNumbers(new String[]{"one", "two"});
// Erreur de compilation
```

4. Wildcards (Jokers)

Les **wildcards** permettent de travailler avec des **types inconnus** dans les génériques.
Il en existe trois types principaux :

- `?` (wildcard non borné)
  => n'importe quel type
- `? extends T` (wildcard avec borne supérieure)
  => n'importe quel type qui hérite de T
- `? super T` (wildcard avec borne inférieure)
  => n'importe quel type parent/interface de T

Wildcard non borné :

```
public static void printList(List<?> list) {
    for (Object element : list) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Utilisation :

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> strList = Arrays.asList("one", "two", "three");

printList(intList); // Affiche 1 2 3
printList(strList); // Affiche one two three
```

Exemple Complet

L'exemple suivant illustre l'**utilisation complète** des **génériques** en Java.

Nous allons :

- Définir des classes `Vis`, `VisCruciforme` et `VisPlate`
- Utiliser un `Tournevis` génériques
- Utiliser un `TournevisPlat`, héritant de `Tournevis`
- Utiliser un `TournevisUniversel`, pour démontrer l'utilisation des méthodes génériques avec et sans wildcards.

Définitions des Vis utilisées

```
public abstract class Vis {
    protected String taille;

    public Vis(String taille) {
        this.taille = taille;
    }

    public abstract void serrer();
    public abstract void desserrer();
}

public class VisCruciforme extends Vis {
    public VisCruciforme(String taille) {
        super(taille);
    }

    @Override
    public void serrer() {
        System.out.println("Serrer la vis cruciforme de taille "
            + taille);
    }

    @Override
    public void desserrer() {
        System.out.println("Desserrer la vis cruciforme de taille "
            + taille);
    }
}
```

```
public class VisPlate extends Vis {
    public VisPlate(String taille) {
        super(taille);
    }

    @Override
    public void serrer() {
        System.out.println("Serrer la vis plate de taille "
            + taille);
    }

    @Override
    public void desserrer() {
        System.out.println("Desserrer la vis plate de taille "
            + taille);
    }
}
```

Classe Générique

```
public class TournevisAEmbout<T extends Vis> {
    // s'adaptera à la vis passée à l'instanciation et la définition
    public void utiliser(T vis) {
        vis.serrer();
        vis.desserrer();
    }
}
```

Utilisation :

```
VisCruciforme visCruciforme = new VisCruciforme("M4");
TournevisAEmbout<VisCruciforme> tournevisCruciforme = new TournevisAEmbout<>();
tournevisCruciforme.utiliser(visCruciforme);
```

tournevisCruciforme ne pourra pas rechanger de type de vis plus tard

Héritage Générique

```
public class TournevisPlat extends TournevisAEmbout<VisPlate> {
    // En héritant on spécifie le type ici
    // Possibilité d'ajouter des méthodes spécifiques pour les vis plates
}
```

Utilisation :

```
VisPlate visPlate = new VisPlate("M5");
TournevisPlat tournevisPlat = new TournevisPlat();
tournevisPlat.utiliser(visPlate);
```

Méthode Générique

```
public static class TournevisUniversel {
    public static <T extends Vis> void utiliser(T vis) {
        // s'adaptera à la vis passée à l'appel
        vis.serrer();
        vis.desserrer();
    }
}
```

Utilisation :

```
TournevisUniversel.utiliser(new VisCruciforme("M4"));
TournevisUniversel.utiliser(new VisPlate("M5"));
```

Wildcard

```
public class TournevisUniversel {
    public static void tournerNImporteQuoi(Collection<?> objets) {
        for (Object objet : objets) {
            System.out.println("Tourner l'objet : " + objet.toString());
        }
    }
}
```

Utilisation :

```
List<Object> objets = Arrays.asList("Bonjour", new Crepe(), new Sablier());
TournevisUniversel.tournerNImporteQuoi(objets);
List<Animal> animaux = Arrays.asList(new Chat(), new ChauveSouris(), new Baleine());
TournevisUniversel.tournerNImporteQuoi(animaux);
// équivalent en précisant explicitement le type
TournevisUniversel.<Collection<Animal>>tournerNImporteQuoi(animaux);
```

Quelle lettre utiliser pour le Paramètre de Type ?

En Java, les **lettres couramment utilisées** pour les paramètres de type générique sont :

- **E** : Pour les **éléments** dans une collection (**Element**).
- **K** : Pour les **clés** dans une map (**Key**).
- **V** : Pour les **valeurs** dans une map (**Value**).
- **T** : Pour représenter un **type** générique (**Type**).
- **S, U, V, etc.** : Pour d'autres types génériques **supplémentaires**.

Ces lettres sont choisies pour leur **clarté** et par **convention** dans la communauté Java.

Conclusion

La généricité en Java permet d'écrire des **classes**, des **interfaces** et des **méthodes** plus **flexibles** et **réutilisables** tout en maintenant la sécurité des types.

Elle facilite le développement de **bibliothèques génériques** et **améliore la lisibilité et la maintenabilité du code**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

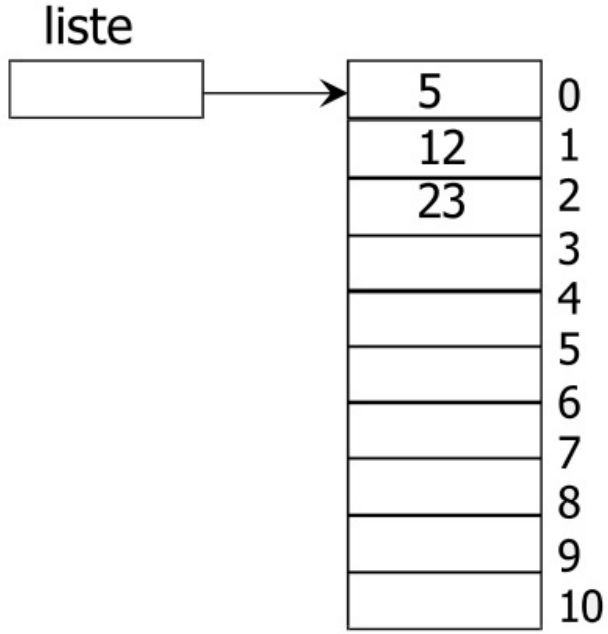
Les Collections

Tableaux de primitives

- Déclaration :
  - Tableau de nombres entiers `int[] tab;`
  - `tab` fera **référence** à un tableau d'entier (**handle**)
- Instanciation du tableau

`tab = new int[11];`

```
liste[0]=5; liste[1]=12; liste[3]=23;
for(int  i=0;i<liste.length;i++){
    System.out.println(liste[i]);
}
```



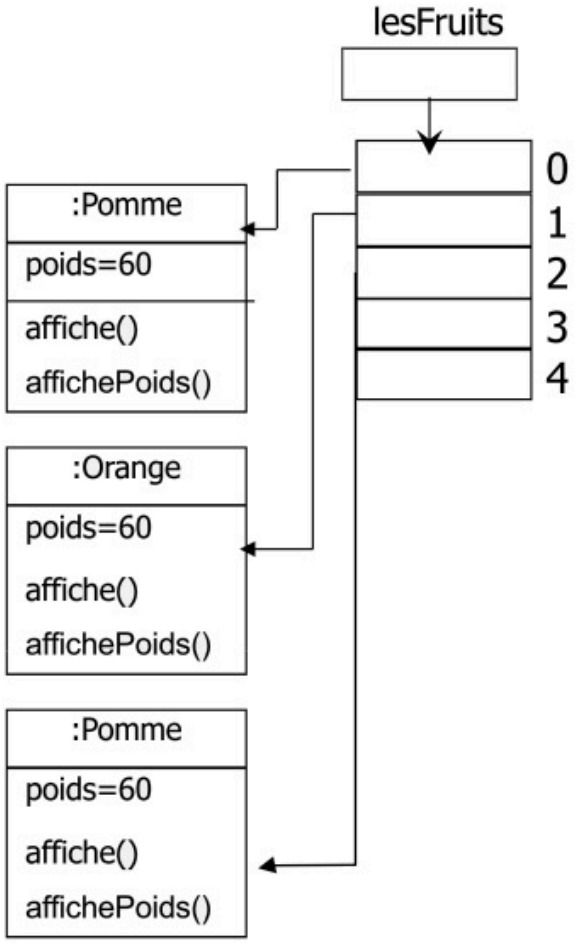
Tableaux d'objets

- Déclaration d'un Tableau d'objets Fruit :

```
Fruit[] lesFruits;
```

- Instanciation du tableau
  - `lesFruits = new Fruit[5];`
- Création des objets :

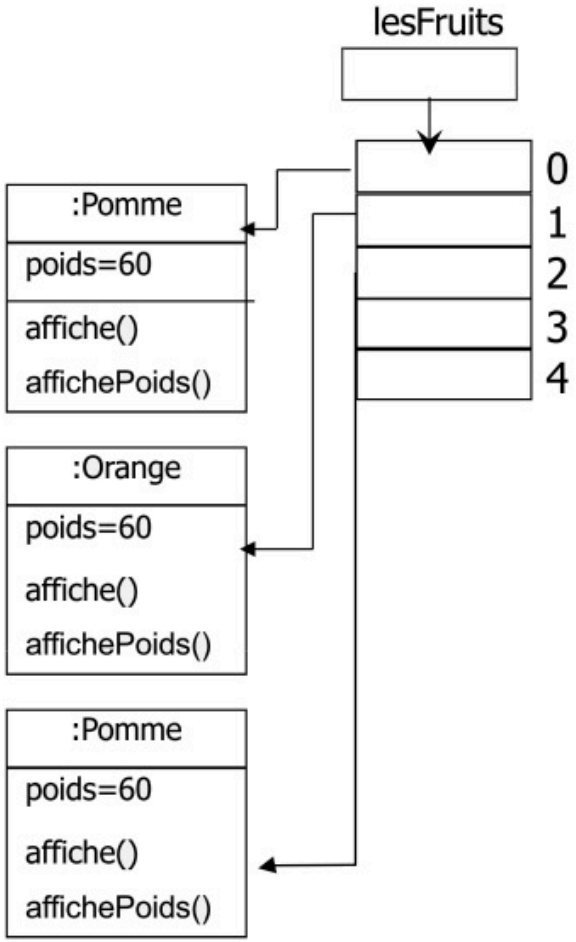
```
lesFruits[0] = new Pomme(60);
lesFruits[1] = new Orange(100);
lesFruits[2] = new Pomme(55);
```



Tableaux d'objets

- Un tableau d'objets est un tableau de **handles/références**
- Manipulation des objets :

```
for(int i=0;i<lesFruits.length;i++){
  lesFruits[i].affiche();
  if(lesFruits[i] instanceOf Pomme)
    ((Pomme)lesFruits[i]).affichePoids();
  else
    ((Orange)lesFruits[i]).affichePoids();
  }
```



API Collections

Java propose l'**API Collections** qui offre un socle riche et des implémentations d'**objets de type collection** enrichies au fur et à mesure des versions de Java.

L'API Collections possède **deux grandes familles** chacune définies par une **Interface Générique** :

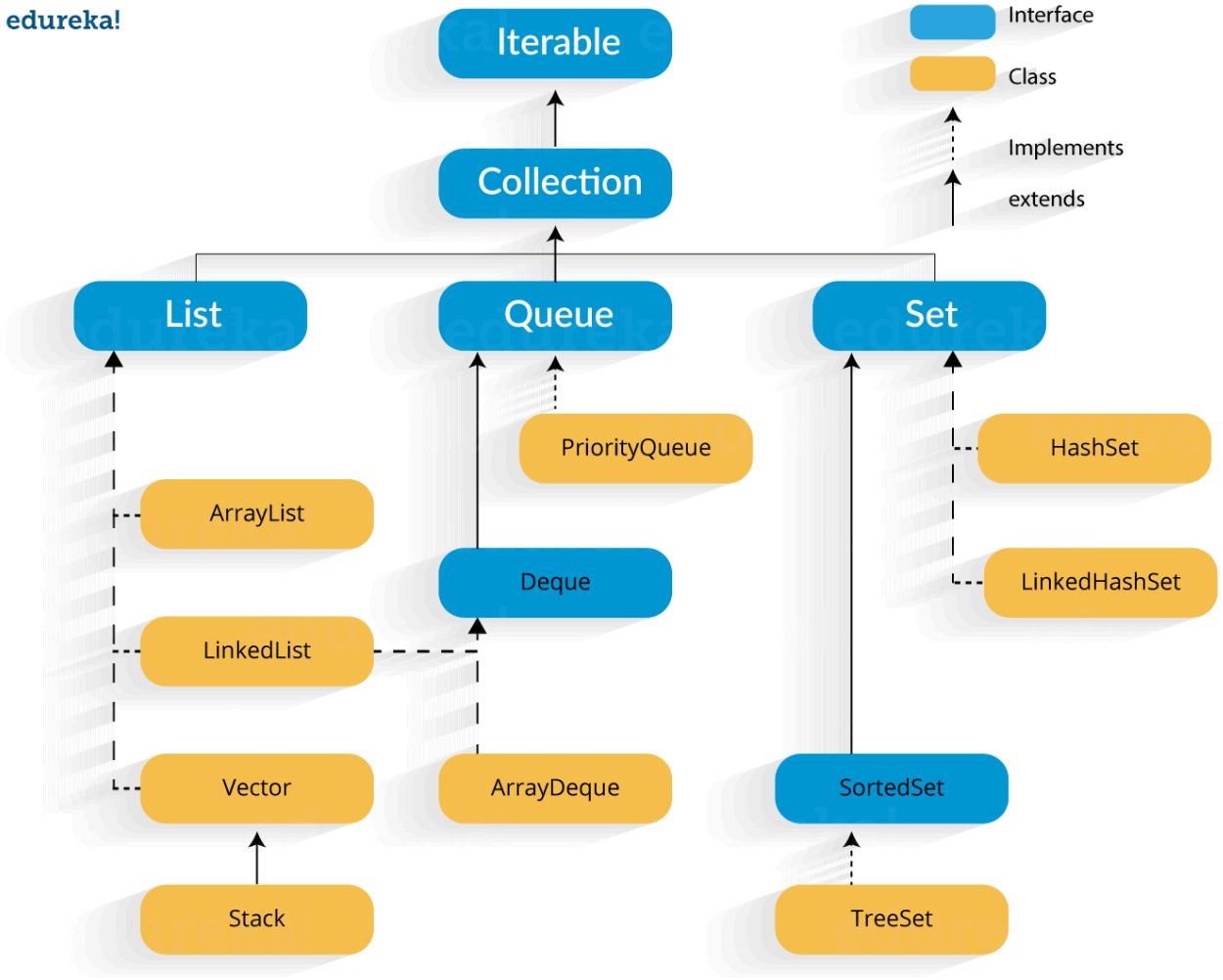
- **java.util.Collection** : pour gérer un groupe d'objets
- **java.util.Map** : pour gérer des éléments de type **paires de clé/valeur**, ils sont assimilables à des **dictionnaires** dans d'autres langages

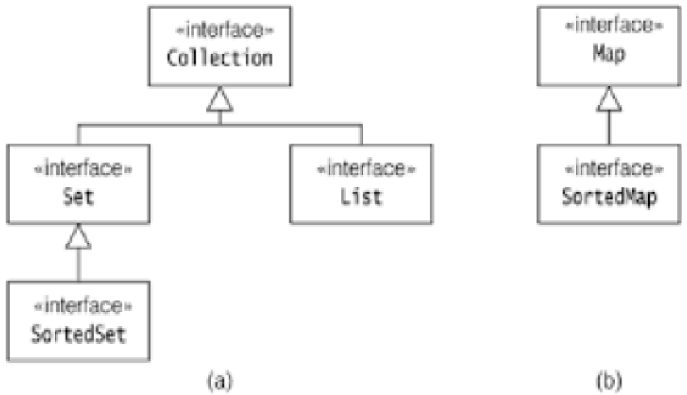
API Collections

l'API Collections définit enfin :

- Deux interfaces pour le **parcours** de certaines **collections** : `Iterator` et `ListIterator`.
- Une interface et une classe pour permettre le **tri** de certaines collections : `Comparable` et `Comparator`
- Des classes utilitaires : `Arrays`, `Collections`
  Elles sont plus ou moins équivalentes aux Wrappers des primitifs (Integer, Boolean, Double, ..)

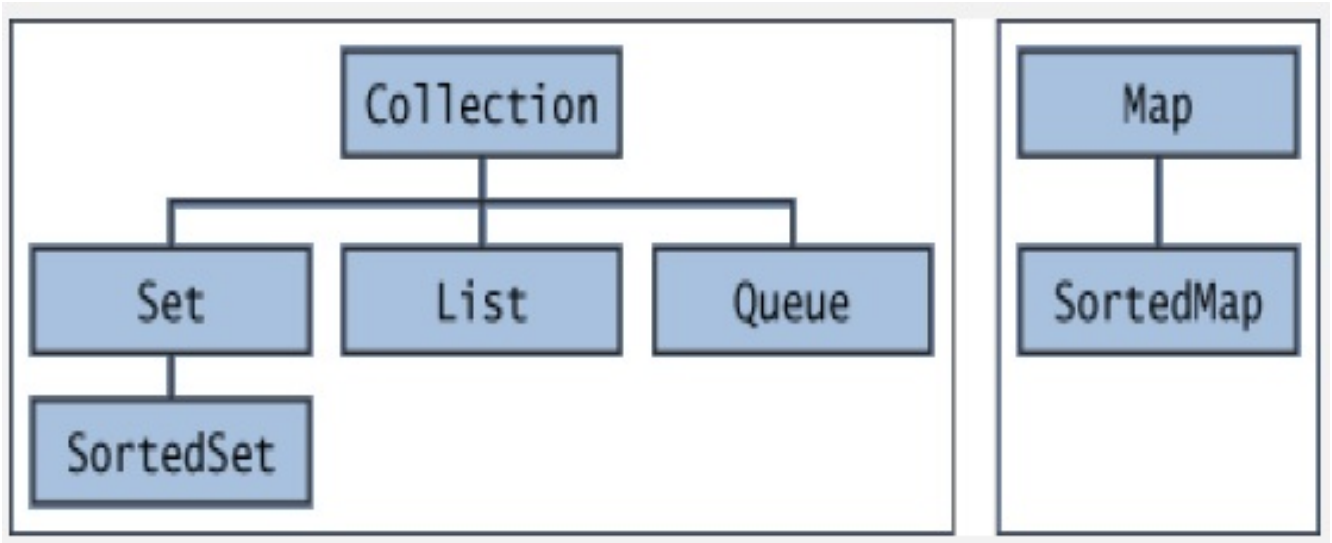
Architecture des Collections





Interfaces des Collections

- Les **interfaces génériques de collection** principales regroupent différents types de collections. Ils **représentent les types de données abstraites** qui font partie de l'infrastructure de collections. Ce sont des **interfaces** donc elles **ne fournissent pas d'implémentation** !



Collections

- Une collection est un **tableau dynamique** d'**objets** de type `Object` (pas de primitifs).
- Une collection fournit un **ensemble de méthodes** qui permettent:
  - **Ajouter** un nouveau objet dans le tableau
  - **Supprimer** un objet du tableau
  - **Rechercher** des objets selon des critères
  - **Trier** le tableau d'objets
  - **Filtrer** les objets du tableau
  - Etc...

Quand utiliser une collection et laquelle utiliser ?

- Dans un problème, les **tableaux** peuvent être utilisés quand la **dimension** du tableau est obligatoirement **fixe**.
- **Dans le cas contraire, il vaut mieux utiliser les collections :**
  - **List**, le plus souvent, pour **regrouper des éléments simplement**
  - **Set** pour **éviter les doublons**
  - **Map** pour des **correspondences** de couples/paires **clé-valeur**
  - **Queue** et **Stack** dans des cas particuliers (cf. **FIFO/LIFO**)
  - Si l'on fait du **Multithreading**, on préférera les **collections Thread-safe**

Collections

Java fournit plusieurs types de collections :

- `ArrayList`
- `HashSet`
- `HashMap`
- `Vector`
- `TreeSet`
- `TreeMap`
- Etc...
- Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections de type **List**, **Set** et **Map**

Comparatif des Collections

Collection	Ordonné	Accès direct	Clé / valeur	Doublons	Null	Thread Safe
ArrayList	Oui	Oui	Non	Oui	Oui	Non
LinkedList	Oui	Non	Non	Oui	Oui	Non
HashSet	Non	Non	Non	Non	Oui	Non
TreeSet	Oui	Non	Non	Non	Non	Non
HashMap	Non	Oui	Oui	Non	Oui	Non
TreeMap	Oui	Oui	Oui	Non	Non	Non
Vector	Oui	Oui	Non	Oui	Oui	Oui
Hashtable	Non	Oui	Oui	Non	Non	Oui
Properties	Non	Oui	Oui	Non	Non	Oui
Stack	Oui	Non	Non	Oui	Oui	Oui
CopyOnWriteArrayList	Oui	Oui	Non	Oui	Oui	Oui
ConcurrentHashMap	Non	Oui	Oui	Non	Non	Oui
CopyOnWriteArraySet	Non	Non	Non	Non	Oui	Oui

A quoi ressemble l'interface Collection

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operation
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);  //optional
    boolean removeAll(Collection<?> c);         //optional
    boolean retainAll(Collection<?> c);         //optional
    void clear();                               //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Iterable

L'interface Générique **Iterable** est plus globale, elles rassemble l'ensemble des choses lur lesquelles on peut itérer, notamment les collection mais aussi les **Iterators**, similaire aux **Générateurs** d'autres langages

Il existe trois manières d'itérer les objets de Iterable.

1. Utilisation du enhanced-for (for-each)
2. Utilisation de la Méthode Iterable.forEach
3. Utilisation de l' interface **Iterator<T>** avec une classe

Iterator

- **Iterator** est une interface qui itère les éléments. Il permet de parcourir la liste et de modifier les éléments.
- **L'interface Iterator a trois méthodes qui sont mentionnées ci-dessous :**
  1. **public boolean hasNext()** — Cette méthode renvoie true si l'itérateur a plus d'éléments.
  2. **public object next()** — Il renvoie l'élément et déplace le pointeur du curseur vers l'élément suivant.
  3. **public void remove()** — Cette méthode supprime les derniers éléments renvoyés par l'itérateur.

Iterator

- La collection de type Iterator du package java.util est souvent utilisée pour afficher les objets d'une autre collection
- En effet il est possible d'obtenir un iterator à partir de chaque collection.

Exemple

- Création d'un vecteur de Fruit.
- `Vector<Fruit> fruits=new Vector<Fruit>();`
- Ajouter des fruits aux vecteur
- `fruits.add(new Pomme(30));`
- `fruits.add(new Orange(25));`
- `fruits.add(new Pomme(60));`
- Création d'un Iterator à partir de ce vecteur
- `Iterator<Fruit> it=fruits.iterator();`
- Parcourir l'Iterator:

List

ArrayList

- `ArrayList<>` est une classe de `java.util` qui implémente l'interface `List<>`.
- Elle permet de **stocker des éléments** de manière **ordonnée** avec des **index**, comme un tableau mais **sa taille est variables** en fonction des éléments qu'elle continent.
- Les éléments peuvent être **ajoutés**, **accédés**, **modifiés**, et **supprimés** par leur index.

Utilisation de ArrayList

- Déclaration d'une `ArrayList` pour stocker des objets `Fruit` :

```
List<Fruit> fruits = new ArrayList<Fruit>();
```

- Ajout de deux objets `Fruit` à la liste :

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
```

- Affichage des objets de la liste :

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

Utilisation de ArrayList

- Utilisation de la boucle `for-each` pour afficher les objets :

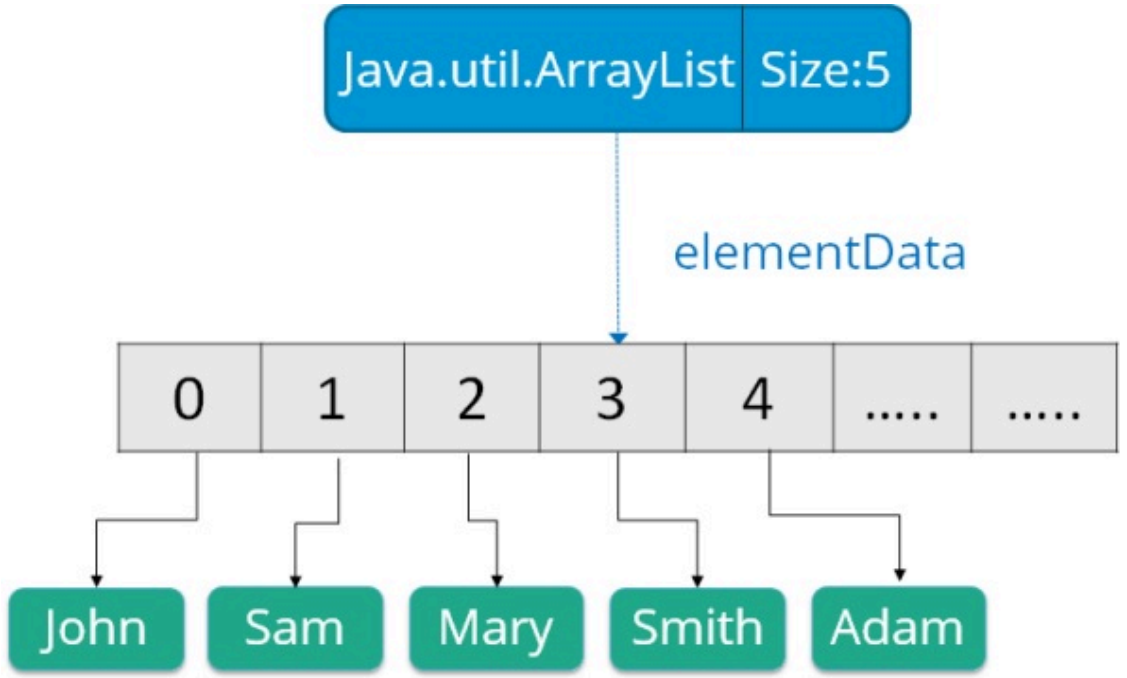
```
for (Fruit f : fruits) {
    System.out.println(f);
}
```

- Suppression du deuxième objet de la liste :

```
fruits.remove(1);
```

Exemple d'utilisation de ArrayList

```
// Déclaration d'une liste de type Fruit
List<Fruit> fruits;
// Création de la liste
fruits = new ArrayList<Fruit>();
// Ajout de 3 objets Pomme,Orange et Pomme à la liste
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
fruits.add(new Pomme(60));
// Parcourir tous les objets
for (int i= 0;i<fruits.size();i++){
    // Faire appel à la méthode affiche()
    // de chaque Fruit de la liste
    fruits.get(i).affiche();
}
// Une autre manière plus simple pour parcourir une liste
for(Fruit f:fruits){ // Pour chaque Fruit de la liste
// Faire appel à la méthode affiche() du Fruit f
    f.affiche();
}
```



Vectors

- `Vector` est une autre classe de `java.util` qui fonctionne de manière similaire à `ArrayList`.
- Les principales différences résident dans la synchronisation et l'expansion dynamique des `Vector`.

=> Utilisés dans le Multithreading

Comparaison ArrayList vs Vector

N°	ArrayList	Vector
1.	ArrayList n'est pas synchronisé.	Le vecteur est synchronisé.
2.	ArrayList incrémente 50 % de la taille actuelle du tableau si le nombre d'éléments dépasse sa capacité.	Les incréments vectoriels de 100 % signifient que la taille du tableau double si le nombre total d'éléments dépasse sa capacité.
3.	ArrayList n'est pas une classe héritée. Il est introduit dans JDK 1.2.	Vector est une classe héritée.
4.	ArrayList est rapide car non synchronisé.	Vector est lent parce qu'il est synchronisé, c'est-à-dire que dans un environnement multithreading, il maintient les autres threads dans un état exécutable ou non jusqu'à ce que le thread actuel libère le verrou de l'objet.
5.	ArrayList utilise l'interface Iterator pour parcourir les éléments.	Un vecteur peut utiliser l'interface Iterator ou l'interface Enumeration pour parcourir les éléments.

Pourquoi utiliser l'interface List plutôt que ArrayList ?

- **Flexibilité :** En utilisant l'interface `List`, il est facile de changer l'implémentation sous-jacente sans modifier le reste du code. Par exemple, on peut remplacer `ArrayList` par `LinkedList` si nécessaire.

```
List<Fruit> fruits = new LinkedList<Fruit>();
```

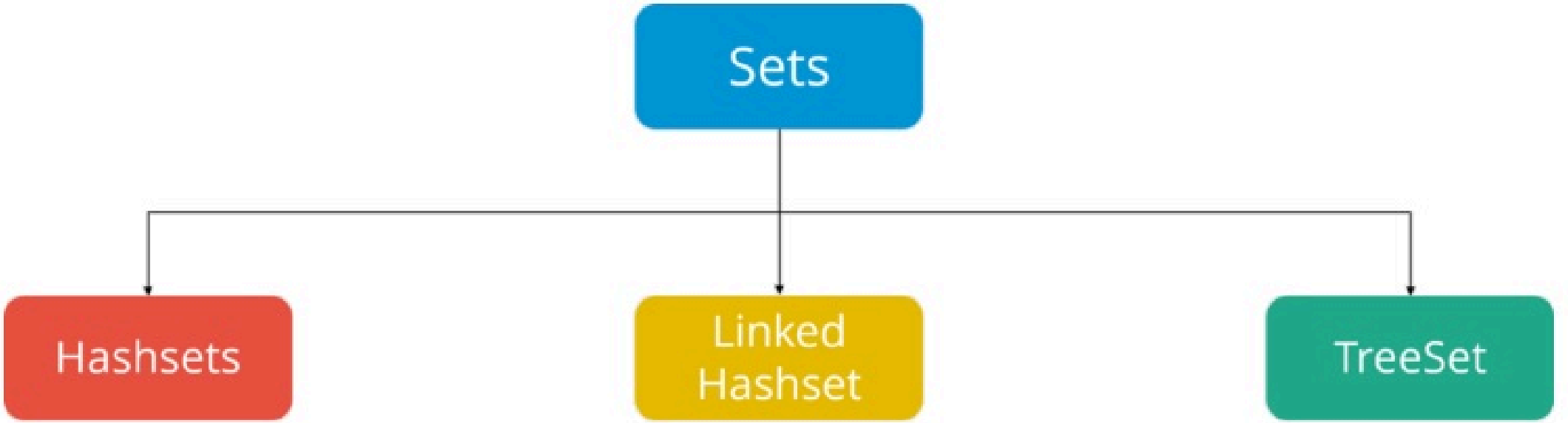
- **Polymorphisme :** Permet d'utiliser différentes implémentations de `List<>` de manière interchangeable.
- **Bonnes pratiques :** favorise la maintenabilité et l'extensibilité du code.

Set

Set<E>

- **Set** est une collection qui ne peut pas contenir d'éléments en double.
- Modélise les **ensembles mathématiques**.
- Exemple :
  - Trouver chaque mot/lettre utilisé dans un Livre
  - Une main de poker (pas possible d'avoir plusieurs cartes identiques).

Types qui implémentent Set<E>



HashSet

- **HashSet** utilise une **table de hachage** pour le stockage.
- Contient uniquement des éléments uniques.
- Hérite de `AbstractSet` et implémente `Set`.
- Les éléments sont triés selon leur **hash**

Un **hash** est une **valeur fixe générée par une fonction de hachage** à partir d'**une entrée de taille variable**, servant à **identifier rapidement et de manière unique** les données d'origine.

Méthodes de HashSet

Method	Description
boolean add(Object o)	Adds the specified element to this set if it is not already present.
boolean contains(Object o)	Returns true if the set contains the specified element.
void clear()	Removes all the elements from the set.
boolean isEmpty()	Returns true if the set contains no elements.
boolean remove(Object o)	Remove the specified element from the set.
Object clone()	Returns a shallow copy of the HashSet instance: the elements themselves are not cloned.
Iterator iterator()	Returns an iterator over the elements in this set.
int size()	Return the number of elements in the set.

TreeSet

- **TreeSet** maintient ses éléments dans l'**ordre croissant** selon **leur contenu**, s'applique surtout au primitifs
- Les objets seront trié différemment si ils implémentent `Comparable`.
- Fournit des opérations supplémentaires pour exploiter cet ordre.
- Utilisé pour les ensembles ordonnés naturellement comme les **listes de mots ou de chiffres**.

Exemple

```
HashSet<String> set = new HashSet<>();
set.add("Java");
set.add("Python");
set.add("Python3");
set.add("C++");
set.add("C++");
set.add("C++");
System.out.println("HashSet : "+ set);


// Démo pour SortedSet (TreeSet)
SortedSet<String> sortedSet = new TreeSet<>();
sortedSet.add("Java");
sortedSet.add("Python");
sortedSet.add("Python3");
sortedSet.add("C++");
sortedSet.add("C++");
sortedSet.add("C++");
System.out.println("SortedSet : "+ sortedSet);

// Méthodes pour SortedSet
System.out.println("1. Premier element : "+ sortedSet.first());
System.out.println("2. Dernier element : "+ sortedSet.last());
SortedSet<String> headset = sortedSet.headSet("Python");
System.out.println("3. Sous-ensemble avant 'Python' : "+headset);
```

Map

Interface Map<K,V>

- **Map** est une interface pour les collections qui **mappent** des **clés** aux **valeurs** correspondantes.
- Appelé Dictionnaire dans d'autres langages
- **Ne peut pas contenir de clés en double**.
- Chaque **clé** correspond à au plus **une valeur**.
- **Pas d'index**

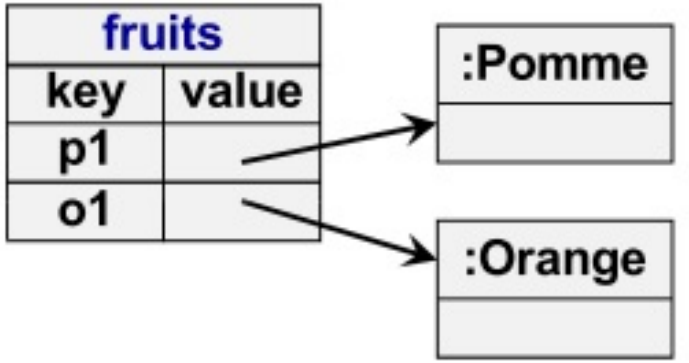
Exemple de Map/Dictionnaire

Un panier dans un site e-commerce :

Produit: `String` ou `Produit`	Quantité : `Integer`
"Pomme"	3
"Banane"	4
"Bière"	4
"Pâtes Tortellini 5kg"	40
"Pâtes Spaghetti 1kg"	5
"Boisson énergisante 50cl"	12
"Concombre"	1
"Cookie nougatine"	128

HashMap<K,V>

- **HashMap<K,V>** est une classe qui **implémente l'interface Map**.
- Ajout d'**objets de même type** et récupération par **clé**.
- **Itération** à travers **chaque couple clé-valeur** (`Map.Entry<K,V>`) de la **collection** via **for-each**, possible **uniquement** après **cast** en `HashSet<Map.Entry<K,V>>`.
- Triés par **hash** comme pour **HashSet**



TreeMap<K,V>

- **TreeMap<K,V>** maintient ses mappages dans l'**ordre croissant des clés**.
- Même fonctionnement que `TreeSet<E>` mais associe des valeurs aux clés.
- Utilisé pour les collections **ordonnées** de paires clé/valeur.

Exemple

```
// Démo pour le HashMap
HashMap<String,Integer> hashMap = new HashMap<>();
hashMap.put("Java",20);
if(!hashMap.containsKey("Java"))
    // si ne contient pas "Java" je l'ajoute
    hashMap.put("Java",22);
hashMap.put("Python",10);
hashMap.put("C++",30);
System.out.println("\nHashMap : "+ hashMap);


// Méthodes pour hashmap
System.out.println("1. Nombre d'Entry du Hashmap : "
    + hashMap.size());

System.out.println("2. Valeur associé à 'Java' : "
    + hashMap.get("Java"));

System.out.println("3. Est ce que 'Test' est présent ? : "
    + hashMap.containsKey("Test"));

System.out.println("4. Suppression de l'entrée avec la clé Python : ");
hashMap.remove("Python");

System.out.println(" Nouveau HashMap : "+ hashMap);
```

```
Set<Map.Entry<String, Integer>> hashSet = hashMap.entrySet();
System.out.println(hashSet);

for(Map.Entry<String, Integer> unEntry : hashSet){
    System.out.println(unEntry);
    System.out.println("Clé :" + unEntry.getKey());
    System.out.println("Valeur :" + unEntry.getValue());
}

for(Map.Entry<String, Integer> unEntry : hashMap.entrySet()){
  // les hashMap n'ont pas d'index donc ils ne sont pas itérable,
  // il faudra le convertir en HashSet
    System.out.println(unEntry);
    System.out.println("Clé :" + unEntry.getKey());
    System.out.println("Valeur :" + unEntry.getValue());
}
```

Exceptions

Cas exceptionnels

- Les **programmes** doivent souvent gérer des **situations exceptionnelles**, rendant le code **complexe** et difficile à lire.
- Exemples : saisie utilisateur en int, division par zéro
- En algorithmie, on appelle ces **situations exceptionnelles** des **Exceptions**
- Java introduit un **mécanisme de gestion des exceptions** pour **séparer** le **code utile** du **traitement de ces cas exceptionnels**.

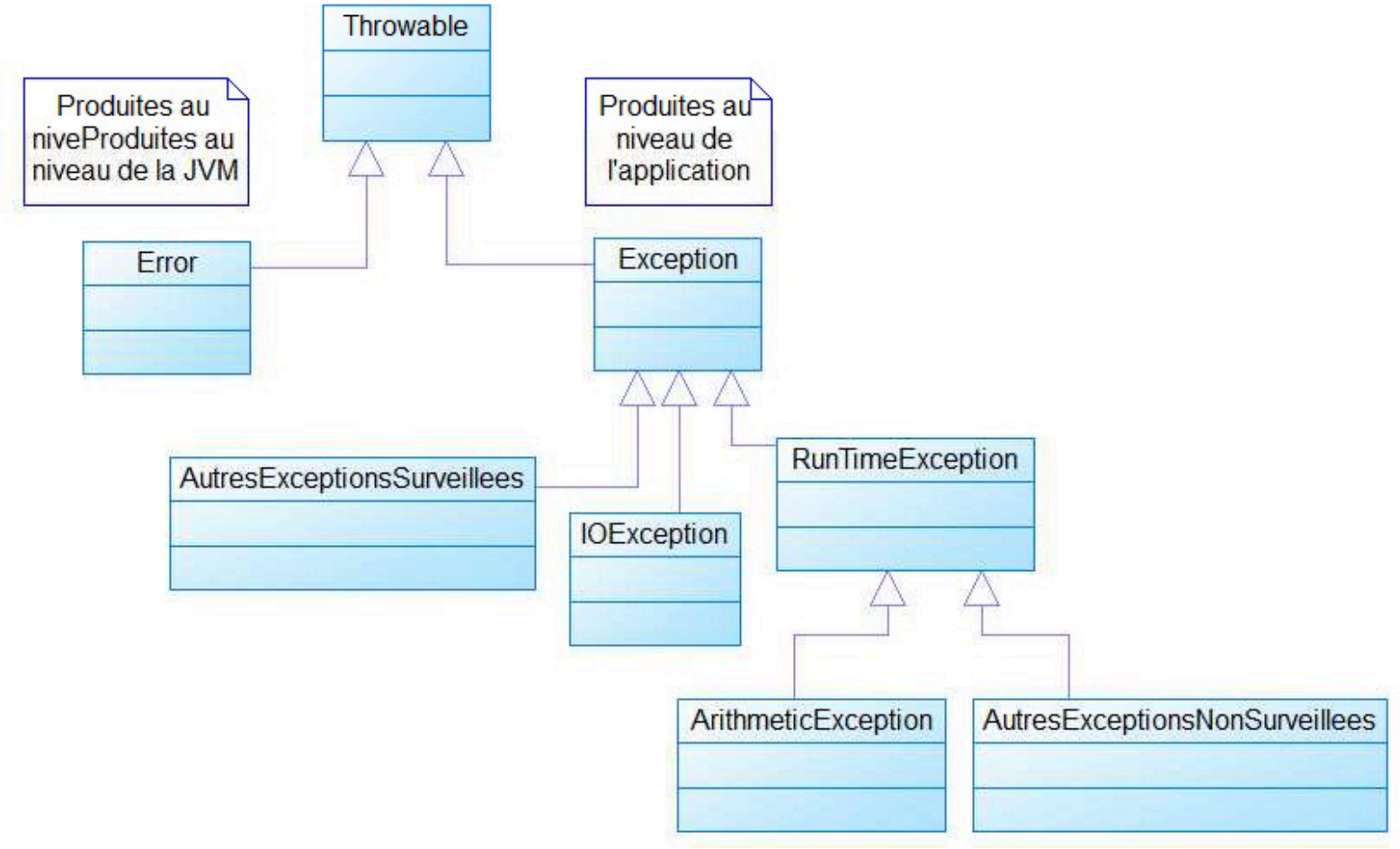
Concept de Throwable et Throw

- `Throwable` est la classe de base pour tous les objets pouvant être **lancés**.
- **"Lancé"** signifie qu'un `Throwable` est **généré/instancié et propagé dans le programme**.
- Lorsqu'une **condition anormale** survient, on utilise le mot-clé `throw` pour **créer et "lancer"** une instance **interrompant le flux normal d'exécution** du programme.
- Permet de **déclencher explicitement** une exception en réponse à une condition spécifique dans le code.

Exemple

```
int a = 1;
int b = 0;
if (b == 0) {
    throw new ArithmeticException("Division par zéro");
}
System.out.println(a/b);
```

Hiérarchie des throwables



Hiérarchie des throwables

- **Throwable** : **base/classe mère**
- `Exception` : erreurs prévues que le **programmeur doit traiter**.
- `Error` : erreurs imprévisibles, **générées par la JVM**.

Types d'Exceptions

- **Surveillées** (`Exception`) : doivent être traitées, signalées par le compilateur.
- **Non surveillées** (`RuntimeException`) : traitement optionnel, non signalées par le compilateur.
- Java **oblige** le programmeur à **traiter les erreurs surveillées**. Elles sont **signalées par le compilateur et l'IDE**.
- Les erreurs **non surveillées peuvent être traitées ou non**. Et ne sont **pas signalées par le compilateur ou l'IDE**.

Exemple

- Saisie de deux entiers, division et affichage du résultat :

```java
public static int calcul(int a, int b) {
    return a / b;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Donné a: ");
    int a = scanner.nextInt();
    System.out.print("Donné b: ");
    int b = scanner.nextInt();
    int resultat = calcul(a, b);
    System.out.println("Résultat = "
        + resultat);
}
```

Exécution

- Cas normal :

```
Donné a: 12
Donné b: 6
Resultat = 2
```

- Cas où b = 0 :

```
Donné a: 12
Donné b: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at org.example.Main.calcul(Main.java:7)
        at org.example.Main.main(Main.java:16)
```

Un bug dans l'application

- Le cas du **scénario 2** indique que qu'une **erreur fatale s'est produite** dans l'application **au moment de l'exécution**.
- Cette exception est de type `ArithmeticException`.
- Elle concerne une **division par zero**

  `/ by zero`

Un bug dans l'application

- L'**origine** de cette exception étant la **méthode calcul** dans la ligne numéro 7.

```
at org.example.Main.calcul(Main.java:7)
```

- Cette exception **n'a pas été traité** dans calcul.
- Elle **remonte ensuite vers main** à la ligne numéro 16 dont elle n'a pas été traitée.

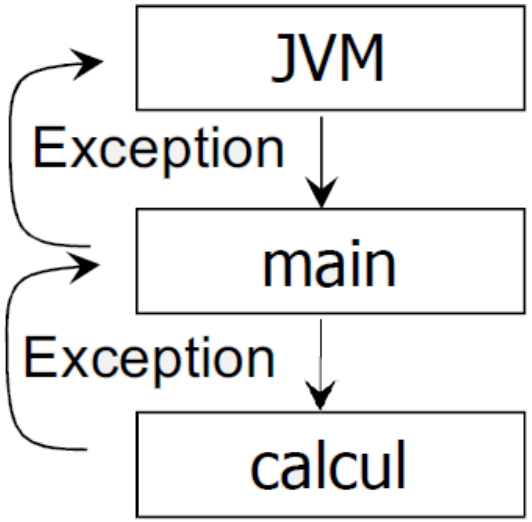
```
at org.example.Main.main(Main.java:16)
```

Un bug dans l'application

- Après l'exception est **signalée à la JVM**.
- Quand une **exception arrive à la JVM**, cette dernière **arrête l'exécution** de l'application, ce qui constitue **un bug fatal**.
- Le fait que le message « Resultat= » **n'a pas été affiché**, montre que l'application **ne continue pas son exécution normale** après la division par zero.

Un bug dans l'application

- Tout ce chemin est affiché dans la console, c'est ce qu'on appelle la **StackTrace**
- Il est précédé par une ligne indiquant le **thread d'exécution** (main en mono-thread), le **Throwable** (Error/Exception), ainsi que le **message lié** (explications)




```
Exception in thread "main" java.lang.
ArithmeticException: / by zero
        at org.example.Main.calcul(Main.java:7)
        at org.example.Main.main(Main.java:16)
```

Traiter l'exception

Dans java, pour traiter les exceptions, on doit utiliser le bloc **try catch** de la manière suivante:

```
public static int calcul(int a,int b){
    int c=a/b;
    return c;
}
public static void main(String[] args) {
    Scanner clavier=new Scanner(System.in);
    System.out.print("Donner a:");
    int a=clavier.nextInt();
    System.out.print("Donner b:");
    int b=clavier.nextInt();
    int resultat=0;
    try{
        // on essaye le bloc suivant
        resultat=calcul(a, b);
    }
    catch (ArithmeticException e) {
        //si l'exception ArithmeticException est levée
        System.out.println("Division par zero");
    }
    System.out.println("Resultat="+resultat);
}
```

Scénario 1

```
Donner a:12
Donner b:6
Resultat=2
```

Scénario 1

```
Donner a:12
Donner b:0
Division par zero
Resultat=0
```

Principale Méthodes d'une Exception

Tous les types d'exceptions possèdent les méthodes suivantes :

- getMessage() : retourne le message de l'exception

```
System.out.println(e.getMessage()); // / by zero
```

- toString() : retourne une chaine qui contient le type de l'exception et le message de l'exception.

```
System.out.println(e.toString()); // java.lang.ArithmeticException: / by zero
```

Principale Méthodes d'une Exception

- printStackTrace: affiche la trace de l'exception

```
e.printStackTrace();
/*
Résultat affiché :
java.lang.ArithmeticException: / by zero
at App1.calcul(App1.java:4)
at App1.main(App1.java:13)
*/
```

Générer, Relancer ou Jeter une Exception

- Exemple avec une classe **Compte** :

```java
package metier;

public class Compte {
  private int code;
  private float solde;

  public void verser(float mt) {
    solde += mt;
  }

  public void retirer(float mt) throws Exception {
    // nous reviendrons sur cette syntaxe ensuite
    if (mt > solde)
        throw new Exception("Solde Insuffisant");
    solde -= mt;
  }

  public float getSolde() {
    return solde;
  }
}
```

```java
package pres;
import java.util.Scanner;
import metier.Compte;
public class Application {
  public static void main(String[] args) {
    Compte cp=new Compte();
    Scanner clavier=new Scanner(System.in);
    System.out.print("Montant à verser:");
    float mt1=clavier.nextFloat();
    cp.verser(mt1);
    System.out.println("Solde Actuel:"+cp.getSolde());
    System.out.print("Montant à retirer:");
    float mt2=clavier.nextFloat();
    cp.retirer(mt2);
    // Le compilateur ou l'IDE signalent l'Exception
    // Il nous oblige à nous en occuper
  }
}
```

Deux solutions pour Traiter l'Exception

- Utilisation de **try-catch**

```
try {
  cp.retirer(mt2);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

- Ou **propagation** de l'exception, elle sera **remontée au niveau supérieur**, ici, la JVM

```
public static void main(String[] args) throws Exception {
  // code ...
}
```

Classe Exception == Exception surveillée

- Ce **comportement** de l'IDE et du Compilateur et la notion de **propagation** concernent surtout les **Exception surveillées**, classes filles de `Exception`.
- Lors de l'utilisation de `RuntimeException` et **ses classes filles**, on parle d'**Exception non-surveillées**, les même syntaxes sont applicables mais l'IDE et le compilateur ne les force pas.

Générer une Exception non surveillée

- Modifiez notre Exemple précédent avec une exception de type **RuntimeException** et observez les différences (enlever et remettre propagation et try...catch)

```
public void retirer(float mt) {
  if (mt > solde) throw new RuntimeException("Solde Insuffisant");
    solde -= mt;
}
```

Personnaliser les Exceptions

- L'exception générée dans la méthode retirer est une exception **métier** (relative à nos besoins spécifiques).
- Il est plus professionnel de **créer une nouvelle Exception** nommée `SoldeInsuffisantException` de la manière suivante :

```
package metier;
public class SoldeInsuffisantException extends Exception {
    // notez la norme de nommage ...Exception
    public SoldeInsuffisantException(String message) {
        super(message);
    }
}
```

Utiliser l'Exception personnalisée

```
public void retirer(float mt) throws SoldeInsuffisantException {
    if (mt > solde) throw new SoldeInsuffisantException("Solde Insuffisant");
        solde -= mt;
}
```

Exemple suivant

Testez avec ces scénarios :

Scénario 1

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0
```

Scénario 2

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```

Scénario 3

```
Montant à verser:azerty
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:840)
at java.util.Scanner.next(Scanner.java:1461)
at java.util.Scanner.nextFloat(Scanner.java:2319)
at pres.Application.main(Application.java:9)
```

Améliorer l'application

- Dans le **scénario 3**, nous découvrons qu'**une autre exception non surveillée est générée** dans le cas ou nous **saisissons une chaine de caractères** et non pas un entier.
- Cette exception est de type `InputMismatchException`, générée par la méthode `nextFloat()` de la classe `Scanner`.
- Nous devrions donc faire **plusieurs catch** dans la méthode main.
- Similairement à un `else if`, on passera par chaque `catch` **jusqu'à trouver une Exception qui correspond**

/!\ Commencer par `Exception` par exemple attraperait toute les Exceptions, les catch suivants seraient inatteignable (héritage)

Multiples catch

```
public static void main(String[] args) {
    Compte cp=new Compte();
        try {
            Scanner clavier = new Scanner(System.in);
            System.out.print("Montant à verser:");
            float mt1 = clavier.nextFloat();
            cp.verser(mt1);
            System.out.println("Solde Actuel:" + cp.getSolde());
            System.out.print("Montant à retirer:");
            float mt2 = clavier.nextFloat();
            cp.retirer(mt2);
        }
        catch (SoldeInsuffisantException e) {
            System.out.println(e.getMessage());
        }
        catch (InputMismatchException e){
            System.out.println("Problème de saisie");
        }
    System.out.println("Solde Final "+cp.getSolde());
}
```

Le cas MontantNegatifException

- L'exception métier MontantNegatifException

```
package metier;
public class MontantNegatifException extends Exception {
  public MontantNegatifException(String message) {
    super(message);
  }
}
```

- La méthode retirer de la classe Compte

```
public void retirer(float mt)throws SoldeInsuffisantException,MontantNegatifException {
  if(mt<0) throw new MontantNegatifException("Montant "+mt+" négatif");
  if(mt>solde) throw new SoldeInsuffisantException("Solde Insuffisant");
  solde=solde-mt;
}
```

Application : Contenu de la méthode main

```
Compte cp=new Compte();
try {
  Scanner clavier=new Scanner(System.in);
  System.out.print("Montant à verser:");
  float mt1=clavier.nextFloat();
  cp.verser(mt1);
  System.out.println("Solde Actuel:"+cp.getSolde());
  System.out.print("Montant à retirer:");
  float mt2=clavier.nextFloat();
}
catch (SoldeInsuffisantException e) {
  System.out.println(e.getMessage());
}
catch (InputMismatchException e) {
  System.out.println("Problème de saisie");
}
catch (MontantNegatifException e) {
  System.out.println(e.getMessage());
}
System.out.println("Solde Final=" + cp.getSolde());
```

Scénario 1

```
Montant à verser:5000
SoldeActuel:5000.0
Montant à retirer:2000
Solde Final=3000.0
```

Scénario 2

```
Montant à verser:5000
SoldeActuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```

Scénario 3

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:-2000
Montant -2000.0 négatif
Solde Final=5000.0
```

Scénario 4

```
Montant à verser:azerty
Problème de saisie
Solde Final=0.0
```

Syntaxe OR

```
Compte cp = new Compte();
try {
  Scanner clavier = new Scanner(System.in);
  System.out.print("Montant à verser:");
  float mt1 = clavier.nextFloat();
  cp.verser(mt1);
  System.out.println("Solde Actuel:" + cp.getSolde());
  System.out.print("Montant à retirer:");
  float mt2 = clavier.nextFloat();
  cp.retirer(mt2);
} catch (SoldeInsuffisantException | InputMismatchException | MontantNegatifException e) {
  // multi-catch avec OR
  System.out.println(e.getMessage());
}
System.out.println("Solde Final=" + cp.getSolde());
```

Le bloc finally

- La syntaxe complète du bloc try est la suivante :

```
try {
  System.out.println("Traitement Normal");
}
catch (SoldeInsuffisantException e) {
  System.out.println("Premier cas Exceptionnel");
}
catch (NegativeArraySizeException e) {
  System.out.println("Deuxième cas Exceptionnel");
}
finally{
  System.out.println("Traitement par défaut!");
}
System.out.println("Suite du programme!");
```

- Le bloc **finally** s'exécute dans tous les scénarios.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Lambdas

Introduction

Les **lambdas** en Java, introduites avec Java 8, sont des **expressions** permettant d'écrire des **fonctions anonymes** (pas d'identificateur / symbol) de manière **concise et expressive**.

- Une lambda permet de capturer **un comportement ou une action en une seule ligne**, ce qui améliore la lisibilité et la maintenabilité du code.
- Elles simplifient notamment le code pour les opérations courantes telles que le filtrage, la transformation et l'itération sur des collections via l'**API Stream** que nous verrons ensuite.

Écriture d'une lambda

Syntaxe **Courte** (**Une** instruction, avec un `return` implicite)

```
(parameters) -> expression
```

Syntaxe **Longue** (**Un Bloc** d'instruction)

```
(parameters) -> {
    instruction;
    instruction;
    instruction;
    return expression;
}
```

Assignation à une **Variable** (**Type différent** selon les cas)

```
TypeLambda<TParam,TRetour> fctLambda = (parameters) -> expression;
```

Récupération de la référence d'une méthode existante

La **récupération** de la **référence d'une méthode** existante en Java se fait avec l'opérateur `::`, introduit avec Java 8.
Cet opérateur permet de **créer une référence à une méthode ou à un constructeur existant**, en évitant d'écrire une lambda.

Les références de méthodes rendent le code plus lisible et concis en réutilisant des méthodes existantes directement dans les expressions lambda.

4 types principaux de références de méthodes

```
//Méthode statique
Consumer<List<Integer>> sort = Collections::sort;

//Méthode d'instance d'un objet particulier
Consumer<String> print = System.out::println;

//Méthode d'instance d'un objet arbitraire d'un type particulier
Function<String, String> toUpperCase = String::toUpperCase;

//Constructeur :
Supplier<ArrayList<String>> supplier = ArrayList::new;
```

Types courants des Lambdas

En Java, les expressions lambda sont généralement utilisées avec des **interfaces** fonctionnelles du package **java.util.function**.

Nous allons voir les types d'interfaces fonctionnelles les plus couramment utilisés.

1. `Predicate<T>`

- **Description** : Représente une fonction qui prend un argument de type `T` et retourne un `boolean`.
- **Utilisation** : Filtrage
- **Exemple** :

```
Predicate<String> startsWithT = s -> s.startsWith("t");
list.stream().filter(startsWithT).forEach(System.out::println);
```

2. `Function<T, R>`

- **Description** : Représente une fonction qui prend un argument de type `T` et retourne un résultat de type `R`.
- **Utilisation** : Transformation / Mapping
- **Exemple** :

```
Function<String, String> toUpperCase = String::toUpperCase;
list.stream().map(toUpperCase).forEach(System.out::println);
```

3. `Consumer<T>`

- **Description** : Représente une opération qui prend un argument de type `T` et ne retourne rien.
- **Utilisation** : Exécution d'actions sur chaque élément
- **Exemple** :

```
Consumer<String> print = System.out::println;
list.forEach(print);
```

4. `Supplier<T>`

- **Description** : Représente une fonction qui ne prend aucun argument et retourne une valeur de type `T`.
- **Utilisation** : Fourniture de valeurs
- **Exemple** :

```
Supplier<StringBuilder> stringBuilderSupplier = StringBuilder::new;
StringBuilder sb = stringBuilderSupplier.get();
```

5. `BiFunction<T, U, R>`

- **Description** : Représente une fonction qui prend deux arguments de types `T` et `U` et retourne un résultat de type `R`.
- **Utilisation** : Opérations nécessitant deux arguments
- **Exemple** :

```
BiFunction<String, String, String> concatenate = (s1, s2) -> s1 + s2;
String result = concatenate.apply("Hello, ", "BiFunction!");
```

Utilisation des Types

Les IDE modernes facilitent la gestion des lambdas :

- **Auto-complétion** : Suggestions de code basées sur le contexte.
- **Inférence de type** : Détermination automatique des types des paramètres.
- **Documentation intégrée** : Aide contextuelle pour les interfaces fonctionnelles et les types génériques.
- **Refactoring** : Maintien des types appropriés lors de la refactorisation.

Les IDE permettent de retrouver facilement les types et d'écrire des lambdas et méthodes génériques de manière efficace.

Stream

Introduction 1/3

- L'introduction de l'**API Stream** dans Java 8 a révolutionné la manière de **traiter les Collections et les tableaux** (anciennement pattern Iterator). Son utilisation **fait penser à du requêtage SQL**.
- Cette nouvelle API propose une **approche simplifiée et efficace** pour effectuer des **traitements sur ces structures de données**.
- Un Stream **ne stocke pas de données**, ce qui le différencie des Collections classiques.
- Il ne **modifie pas les données de la source originale**, garantissant la cohérence des données lors du traitement.

Introduction 2/3

Un autre aspect de l'API Stream est son **chargement paresseux des données** :

- Les données ne sont **chargées que lorsque nécessaire**, **optimisant** les **performances** des applications.

Les opérations sur les Streams sont divisées en **deux catégories** :

- **Opérations intermédiaires** (comme map ou filter) : paresseuses et renvoient **un nouveau Stream**.
- **Opérations terminales** : consomment le Stream et produisent **un résultat final**.

Introduction 3/3

En résumé, l'API Stream de Java 8 offre une approche moderne et flexible pour **manipuler les données** :

- Modèle **paresseux** et fournissant des **opérations puissantes**.
- **Simplifie** et **optimise** les **traitements** sur les **Collections** et les **tableaux** en Java.
- Un **stream** est représenté par une instance de l'**interface générique** `Stream`, on peut le **visualiser comme une collection** mais ce n'en est pas une !

Pour plus de détails : Doc Stream Java

Création d'un stream

- Utilisation d'un objet `Stream.Builder<T>`

```
Stream<String> stream = Stream.<String>builder()
                              .add("element1")
                              .add("element2")
                              .build();
```

- À partir d'un **tableau** grâce aux méthodes `Arrays.stream`

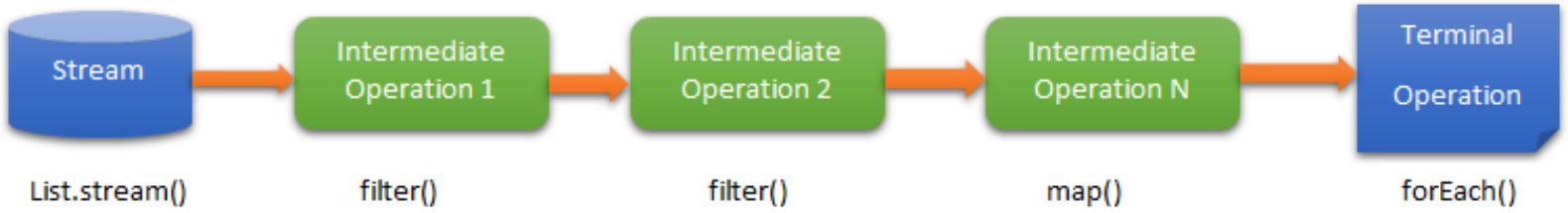
```
int[] array = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(array);
```

- À partir d'une **collection** grâce à la méthode Collection.stream

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> stream = list.stream();
```

Création d'un stream

- Après la création d'un stream, on peut appliquer :
  - Une succession d'**opérations intermédiaires** (filter(), map(),...)
  - Une **opération terminale** (forEach(), collect())
- Les opérations intermédiaires **renvoient un Stream**, permettant de les **enchaîner**. On parle de **"Méthodes Chaînées"**
- L'opération terminale renvoie **un autre type** (ou **void**).



Exemple

```
// Création d'une liste de prénoms
List<String> prenoms = Arrays.asList(
  "Alice", "Bob", "Caroline", "David", "Anna", "Catherine");

// Filtrer pour ne garder que les prénoms commençant par "A"
List<String> prenomsCommencantParA = prenoms.stream()
            .filter(prenom -> prenom.startsWith("A"))
            .collect(Collectors.toList());
System.out.println("Prénoms commençant par 'A': "
    + prenomsCommencantParA);

// Compter les prénoms contenant un "v"
long countWithC = prenoms.stream()
                         .filter(prenom -> prenom.contains("v"))
                         .count();

System.out.println("Nombre de personnes dont le prénom contient un 'v': "
    +countWithC);
```

Opérations intermédiaires 1/2

Pour manipuler un stream, plusieurs **opérations intermédiaires** sont possibles :

- `filter(Predicate)` : renvoie un Stream contenant les éléments pour lesquels l'évaluation du Predicate vaut true
- `distinct()` : renvoie un Stream contenant uniquement les éléments uniques (retire les doublons)
- `limit(n)` : renvoie un Stream contenant le nombre d'éléments spécifié

Opérations intermédiaires 2/2

- `skip(n)` : renvoie un Stream dont les n premiers éléments sont ignorés
- `map(Function)` : applique la Function fournie en paramètre pour transformer l'élément
- `flatMap(Function)` : applique la Function pour transformer l'élément en zéro, un ou plusieurs éléments

Opérations terminales 1/2

Les **opérations terminales consomment le stream** et génèrent un **résultat** :

- `reduce()` : applique une Function pour combiner les éléments afin de produire le résultat
- `collect()` : transforme un Stream pour contenir le résultat des traitements dans un conteneur mutable
- `anyMatch(Predicate)` : renvoie un booléen indiquant si l'évaluation du Predicate sur au moins un élément vaut true

Opérations terminales 2/2

- `allMatch(Predicate)` : renvoie un booléen indiquant si l'évaluation du Predicate sur tous les éléments vaut true
- `noneMatch(Predicate)` : renvoie un booléen indiquant si l'évaluation du Predicate sur tous les éléments vaut false
- `findAny()` : renvoie un Optional encapsulant un élément du Stream s'il existe
- `findFirst()` : renvoie un Optional encapsulant le premier élément du Stream s'il existe

Optional

- **Optional** est une classe introduite dans Java pour représenter une valeur qui **peut être absente**.
- Elle permet de rendre **explicite** la possibilité qu'un **résultat puisse être nul**, aidant à éviter les erreurs de type `NullPointerException`.
- Les `Optional` sont souvent utilisés avec les `Stream` pour représenter une valeur qui **peut être absente**
- Exemple : Lorsqu'une opération de recherche dans un Stream ne trouve pas de résultat

Optional

Les **Optional** fournissent plusieurs méthodes intéressantes :

- `isPresent()` : Vérifie si une valeur est présente.
- `get()` : Récupère la valeur si présente, sinon lance une exception.

Pour plus de détails : Doc Optional

Exemple Optional

```
public static void main(String[] args) {
    List<String> list = Arrays.asList("apple", "banana", "cherry");

    // Recherche d'un élément dans la liste
    Optional<String> result = list.stream()
                                  .filter(s -> s.startsWith("p")) // Filtre pour les éléments commençant par "p"
                                  .findFirst(); // Récupère le premier élément correspondant

    // Vérification de la présence de l'élément
    if (result.isPresent()) {
        System.out.println("Premier élément commençant par 'p': " + result.get());
    } else {
        System.out.println("Aucun élément commençant par 'p' trouvé.");
    }
}
```

Traitement de fichier par flux

- Les streams sont couramment utilisés pour le traitement de **flux de données** en Java, y compris pour **la lecture et le traitement de fichiers**.
- Ils offrent une approche **fonctionnelle** et **expressive** pour traiter des **collections** de données, des **flux** d'entrée/sortie et d'autres **sources de données**.

Traitement de fichier par flux

- Cependant, les streams **ne conviennent pas à tous les types de traitement de flux**, en particulier pour les opérations nécessitant **un accès aléatoire** aux données ou pour les **fichiers de très grande taille** où la **gestion de la mémoire** pourrait devenir un **problème**.
- Dans de tels cas, d'autres approches comme l'utilisation de **flux d'entrée/sortie traditionnels** pourraient être **plus appropriées**.

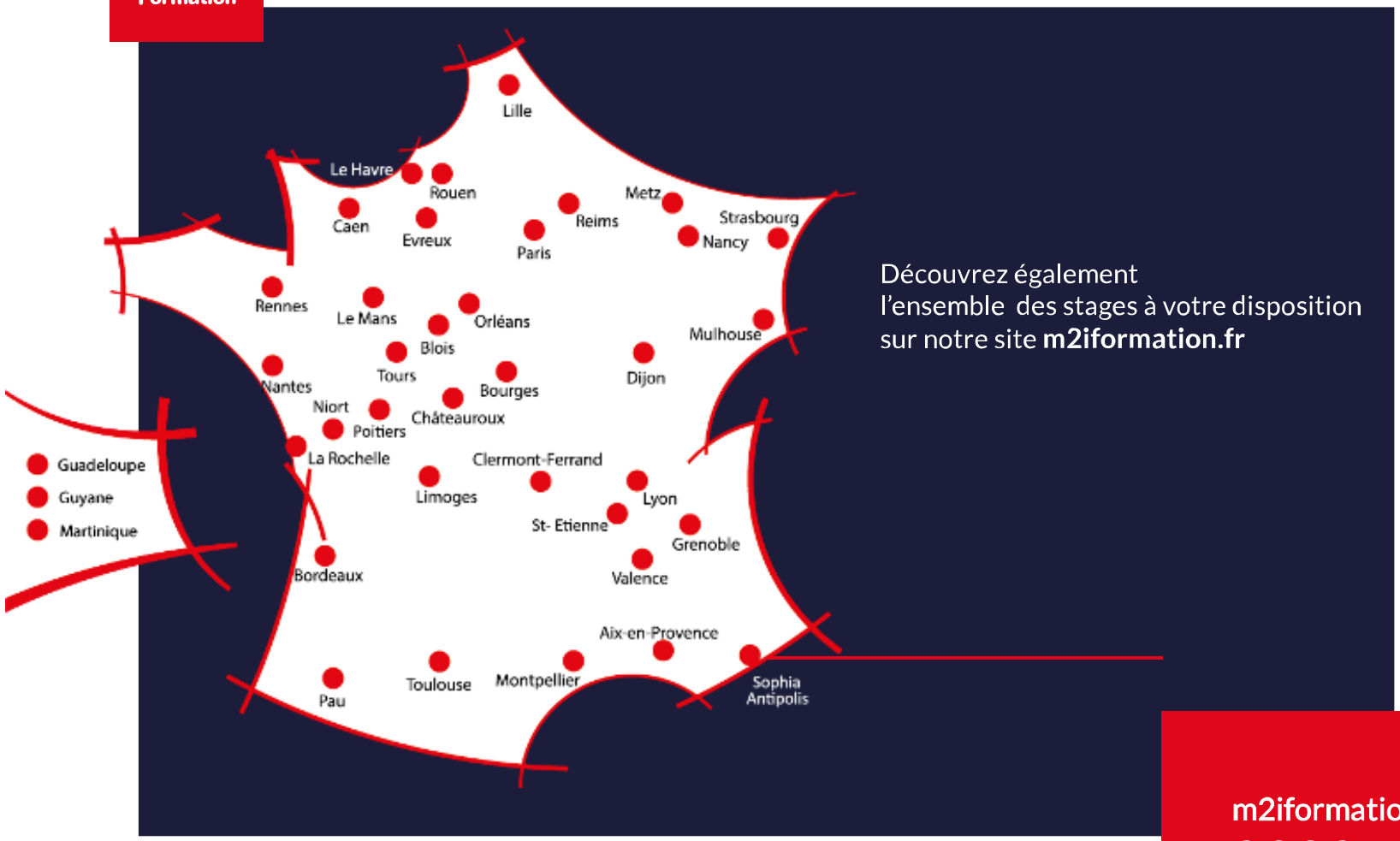
Exemple Traitement de fichier par flux

```java
public static void main(String[] args) {
    String csvFile = "src/main/resources/users.csv";

    try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {
        br.lines() // Crée un stream de lignes du fichier
          .skip(1) // ne tiens pas compte de la premiére ligne
          .map(line -> line.split(",")) // Sépare chaque ligne en un tableau de valeurs
          .forEach(array -> System.out.println(Arrays.toString(array))); // Affiche chaque tableau
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Les fichiers avec l'extension `.csv` servent à représenter des tables/tableaux avec un délimiteur pour les lignes (`\n`) et un délimiteur pour les colonnes (`,` , `;` , `|`)

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr