Développement JAVA

JAVA : LES FONDAMENTAUX

© Boris SAUVAGE - 2023

Objectifs pédagogiques

- Connaître la sémantique du langage orienté objet Java et mettre en pratique les notions

Sommaire

- **Introduction**
  - Le langage Java, l’historique et les différentes versions de Java

- **Le langage Java et son fonctionnement**
  - Présentation de la JVM du JRE, du JDK

Développement JAVA

INTRODUCTION

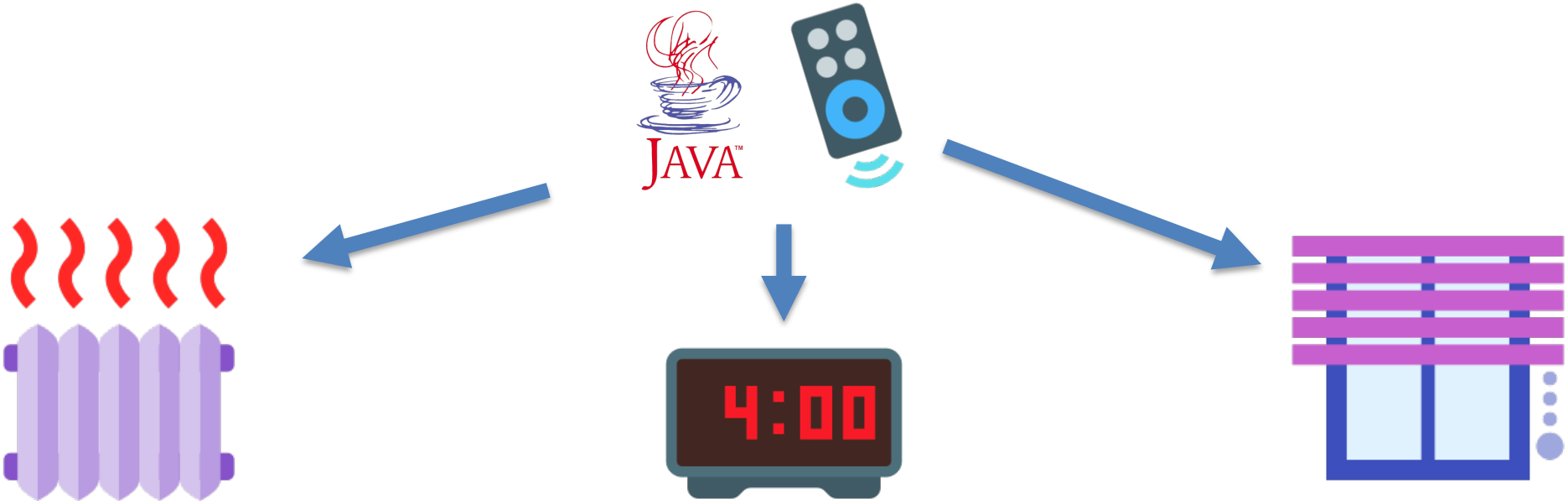
Introduction

Java

- Langage orienté objet
- Officiellement présenté en mai 1995 au SunWorld
- Créé par James Gosling et Patrick Naughton (employés de Sun Microsystems)
- Sun Microsystems est racheté par Oracle en 2009
- Langage avec une grande portabilité (Unix, Windows, MacOS)
  - Un programme développé en Java s’exécutera sur toutes les plateformes

Historique 1/4

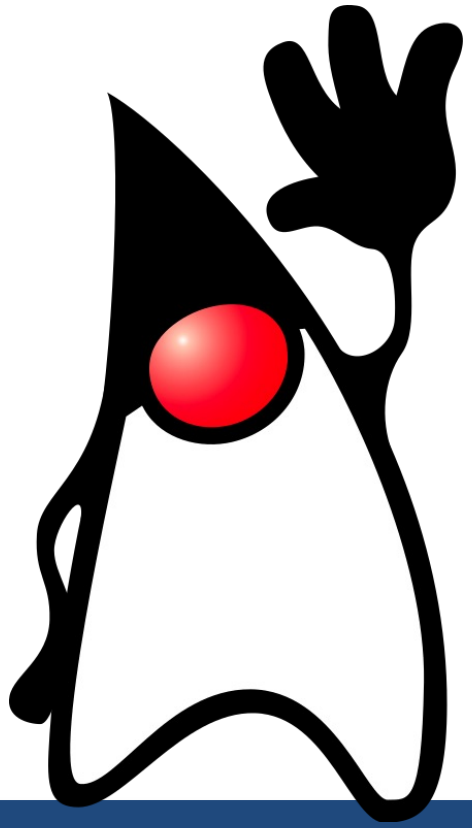
- **1990 : Le Green Project**
  - Une équipe d'ingénieurs de Sun conçoit un nouveau langage électronique grand public
  - Langage portable, simple et compact
  - Devait permettre à des appareils ménagers hétérogènes de communiquer entre eux



Historique 2/4

- **1992 : Début prometteur**
  - Création du langage Oak par Grosling
  - Première démonstration (PDA Star 7)
  - Présentation de la mascotte de java : Duke
  - Création de la filiale FirstPerson, Inc.

- **1994 : Marché de la domotique ne décolle pas**
  - Sun doit faire un choix :
    - Stopper le projet, et attendre que le marché se développe
    - Trouver un autre domaine pour utiliser ce langage



© Boris SAUVAGE - 2023

Historique 3/4

- **Début 1995 : Le marché de l'Internet est en pleine explosion**
  - Les sites Internet statiques et Sun travaille sur son propre navigateur Web
  - Recherche d'un moyen pour télécharger des applications à exécuter dans le navigateur
  - La **Green Team** réalise que l'Internet aura besoin d'un langage multiplateformes

« WebRunner était juste une démo, mais une démo impressionnante :

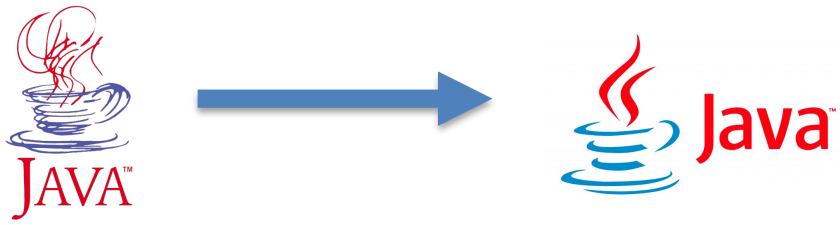
Il a porté à l'écran, pour la première fois, des objets animés et un contenu dynamique exécutable à l'intérieur d'un navigateur Web […] »

Historique 4/4

- **Mai 1995 : Sun met à disposition le code source de Java sur internet**
  - En quelques mois, Java devient célèbre
  - Plusieurs milliers d'utilisateurs le téléchargent chaque mois

- **Fin 1995 : Netscape intègre Java dans son navigateur**
  - Java devient potentiellement utilisable par des millions d'Internautes

- **1996 : Microsoft intègre Java dans Internet Explorer**
  - Sun rallie de nombreux partenaires (IBM, Oracle, Netscape…) autour de Java



Développement JAVA

FONCTIONNEMENT DE JAVA

Le Java : plateforme ou langage ?

- Java n'est pas seulement un langage de programmation

- Java est composé de 2 plateformes : **JRE** et **JDK**

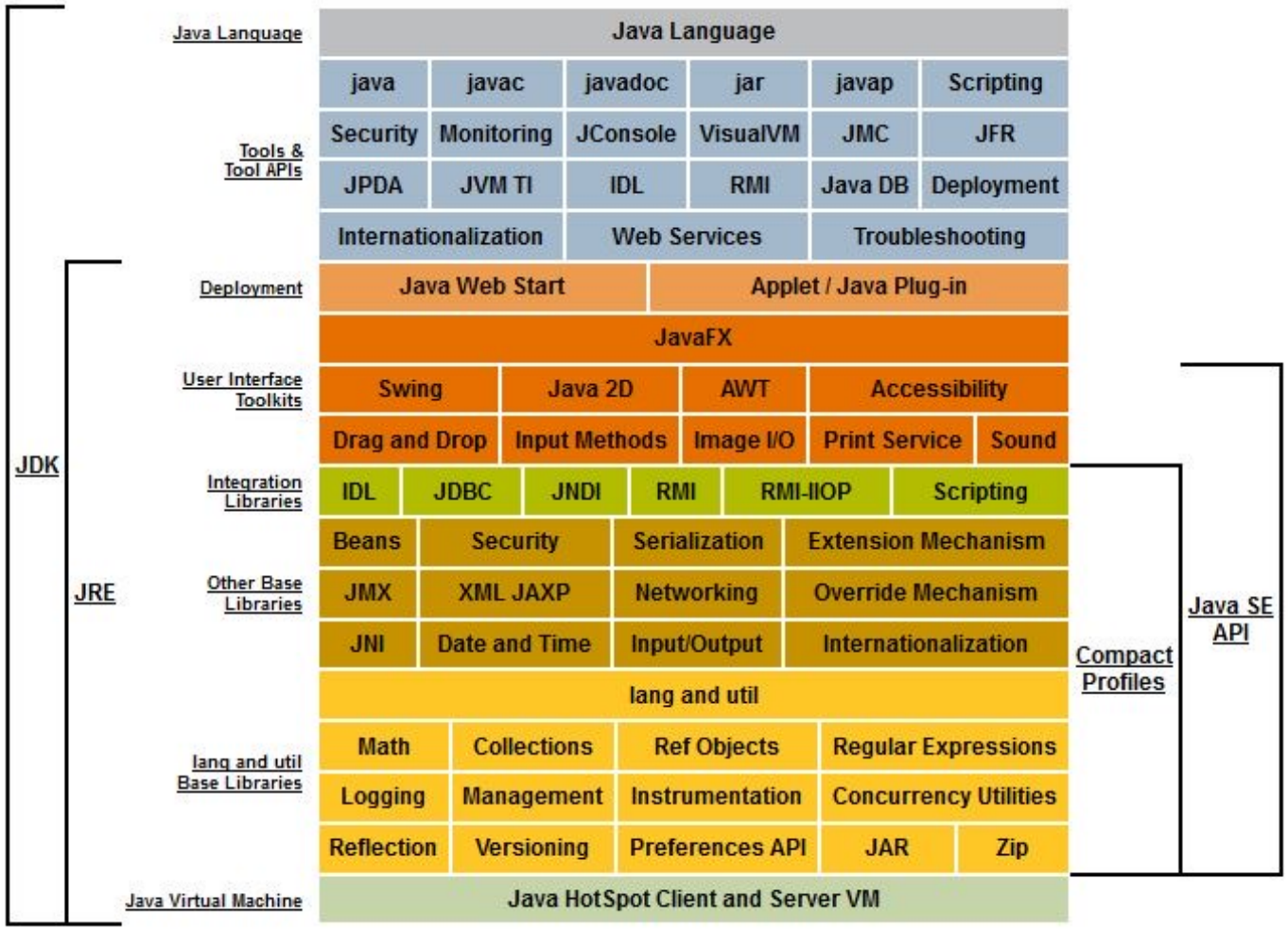
- **JRE : Environnement d'exécution (J**ava **R**untime **E**nvironnement**)**
  - **La machine virtuelle** qui exécute les programmes Java (**J**ava **V**irtual **M**achine)
  - Un ensemble de classes et bibliothèques pour l'exécution des programmes

- **JDK : Kit pour développer en Java (J**ava **D**evelopment **K**it**)**
  - Ensemble de logiciels utilisables pour développer les applications Java
  - Contient un compilateur, un ensemble de classes API et le JRE

© Boris SAUVAGE - 2023

Notions fondamentales

Du langage jusque la JVM



Le langage Java 1/6

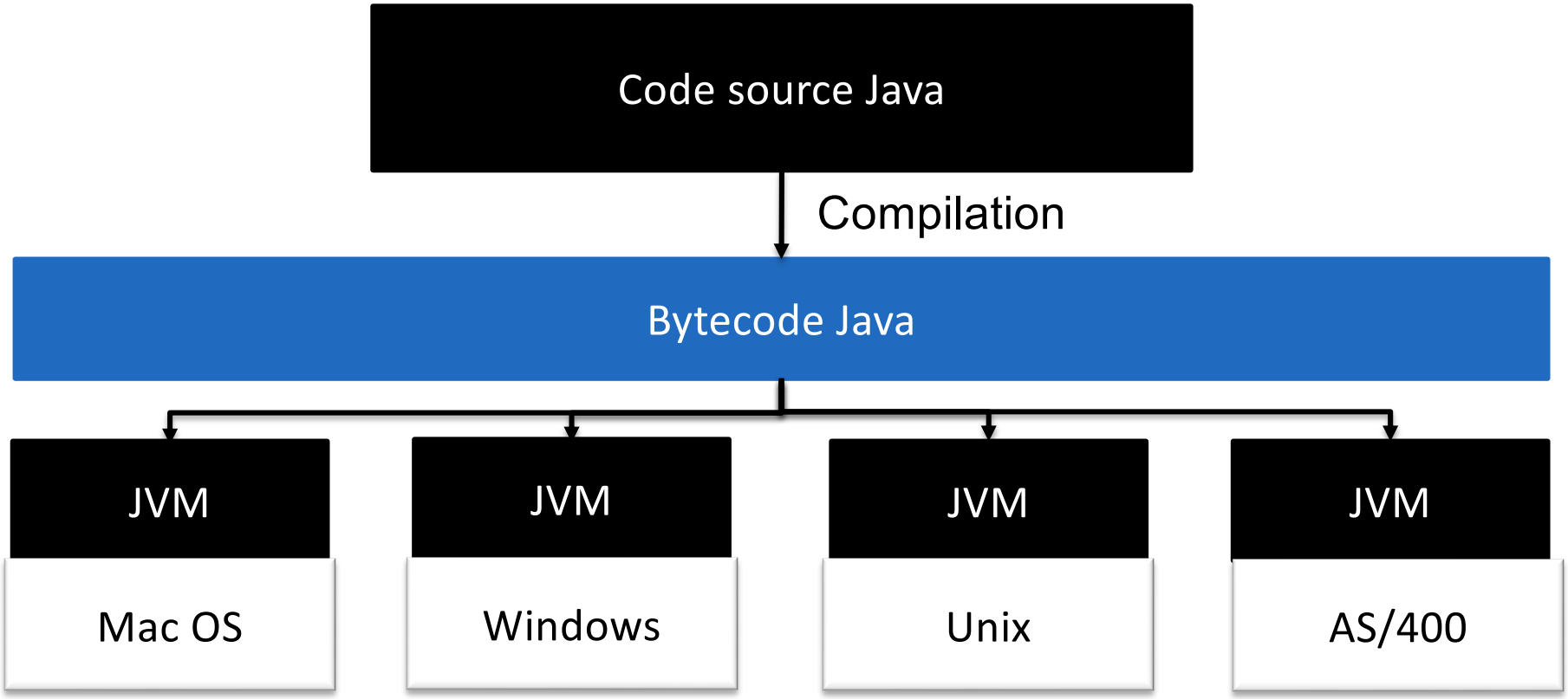
- **Portable**
- **Compact**
- **Robuste**
- **Orienté Réseau**
- **Multitâche**

© Boris SAUVAGE - 2023

Fonctionnement de Java

Le langage Java 2/6

- **Langage portable**
  - Un même code source s'exécute sur toutes les plateformes grâce à une machine virtuelle



Le langage Java 3/6

- **Langage compact**
  - Le Bytecode Java généré par la compilation dispose d'instructions de haut niveau
  - Aspect privilégié depuis l'origine de Java
  - La machine virtuelle fait une grande part du travail

© Boris SAUVAGE - 2023

Le langage Java 4/6

- **Langage robuste**
  - Le développeur ne manipule pas directement la mémoire
    - Pas de notion de pointeur
  - La machine virtuelle s'occupe de la libération de la mémoire
    - Les objets sont détruit par le **Garbage Collector** (ramasse miettes)
    - Travail du développeur simplifié
    - Pas de risque de libérer deux fois la même zone ou d'oublier de libérer

<u>Exemples :</u>

- Il y a un contrôle de type très strict
- Les attributs sont initialisés
- Les bornes des tableaux sont contrôlées
- Les erreurs potentielles doivent être gérée avec un système d'exceptions

© Boris SAUVAGE - 2023

Fonctionnement de Java

Le langage Java 5/6

- **Langage orienté réseau**
  - Depuis toujours, en commençant par les Applets, Java est adapté pour l'Internet
  - De nombreuses bibliothèques sont disponibles pour simplifier les échanges
    - Client HTTP
    - Client FTP
    - Communication entre objets distants
    - etc

Le langage Java 6/6

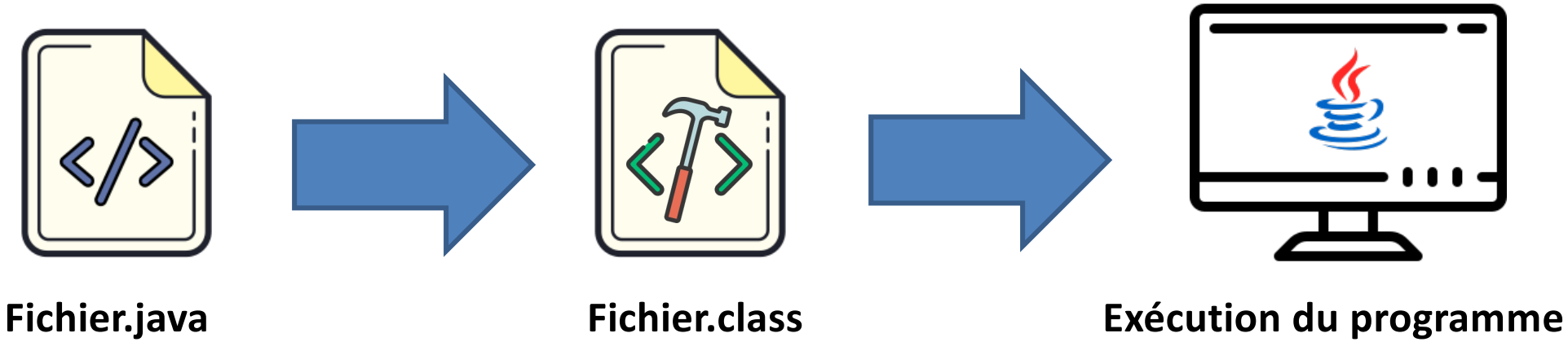
- **Langage multitâche**
  - Intégration native de la notion des Threads
    - La JVM peut gérer plusieurs parties d'un programme simultanément
  - La JVM est capable de profiter de plusieurs processeurs s'ils sont disponible
    - Possibilité de bénéficier d'un réel parallélisme
    - Portabilité disponible pour les programmes multithreading
  - Contexte adapté à l'exécution d'une application serveur
    - Traitement simultané des requêtes de plusieurs utilisateurs

Fonctionnement de Java

Projet Java 1/2

- Constitué de fichiers portant l'extension **.java**
  - Les **fichiers .java** contiennent le code source du projet

- Le projet java est compilé et exécuté grâce à la machine virtuelle Java
  - La JVM est incluse dans le JRE



Projet Java 2/2

Au sein d'un projet Java, le code Java est représenté :

- **Physiquement**
  - Les fichiers sont organisés en répertoires
  - Initialement avec l'extension .java : **Example.java**
  - Après compilation, l'extension sera .class : **Example.class**

- **Logiquement**
  - Les classes sont organisées en paquetages **(packages)**

© Boris SAUVAGE - 2023

Développement JAVA

LES VERSIONS

Les différentes versions Java 1/6

- Java 1 : Janvier 1996
- Java 1.1 : Janvier 1997
- Java 1.2 : Décembre 1998
- Java 1.3 : Mai 2000
- Java 1.4 : Février 2002
- Java 1.5 : Septembre 2004
- Java 6 : Décembre 2006
- Java 7 : Juillet 2011

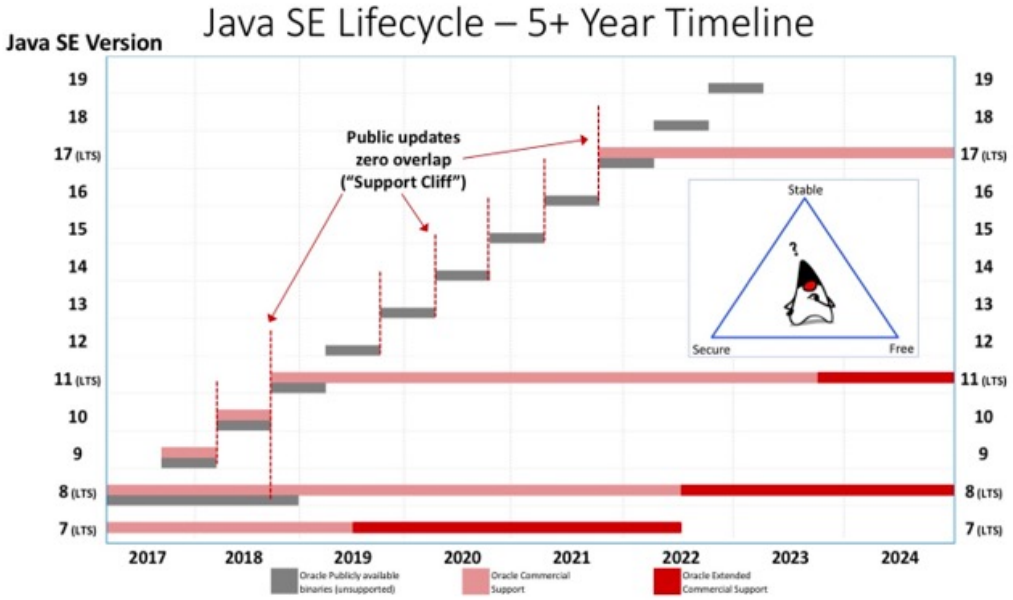
© Boris SAUVAGE - 2023

Les différentes versions Java 2/6

Avant, il pouvait y avoir 5 années entre 2 nouvelles versions Java)

Désormais, le planning sera le suivant :

- 2 versions Majeures tous les 6 mois (Mars et Septembre)
- 1 version LTS (Long Term Support) tous les 3 ans à partir de Java 11



Les différentes versions Java 3/6

- **Java 8 : Mars 2014**
  - Ajout des lambdas avec utilisations avec les collections et les streams
  - Les optionnels
  - Les méthodes par défaut des interfaces
  - Les dates
- Java 9 : Septembre 2017
  - Projet Jigsaw (modularité chargés au sein du JDK)
  - Projet Kulla (création d'un Shell sous le format REPL)
  - Projet Valhalla (amélioration des types Java)
  - Un nouveau HTTP Client avec gestion native du JSON
- Java 10 : Mars 2018
  - JEP 286 (Inférence des types des variables locales)
  - Amélioration des APIs *Stream* et *Optional*

© Boris SAUVAGE - 2023

Les différentes versions Java 4/6

- **Java 11 : Septembre 2018**
  - **Support Long terme (Mars 2022 et Support étendu jusque Décembre 2030)**
  - Intègre les nouveautés (Inférence, API HttpClient, Suppression de modules)
  - Amélioration des paramètres des lambda

- Java 12 : Mars 2019
  - Optimisation du Garbage Collector
  - Les Expressions Switch
  - L' API Constants (méta données dans les fichiers .class, utile pour les langages sur JVM)
  - Default CDS Archives (chargement des informations des classes de la JVM plus rapide)

- Java 13 : Septembre 2019
  - JEP 355 (Text Blocks)
  - JEP 354 (Switch)

© Boris SAUVAGE - 2023

Les différentes versions Java 5/6

- Java 14 : Mars 2020
  - JEP 343 (Packaging Tool, Module Incubateur pour diffuser et recueillir des commentaires pour des outils)
  - JEP 358 (Compléments sur les *NullPointerExceptions*)
  - JEP 361 (Switch avec les cas fléchés)
  - JEP (Text Blocks)

- Java 15 : Septembre 2020

- Java 16 : Mars 2021

- **Java 17 : Septembre 2021**
  - **Support Long terme (Septembre 2026 et Support étendu jusque Septembre 2029)**

© Boris SAUVAGE - 2023

Les différentes versions Java 6/6

- Java 18 : Mars 2022
- Java 19 : Septembre 2022
- Java 20 : Mars 2023
- **Java 21 : Septembre 2023**
  - **Support Long terme (Septembre 2023 et Support étendu jusque Septembre 2031)**
- Il est possible de comparer les différentes versions Java :
  - https://javaalmanac.io

© Boris SAUVAGE - 2023

Les versions

Choix de la version Java

- Les versions LTS sont Java 8, Java 11, Java 17
  - Les lambdas sont mieux intégrés sur Java 13/14 que sur Java 8
  - Démarrer un nouveau projet sur Java 13 et stabiliser pour passer sur Java 14
  - Dans les cas spécifiques, faire un projet sur Java 9-10-12

- Oracle distribue deux JDKs :
  - OracleJDK : **(favorise la stabilité)**
    - Support disponible avec système de licence pour commercialisation
    - Utilisation en production avec le modèle d'abonnement Java SE obligatoire
  - OpenJDK **(intègrera les features release – 6 mois)**
    - Complétement gratuit
    - Utilisation de la License GNU avec exception classpath (Pas besoin de fournir le code source)
    - Intégration de fonctionnalités payante du OracleSDK, en gratuit (Java Flight Recorder…)

Les versions

Java 8

- Grâce à cette nouvelle version, Java est repassé langage de l'année (index TIOBE)
- Nouveautés apportés :
  - Les expressions lambda
  - Les références de méthodes
  - La programmation fonctionnelle
  - L'API Stream et Collector
  - L'API Date & Time
  - La classe Optional
  - Les CompletableFuture
  - La Permgen (Permanent Generation) remplacée par le Metaspace
  - Les méthodes statiques et par défaut sur les interfaces
- Java est devenu plus complet et complexe

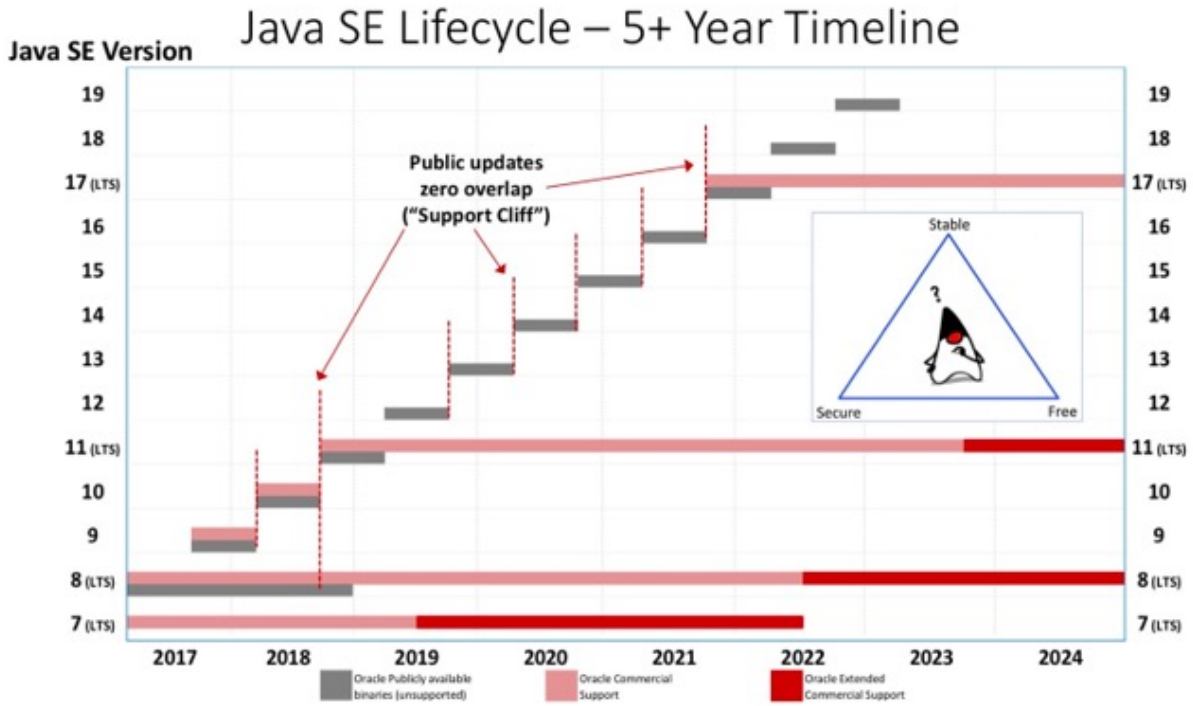
Les versions

2017 : Le nouveau modèle de release

Avant, il pouvait y avoir 5 années entre 2 nouvelles versions Java (**S**table, **S**écurisées, **G**ratuites)

Désormais, le planning sera le suivant :

- Deux versions Majeures tous les 6 mois (Mars et Septembre)
- Une Version LTS (Long Term Support) tous les 3 ans à partir de Java 11



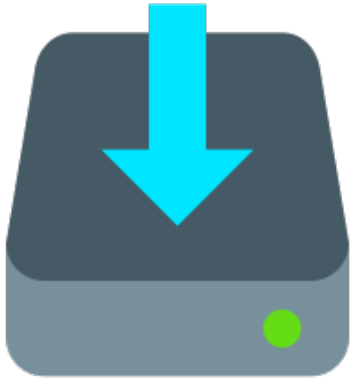
Les versions

2017 : Le nouveau modèle de release

- **Sécurité** et **gratuité :** rythme de migration d'Oracle pour bénéficier des dernières mises à jour de sécurité gratuitement mais plus jeunes et moins stables
- **Stabilité** et **gratuité :** utilisation des version LTS tous les 3 ans
  - versions plus stables car les LTS seront plus stables que les releases tous les 6 mois
  - Sans payer, pas le droit aux dernières mises à jour de sécurité
- **Stabilité** et **sécurité :** utilisation des versions LTS avec mises à jours
  - Paiement de la License Oracle pour les mises à jour
- Les entreprises doivent modifier la façon de fonctionner en choisissant le format qui leur correspondra et les 2 composantes sur 3 du triangle **stabilité-sécurité-gratuité**.

Développement JAVA

INSTALLATION & PROJET



Installation

Java et ses outils

Pour développer en Java, il est nécessaire de mettre en place une configuration

- Nous téléchargerons et utiliserons un IDE :

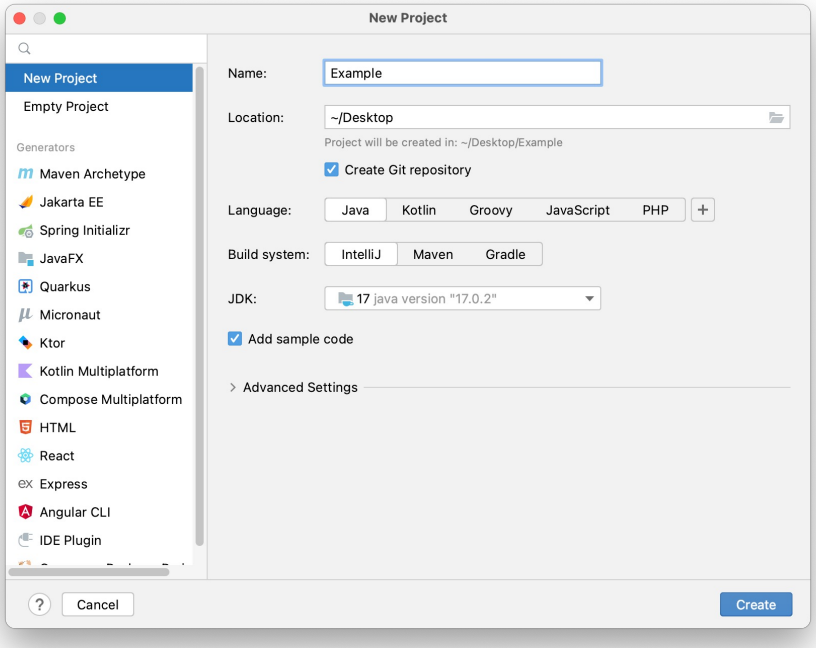
- Nous téléchargerons et utiliserons un JDK (si l'IDE ne le fait pas) :
  - https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

Installation

Création d’un projet Java

Lancement et création d’un projet sur votre IDE :

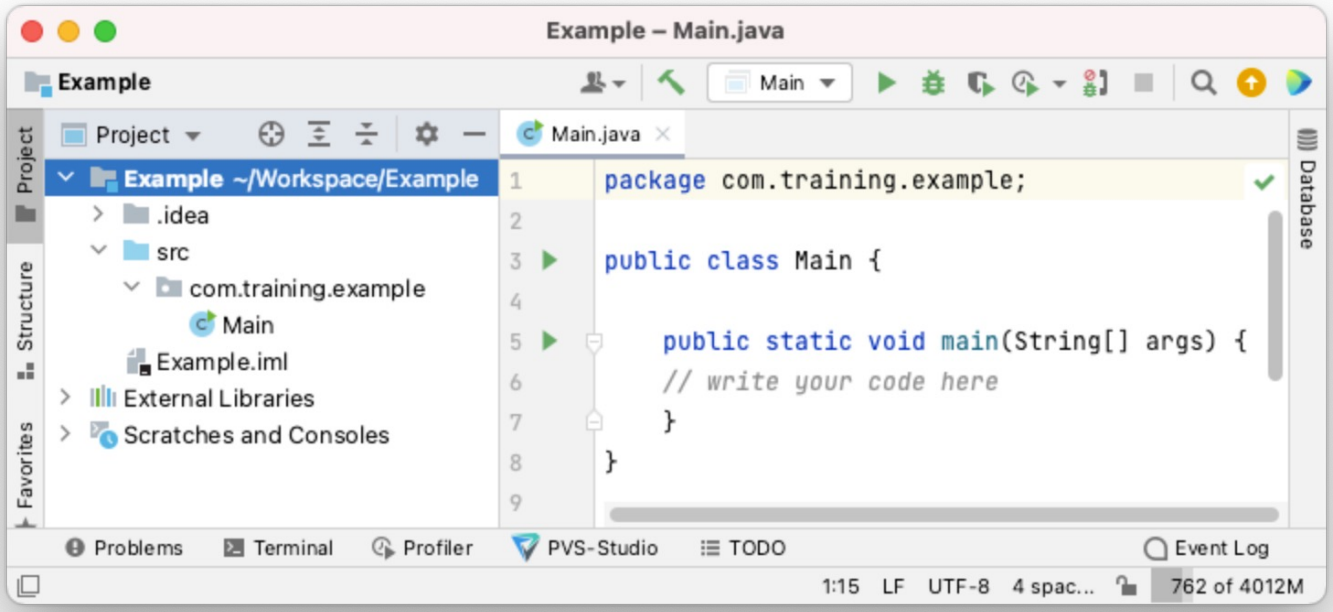
- Sélection du type de projet **Java**, avec un Project SDK équivalent à votre JDK (Java 17)
- Création depuis un *template* (Le projet créé inclura une classe avec un **main()**)
- Précision du nom du projet, de son chemin et du paquet principal



Structure d’un projet Java

Visualisation du projet créé

- Le dossier source **(src)** contiendra votre package principal et vos sources
- Le package com.training.example contiendra vos classes Java
- Le dossier **.idea** et le fichier **.iml** appartiennent à la configuration



Installation

Compilation et exécution

- Nous pouvons compiler et exécuter notre projet avec le bouton

Example – Main.java

src > com > training > example > Main > main

Main

Project

Example ~/Workspace/Example

.idea

out

src

com.training.example

Main

Example.iml

External Libraries

Scratches and Consoles

Main.java

```
1  package com.training.example;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.println("Hello Java !");
7      }
8  }
```

Run: Main

```
Hello Java !
```

Run | Problems | Terminal | Profiler | PVS-Studio | TODO | Build | Event Log

Build completed successfully in 3 sec, 222 ms (moments ago) 5:1 LF UTF-8 4 spac... 1127 of 4012M

Exercice

- Installer les outils
- Créer un premier projet java qui affiche votre ***Nom*** *et* ***Prénom***
- Exécuter le projet

Développement JAVA

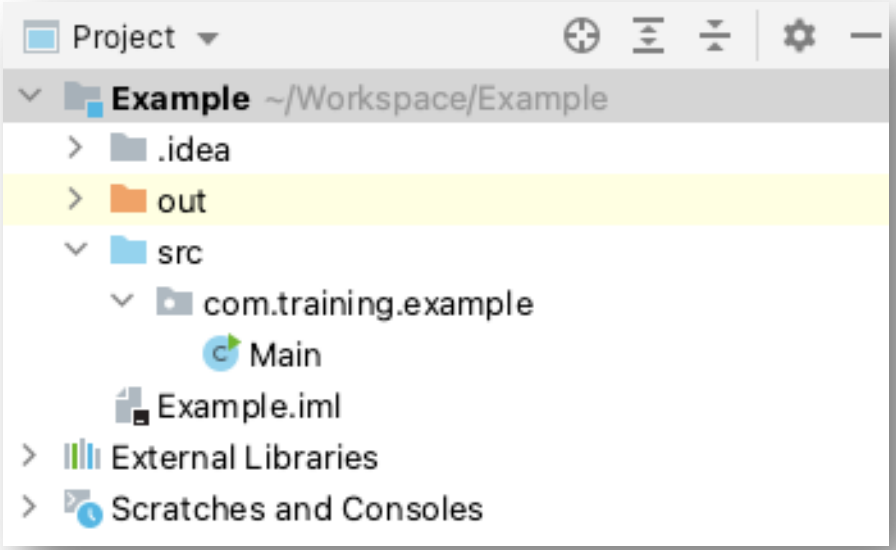
ZOOM SUR LE CODE

Zoom sur le code

Projet Java SE

Un programme Java SE est constitué de plusieurs parties :

- **src :** dossier dans lequel se situent les fichiers code source
- **out :** dossier contenant la compilation du projet
- **package :** nom de domaine inversé contenant vos classes

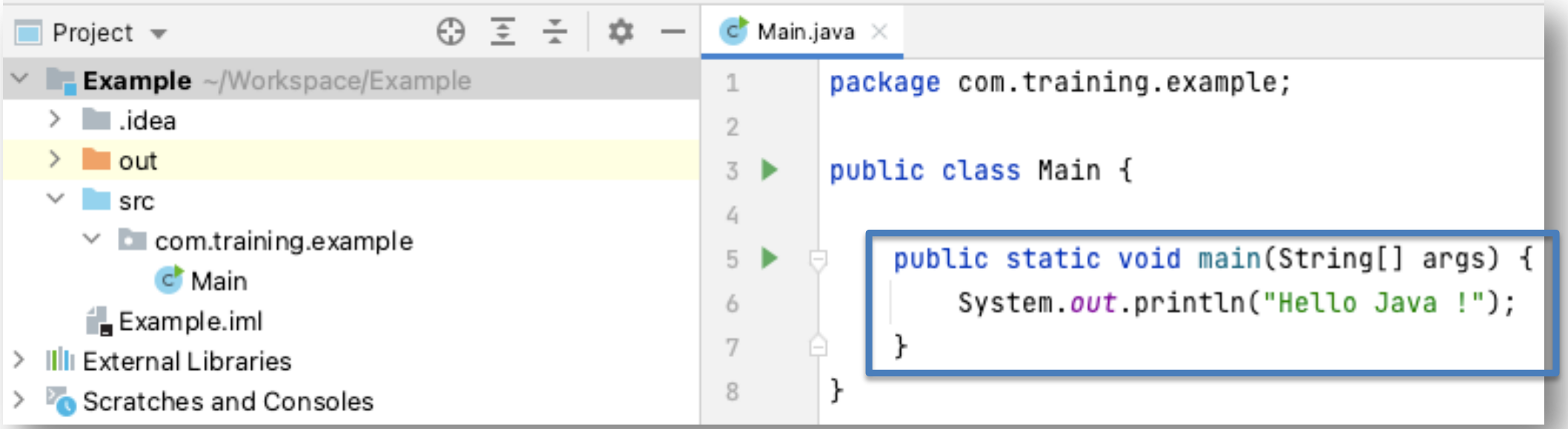


Zoom sur le code

Projet Java SE

Un projet Java SE doit contenir un point d'entrée pour exécuter votre programme

- main(..) sera toujours écrit de la même manière
- Une classe de votre projet devra comporter cette méthode

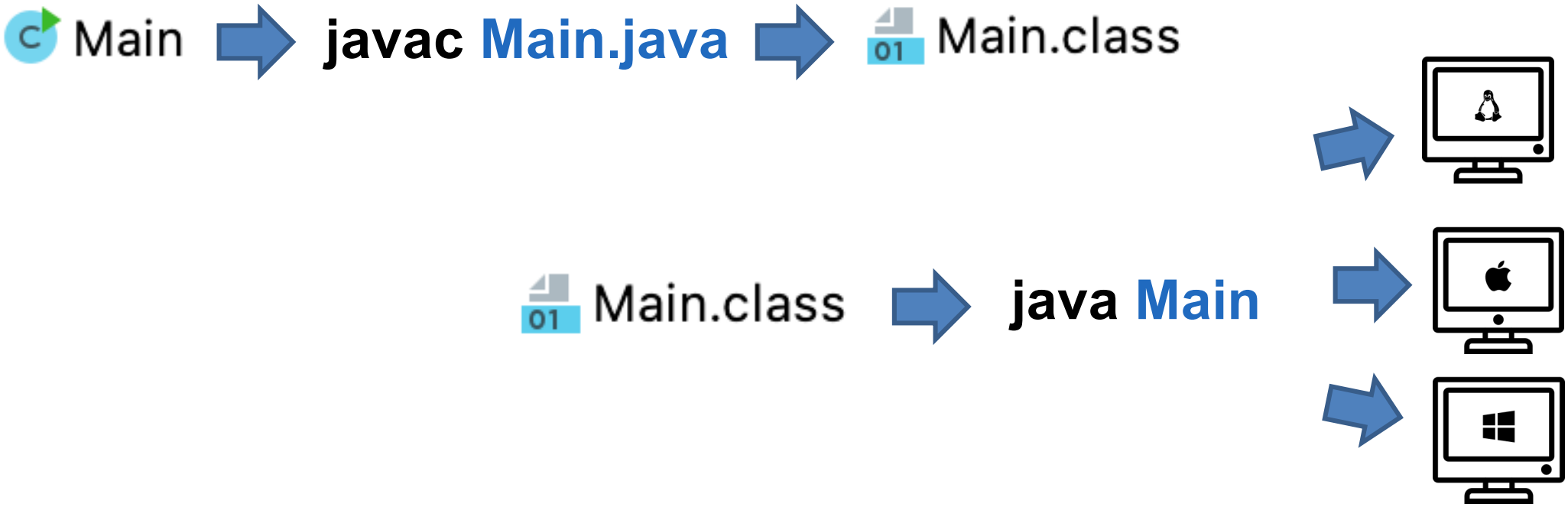


Zoom sur le code

Utilisation de javac et java 1/2

Pour fonctionner, un projet Java SE doit être compilé puis exécuter par la JVM

- **javac :** commande pour compiler le projet
- **java** : commande pour exécuter le programme



Zoom sur le code

Utilisation de javac et java 2/2

Depuis Java 9, il est possible d'utiliser la commande java pour compiler et exécuter :

Développement JAVA

JAVA SE - JAVA EE – JAVA ME

JAVA SE - JAVA EE – JAVA ME

- Java SE (Java Standard Edition)
  - utilisé pour le développement d'applications qui s'exécutent uniquement sur le système
- Java EE (Java Enterprise Edition)
  - utilisé pour développer des applications Web qui s'exécutent à l'aide du navigateur Web
- Java ME (Java Micro Edition)
  - utilisé pour le développement d'applications mobiles comme Android

© Boris SAUVAGE - 2023

Développement JAVA

LES BASES DU LANGAGE

Sommaire

- Structure du langage
- Les variables
- L’affichage
- Les commentaires
- Le formatage du code
- Les littéraux et les traits de soulignements
- Les types primitifs
- Les types objets

© Boris SAUVAGE - 2023

Structure du langage

Java est un langage de programmation impératif

- Un programme Java est composé d'instructions délimitées par **un point-virgule**

```
int age = 30;
 System.out.print("L'âge est de : " + age);
```

- Java permet de structurer le code en bloc, délimités par des **accolades**
  - Permet d'isoler le code conditionnel ou encore de créer des blocs anonymes

```
int age = 30;
if(age > 18) {
 System.out.print("Majorité atteinte");
}
```

```
{
    int age = 30;
}
```

Les variables 1/3

- Au sein des programmes, Java permet de définir des variables
  - Une variable permet d'associer un nom/identifiant à une valeur
  - Une variable est dynamique, elle peut changer de valeur au cours du temps

- La syntaxe est la suivante :

type nomDeLaVariable = valeur;

Exemple :

```
public static void main(String[] args) {

    int age = 30;
    System.out.println(age);

}
```

```
30
```

Les variables 2/3

- Java est fortement typé, il fournit de nombreux types pour nos variables :

```
public static void main(String[] args) {

    char c = 'B';
    boolean b = false;

    byte bb = 127;
    short s = 32767;
    int i = 2147483647;
    long l = 9223372036854775807L;

    float f = 3.4028235E38f;
    double d = 1.7976931348623157E308;
}
```

Les variables 3/3

- Depuis Java 10, il est possible d'utiliser l'identifiant **var** à la place du type
  - Cette syntaxe permet d'éviter la répétition quand le type est évident.

```
public static void main(String[] args) {

    var maVariable = 0;
}
```

- Le compilateur déterminera le type de la variable grâce à la valeur d'initialisation
  - Ici, la valeur 0 est affectée à la variable maVariable
  - le compilateur déduit que le type de cette variable est int (on dit qu'il infère le type)

- L'inférence de type repose sur la valeur d'initialisation
  - var est donc limité à la déclaration de variables précisant une valeur d'initialisation

© Boris SAUVAGE - 2023

L’affichage

- Ces variables peuvent être manipulées, changées, affichées

```
public static void main(String[] args) {
   int age = 30;
   System.out.println(age);
   age = age + 12;
   System.out.print("L'âge modifié est : " + age);
}
```

```
30
L’âge modifié est : 42
```

- System.*out*.print(*valeur*) permet d’afficher une information dans la console

- System.*out*.print**ln**(*valeur*) affiche et retourne à la ligne
  - Place le curseur sur la ligne suivante après l’affichage **(\n)**

Les commentaires

- En Java, il est possible d'indiquer des portions de code ignorées par le compilateur
  - Permettent d'indiquer aux développeurs une précision dans le code
  - N'influencent pas l'exécution du programme
- Ces **commentaires** sont disponible sur une seule ou plusieurs lignes

```
public static void main(String[] args) {
    /* Exemple de commentaire
                        sur plusieurs lignes */
    int age = 30;
    // Exemple de commentaire sur une seule ligne
    System.out.println(age);
}
```

© Boris SAUVAGE - 2023

Le formatage du code

- En Java, il est recommandé d'utiliser la convention de formatage du code
  - Le code source est très souvent relu par les développeurs

```
public static void main(String[] args) {
   double i=0.0;i= Math.sqrt(16);if(i>1){i=-i;}
}
```

```
public static void main(String[] args) {
   double i = 0.0;
   i = Math.sqrt(16);
   if (i > 1) {
      i = -i;
   }
}
```

- *Intellij* propose de reformater votre code dans ***Code / Reformate Code***

- La convention de code défini des règles à respecter :
  - https://www.oracle.com/java/technologies/javase/codeconventions-contents.html
  - https://google.github.io/styleguide/javaguide.html

Développement JAVA

LITTÉRAUX & TYPES PRIMITIFS

© Boris SAUVAGE - 2023

Variable vs Littérale

Variables vs données littérales

- Les variables sont des emplacements de mémoire réservés qui stockent des valeurs avec des noms symboliques

- Les littéraux de données sont des représentations de code source de valeurs fixes

Association

- Les variables sont associées à l'emplacement de la mémoire

- Les littéraux de données sont associés à des valeurs fixes placées à l'intérieur des variables

© Boris SAUVAGE - 2023

Les littéraux en Java

Un littéral Java est un élément syntaxique représentant une valeur

- Les littéraux booléens
- Les littéraux de caractère
- Les littéraux entiers
- Les littéraux à virgule flottante
- Les littéraux de chaîne de caractères
- Les littéraux nul
- Les littéraux hexadécimaux, décimaux, octaux, binaires

© Boris SAUVAGE - 2023

Les littéraux booléens

- 2 valeurs booléens possibles sont représentées par les littéraux **true** et **false**

- Ils sont sensibles à la casse

Exemple :

```
public static void main(String[] args) {

    boolean isOk = true;
    isOk = false;

}
```

© Boris SAUVAGE - 2023

Les littéraux de caractères

- Expriment des caractères Unicode (codé en 16 bits, soit 2 octets)
- Correspondant à un type entier
- Le caractère se place entre des guillemets simple
- https://www.asciitable.com/

```
public static void main(String[] args) {

    char myChar = 66;
    char myCharBis = 'B';

}
```

```
B
B
```

Les littéraux de caractères

Exemple :

```
System.out.printf("%1$-8s %2$-10s %3$s\n", "Decimal", "ASCII", "Hex");
  int min = 0;
  int max = 255;
  for (int i = min; i <= max; i++) {
    char ch = (char) i;
    String display = "";
    if (Character.isWhitespace(ch)) {
      switch (ch) {
        case '\r' -> display = "\\r";
        case '\t' -> display = "\\t";
        case '\n' -> display = "\\n";
        case '\f' -> display = "\\f";
        case ' ' -> display = "space";
        default -> display = "whitespace";
      }
    } else if (Character.isISOControl(ch)) {
      display = "control";
    } else {
      display = Character.toString(ch);
    }
    System.out.printf("%1$-8d %2$-10s %3$s\n", i, display, Integer.toHexString(i));
  }
```

Decimal	ASCII	Hex
..		
65	A	41
66	B	42
67	C	43
68	D	44
69	E	45
70	F	46
...		

Les littéraux entiers

- Expriment des valeurs entières
- Il existe plusieurs **4 types** pour définir un entier

	byte	**short**	**int**	**long**
MIN_VALUE	- 128	-32 768	-2 147 483 648	-9 223 372 036 854 775 808L
MAX_VALUE	127	32 767	2 147 483 647	9 223 372 036 854 775 807L
Taille (octets)	1	2	4	8
Taille (bits)	8	16	32	64

```
byte bb = 127;
short s = 32767;
int i = 2147483647;
long l = 9223372036854775807L;      // suffixed by L : otherwise, its type is int
```

© Boris SAUVAGE - 2023

Les littéraux entiers

- Expriment des valeurs entières
- Il existe plusieurs **4 types** pour définir un entier

	byte	**short**	**int**	**long**
MIN_VALUE	- 128	-32 768	-2 147 483 648	-9 223 372 036 854 775 808L
MAX_VALUE	127	32 767	2 147 483 647	9 223 372 036 854 775 807L
Taille (octets)	1	2	4	8
Taille (bits)	8	16	32	64

```
byte bb = 127;
short s = 32767;
int i = 2147483647;
long l = 9223372036854775807L;      // suffixed by L : otherwise, its type is int
```

© Boris SAUVAGE - 2023

Les littéraux à virgule flottante

- Expriment des valeurs entières
- Il existe plusieurs **2 types** pour définir un type flottant

	float	**double**
MIN_VALUE	1.4 **E-45**	4.9 **E**-324
MAX_VALUE	3.4028235 **E38**	1.7976931348623157 **E308**
Taille (octets)	4	8
Taille (bits)	32	64

```
float ff = 3.4028235E38f;        // suffixed by f : otherwise, its type is double
double d = 1.7976931348623157E308;
```

Les littéraux de chaîne de caractères

- représente des valeurs de chaîne dans le code source Java
- La chaîne de caractère se place entre des guillemets doubles
- Des caractères d'échappement sont disponibles
  - **\n** pour un saut de ligne ou bien encore

Exemples :

```
public static void main(String[] args) {

  String welcomeMsg = "Welcome to : \nBoris";
   System.out.print(welcomeMsg);

}
```

```
Welcome to :
Boris
```

Les littéraux nul

- Représente une valeur unique nul, de type **null**
- Utilisé avec des types objets (pointant vers une zone mémoire vide)
- Les types objets sont représentés par des classes Java

Exemples :

```
public static void main(String[] args) {
    MyClass myVar = null;
    String myVar2 = null;
}
```

Les littéraux Hexa, décimaux.. 1/2

Le système décimal (Base 10) nous permet de compter dans notre quotidien.

Il existe d'autres systèmes :

- Les littéraux hexadécimaux (base-16)
  - Basé sur 16 symboles, 0-9 et A-F pour représenter 10-16
- Les littéraux décimaux (base-10)
  - Basé sur 10 symboles de 0 à 9
- Les littéraux octaux (base-8)
  - Basé sur 8 symboles, de 0 à 7
- Les littéraux binaires (base-2), introduite en Java 7
  - Basé sur 2 symboles, de 0 à 1

© Boris SAUVAGE - 2023

Les littéraux Hexa, décimaux.. 2/2

Exemple des littéraux décimaux, hexadécimaux, octaux et binaires

```
public static void main(String[] args) {

    int dec = 2025;                    // decimal literal : no prefix
    int hex = 0x7E9;                   // hexadecimal literal : prefix 0x
    int oct = 03751;                   //    octal literal : prefix 0
    int bin = 0b11111101001;           //  binary literal : prefix 0b

}
```

© Boris SAUVAGE - 2023

Les traits de soulignement

- Depuis Java 7, les traits de soulignement sont utilisable pour améliorer la lisibilité
- Permettent de séparer des groupes de chiffres dans un littéral

```
public static void main(String[] args) {
   int i1 = 123456;
   int i2 = 123_456;
   System.out.println("i1 et i2 ont la même valeur");

   byte b = 1_2_3;
   short s = 2_016;
   int i = 999_99_9999;
   long l = 1234_5678_9012_3456L;
   float f =  3.14_15F;
   double d =  3.14_15_92_65_35;
}
```

© Boris SAUVAGE - 2023

Exercice

- Ecrire un programme qui calcule le TTC d'une paire de chaussures à **56 euros HT**

REMARQUE : Une multiplication s'écrit :

```
int valueA = 100;
int valueB = 200;
int result = valueA * valueB;
```

Développement JAVA

TYPES PRIMITIFS ET OBJETS

© Boris SAUVAGE - 2023

Les différents types en Java

En Java, nous distinguons 2 catégories de types pour les variables :

- Les types primitifs
  - Contiennent une seule valeur

- Les types objets
  - Variables complexes car ils peuvent contenir plusieurs valeurs
  - Sont représenté par des classes Java

© Boris SAUVAGE - 2023

Les types primitifs

- Java est un langage orienté objet mais propose des *types natifs*
- Ils se nomment en minuscules
- Les types primitifs sont défini par les mots clés suivants :

boolean	byte	char	double	float
int	long	short		

© Boris SAUVAGE - 2023

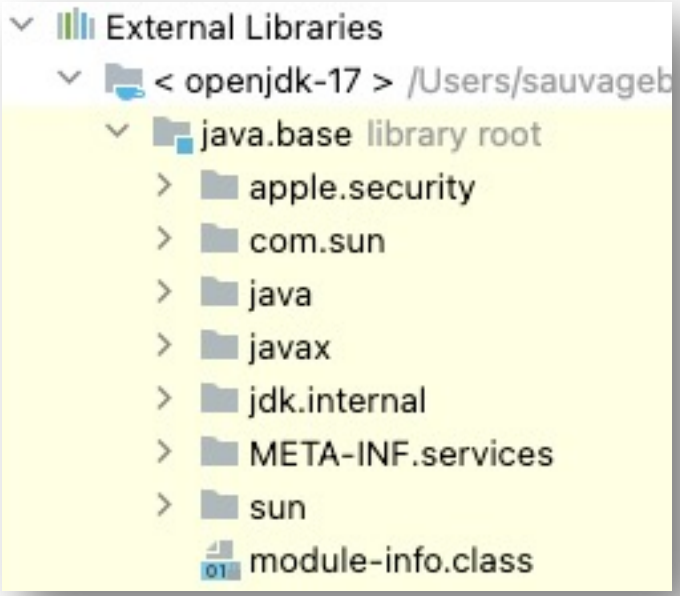
Les types objets

- Java propose la création et l'utilisation de nombreuses classes
- Les classes peuvent être instanciées et utilisées dans le projet
- Il existe de nombreuses classes de base

<u>Exemples :</u>

- String
- Object
- Arrays

```
public static void main(String[] args) {
    String msg = "Hi Boris !";
    System.out.println(msg);
}
```



Les types enveloppe 1/3

- Java fournit des classes qui enveloppent la valeur des types primitifs (**Wrappers**)

Type	Classe associée
boolean	java.lang.Boolean
char	java.lang.Character
int	java.lang.Integer
byte	java.lang.Byte
short	java.lang.Short
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

© Boris SAUVAGE - 2023

Les types enveloppe 2/3

Java fournit des classes qui enveloppent la valeur des types primitifs

- Depuis les types enveloppes, les valeurs sont récupérables avec **xxxValue()**

```
Integer i = Integer.valueOf(2);
int x = 1 + i.intValue();
```

- Il est possible de convertir des String en type primitif avec **parseXXX()**

```
boolean b = Boolean.parseBoolean("true");
byte by = Byte.parseByte("1");
short s = Short.parseShort("1");
int i = Integer.parseInt("1");
long l = Long.parseLong("1");
float f = Float.parseFloat("1");
double d = Double.parseDouble("1");
```

© Boris SAUVAGE - 2023

Les types enveloppe 3/3

- Une classe enveloppe référence un objet (pouvant donc être **null**)

```
Integer i = null;
int x = 0;
```

```
boolean b = Boolean.parseBoolean("true");
byte by = Byte.parseByte("1");
short s = Short.parseShort("1");
int i = Integer.parseInt("1");
long l = Long.parseLong("1");
float f = Float.parseFloat("1");
double d = Double.parseDouble("1");
```

<u>REMARQUE :</u> La plupart des types **Wrapper** héritent de la classe java.lang.Number

© Boris SAUVAGE - 2023

Types primitifs et objets

Autoboxing

- Le passage d'un type primitif à un type enveloppe est le **boxing**
  - L'opération inverse s'appelle **l'unboxing**

- Grâce au compilateur, l'écriture des conversions est automatiquement allégées

```
Integer i = 1;
```

=

```
Integer i = Integer.valueOf(1);
```

```
Integer i = 1;
int j = i;
```

=

```
Integer i = Integer.valueOf(1);     // boxing
int j = i.intValue();               //  unboxing
```

<u>REMARQUE :</u> L'**unboxing** d'une variable **null** aboutira à une NullPointerException

```
Integer i = null;
int j = i;                  //  unboxing : Erreur NullPointerException
```

Conventions de nommages 1/4

- En java, il existe une convention de nommage pour nommer les identifiants
  - Comme pour la convention de formatage de code, elle n'impacte pas la compilation
- Garantie une bonne lisibilité et compréhension de son code
  - Les développeurs Java sont particulièrement attaché au respect des conventions
- Les identifiants affectés par cette convention sont les suivants :
  - Packages
  - Classes et Interfaces
  - Méthodes
  - Variables
  - Constantes
  - Annotations

© Boris SAUVAGE - 2023

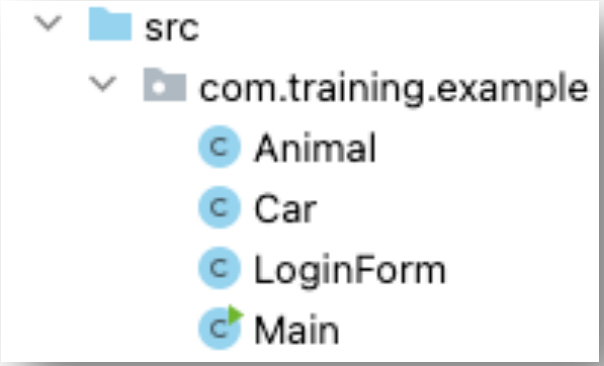
Conventions de nommages 2/4

Les packages

- S'écrit toujours en minuscule
- Le _ est tolérée pour représenter une séparation

Les classes et interfaces

- Les noms ne doivent pas être des verbes
- La 1ère lettre de chaque mot doit être en majuscule
  - **(notation Pascal Case)**



Conventions de nommages 3/4

Les variables

```
int myValue;
```

- Les variables jetables utilisent des lettre d'usage (i, j, x)
- Nommage explicite, sans abréviation, sans commencer par _ ou **$**
- La 1ère lettre est en minuscule, puis les mots sont séparés par une majuscule
  - **(notation Camel Case)**

Les constantes

```
int NB_MAX_USERS = 50;
```

- S'écrit intégralement en lettres majuscules et les mots sont séparés par _

© Boris SAUVAGE - 2023

Conventions de nommages 4/4

Les annotations

```
@Autowired    @EJB
```

- Les sigles intégralement en majuscules sont tolérés
- La 1ère lettre est en majuscule, puis les mots sont séparés par une majuscule
  - **(notation Pascal Case)**

Les méthodes

- La 1ère lettre est en minuscule, puis les mots sont séparés par une majuscule
  - **(notation Camel Case)**
- doivent être des verbes

```
public int getMyValue() {
   return myValue;
}
```

Mots clés Java 1/3

- Les mots-clés pour les **objets :**

abstract	class	enum	extends	implements
import	interface	native	package	super
this				

- Les mots-clés pour les **types :**

boolean	byte	char	double	float
int	long	short	void	

- Les mots-clés pour les **modificateurs :**

private	protected	public		

© Boris SAUVAGE - 2023

Mots clés Java 2/3

- Les mots-clés pour les **états :**

const	false	final	new	null
static	transient	true	volative	

- Les mots-clés pour les **boucles :**

continue	do	for	while	

- Les mots-clés pour les **branchements :**

assert	break	case	default	else
if	instanceof	return	switch	synchronised

- Les mots-clés pour les **exceptions :**

catch	finally	throw	throws	try

© Boris SAUVAGE - 2023

Mots clés Java 3/3

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	if	implements	import
instanceof	int	interface	long	native
new	null	package	private	protected
public	return	short	static	super
switch	synchronized	this	throw	throws
transient	true	try	void	volatile
while				

© Boris SAUVAGE - 2023

Exercice

- Ecrire un programme qui demande à l'utilisateur de saisir le prix HT d'un objet et qui donne sa valeur TTC

REMARQUE : Une multiplication s'écrit :

```
int valueA = 100;
int valueB = 200;
int result = valueA * valueB;
```

Développement JAVA

LES PACKAGES

Les packages

Introduction

En programmation, il est courant de manipuler des classes portant le même nom

Sans package, des problèmes pourraient survenir :

- Le CLASSPATH se retrouvera avec deux classes différentes portant le même nom
- la JVM chargera la 1ère classe qu'elle peut trouver et ignorera la seconde

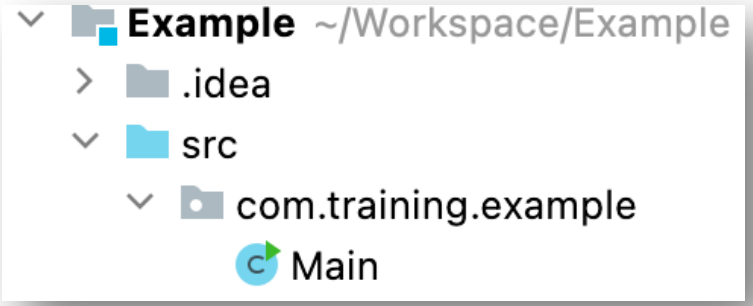
- La création de **packages** (espaces de noms) permet de rendre cela possible
  - Une classe peut appartenir à un seul package
  - Un package est représenté par des répertoires

```
package com.training.example;

public class User {
}
```

Projet & package

- Une classe sans package **(au début du fichier)** appartient au package par défaut
  - C'est un package sans nom

- Une bonne pratique est de créer un premier package parent



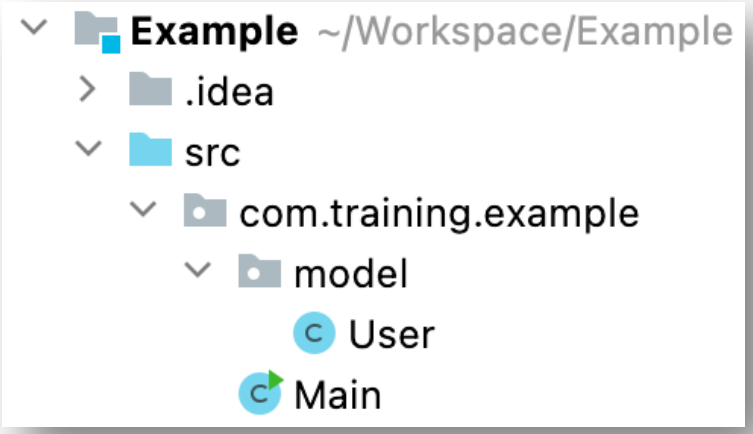

```
package com.training.example;

public class Main {

    public static void main(String[] args) {
        System.out.println("Example");
    }
}
```

- Exemple : https://play.google.com/store/apps/details?id=com.facebook.katana

Les packages

Projet & sous-package

- Par la suite, il est possible de créer des sous-packages dans le package parent




```
package com.training.example.model;

public class User {
                ...
}
```

Package et nommage

- Pour nommer le package principal, nous utilisons un nom de domaine inversé

- Une classe est désignée par son nom complet (chemin de packages + classe)

```
public class Main {

    public static void main(String[] args) {
        java.lang.String msg = "Example";
        System.out.println(msg);
  }

}
```

Import de noms (classe)

- Le mot clé **import** permet d'importer l'espace de nom d'une classe pour l'utiliser :

```
package com.training.example;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {
        int[] arrays = {1, 2, 3};
        System.out.println(Arrays.toString(arrays));
    }
}
```

- L'import de tous les noms de méthodes et attributs statiques :

```
Import java.util.*;
```

© Boris SAUVAGE - 2023

Import de noms (statique)

- **import static** permet d'importer un nom de méthode ou un d'attribut statique

```
package com.training.example;

import java.lang.System.out;

public class Main {

    public static void main(String[] args) {
        out.println("Example");
    }
}
```

- L'import de tous les noms de méthodes et attributs statiques :

```
Import static java.lang.Math.*;
```

© Boris SAUVAGE - 2023

Import de noms (classe interne)

- Le mot clé **import** permet d'importer un nom de classe interne **(inner class)**

```
package com.training.example;

import java.util.Map.Entry;
..
public class Main {

   public static void main(String[] args) {
      Map<String, Integer> m = new LinkedHashMap<>();
      m.put("Appart-5472", 850);
      m.put("Appart-2495", 1020);

      Set<Entry<String, Integer>> s = m.entrySet();
      s.forEach(set -> System.out.println(set.getKey() + ": " + set.getValue()));
   }
}
```

Gestion des imports sur IDE 1/3

- Le caractères * permet d'importer tous les noms d'un package
  - Son utilisation n'est pas conseillée

- Intellij IDEA permet de simplifier la gestion des importations :
  - La saisie d'une classe qui ne fait pas partie de l'espace de nom ajoute **automatiquement** un import :

```
public static void main(String[] args) {
    Random
    Random java.util
    RandomAccess java.util
    RandomAccessFile java.io
    SplittableRandom java.util
    SecureRandomSpi java.security
    ThreadLocalRandom java.util.concurrent
    SecureRandom java.security
    SecureRandomParameters java.security
Press ^Space to see non-imported classes Next Tip
```

```
import java.util.Random;

public class Main {

    public static void main(String[] args) {
        Random r = new Random();
```

Gestion des imports sur IDE 2/3

- **L'importation manuelle** est possible avec le raccourci **Alt+Enter** ou **Option+Return**

```
import java.util.Random;

public class Main {

    public static void main(String[] args) {
        Random r = new Random();
```

Gestion des imports sur IDE 3/3

- **Les actions contextuelles** proposent de résoudre les problèmes détectés
  - Proposition de correctifs rapides pour des erreurs mises en évidence

- Utilisable de plusieurs manières différentes :
  - en survolant le problème
  - en cliquant sur l'ampoule
  - Avec le raccourci **Alt+Enter**

Alt+Enter

```
Random
```

Cannot resolve symbol 'Random'

Import class ⌥⇧⏎ More actions... ⌥⏎

No documentation found.

```
import java.util.Random;

public class Main {

    public static void main(String[] args) {
        Random r = new Random();
```

Exercice

- Reprenez les exercices précédents :
  - Organisez-les avec des packages
  - Respectez les conventions de nommages des classes et des variables

Développement JAVA

LES OPÉRATEURS

Opérateurs d'affectation 1/2

- L'affectation est réalisée grâce à l'opérateur **=**
- Cet opérateur copie la valeur du paramètre de droite dans le paramètre de gauche
- Java opère donc par copie

```
public class Main {

  public static void main(String[] args) {
    int variableA = 1;
    System.out.println(variableA);

    int variableB = variableA;          // variableB reçoit la copie de la valeur de variableA
    System.out.println(variableB);

    variableA = 10;
    System.out.println(variableA);      // variableA vaut 10 mais variableB vaut toujours 1
    }
}
```

```
1
1
10
```

Opérateurs d'affectation 2/2

- L'opérateur d'affectation est aussi utilisé pour l'initialisation
- Exemple d'initialisation :

```
public static void main(String[] args) {
    // Initialisation
    int variableA = 1;
}
```

- Exemple d'affectation :

```
public static void main(String[] args) {
    int variableA;
    // Affectation
    variableA = 1;
}
```

© Boris SAUVAGE - 2023

Opérateurs arithmétiques 1/2

- Les opérateurs arithmétiques à deux opérandes :

*	Multiplication
/	Division
%	(Modulo) Reste de la division
+	Addition
-	Soustraction

- <u>Exemple :</u>

```
int i = 30 * 3;

int j = 30 % 2;

int k = 30 / 5;
```

```
90
0
6
```

Opérateurs arithmétiques 2/2

- Les opérateurs arithmétiques unaires :

variable++	Post incrémentation
++variable	Pré-incrémentation
variable--	Post décrémentation
--variable	Pré-décrémentation

  - Les opérateurs préfixés se font avant l'affectation
  - Les opérateurs postfixés se font après l'affectation

- <u>Exemple :</u>

```
int i = 30;
int j = i++;
System.out.println(j);

int k = 30;
int l = ++k;
System.out.println(l);
```

```
30
31
```

Opérateur de concaténation

- Les chaînes de caractères peuvent être concaténées avec l'opérateur **+**

```
String msg1 = "Hi ";
String msg2 = "Boris ";
String msg3 = "!";
System.out.println(msg1 + msg2 + msg3);
```

```
Hi Boris !
```

- Il est aussi possible de concaténer un objet de type string avec un autre type

```
Integer age = 30;
String msg = "I'm ";
System.out.println(msg + age);
```

```
I'm 30
```

- Lors de l'affichage, le compilateur utilise les méthodes *toString()* des objets

```
System.out.println(msg.toString() + age.toString());
```

Opérateurs relationnels 1/2

- Les opérateurs relationnels produisent un résultat booléen

<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent

  – Ils s'utilisent pour les nombres ou les caractères **(char)**

```
int age = 30;
int ageNeighbour = 48;
boolean isOlder = age < ageNeighbour;
System.out.println(isOlder);
```

© Boris SAUVAGE - 2023

Opérateurs relationnels 2/2

- Les opérateurs **==** et **!=** comparent les valeurs contenus dans 2 variables

<u>Exemples :</u>

```
int age = 30;
int ageNeighbour = 30;
boolean isSame = (age == ageNeighbour);
System.out.println(isSame);
```

```
true
```

REMARQUE : Avec des objets, ils comparent la références mémoire contenues

```
Integer age = 30;
Integer ageNeighbour = 30;
boolean isSame = (age == ageNeighbour);
System.out.println(isSame);
```

```
false
```

Opérateurs logiques 1/4

- Prennent des booléens comme opérandes et produisent un résultat booléen

!	Négation
&&	ET logique
||	OU logique

- <u>Exemple :</u>

```
boolean aaa = true;
System.out.println(aaa);

boolean bbb = !aaa;
System.out.println(bbb);

boolean ccc = aaa && bbb;
System.out.println(ccc);

boolean ddd = aaa || bbb;
System.out.println(ddd);
```

```
true
false
false
true
```

Opérateurs logiques 2/4

L'algèbre de Boole (calcul booléen) permet d'évaluer des expressions à 2 valeurs

- **ET logique && :**
  - renvoie vrai si les 2 expressions évaluées renvoient vrai

```
System.out.println(true && true);
System.out.println(false && false);

System.out.println(true && false);
System.out.println(false && true);
```

```
true
false
false
false
```

- **Evaluation ET Logique :** si la 1ère expression est fausse, Java n'évalue pas la n-ième

Opérateurs logiques 3/4

L'algèbre de Boole (calcul booléen) permet d'évaluer des expressions à 2 valeurs

- **OU logique || :**
  - renvoie vrai si 1 expression évaluée renvoie vrai

```
System.out.println(true || true);
System.out.println(false || false);

System.out.println(true || false);
System.out.println(false || true);
```

```
true
false
true
true
```

- **Evaluation OU Logique :** si la 1ère expression est vraie, Java n'évalue pas la n-ième

Opérateurs logiques 4/4

L'algèbre de Boole (calcul booléen) permet d'évaluer des expressions à 2 valeurs

- **NON logique ! :**
  - renvoie l'inverse de l'évaluation initiale

```
System.out.println(!true);
System.out.println(!false);
```

```
false
true
```

Opérateurs et assignations

- Il existe une forme compacte permettant une assignation avec une opération

*=	Multiplication
/=	Division
%=	(Modulo) Reste de la division
+=	Addition
-=	Soustraction

- <u>Exemple :</u>

```
int i, j, k = 30;
i *= 3;
j %= 2;
k /= 5;
```

```
90
0
6
```

Les opérateurs

L'opérateur .

- Permet d'accéder aux attributs et méthodes d'une classe ou d'un objet

Exemple :

```
String s = "Hello, my name is Boris";
int length = s.length();
System.out.println("sentence contains " + length  + " characters");
```

```
sentence contains 23 characters
```

```
int length = "Hello, my name is Boris".length();
System.out.println("sentence contains " + length  + " characters");
```

```
sentence contains 23 characters
```

Les opérateurs

L'opérateur ,

- Permet de séparer les paramètres dans la définition et l'appel des méthodes

Exemple :

```
int a = 0, b = 1, c = 2;
System.out.println(a);
System.out.println(b);
System.out.println(c);
```

```
0
1
2
```

```
int result = addition(18, 24);

System.out.println(result);
```

```
42
```

Exercice

- Vous avez ouvert un PEL (Plan Epargne Logement) en juillet 2000 et vous n’avez jamais perçu les intérêts de celui-ci.
- Avec 8502 euros sur le PEL, calculez les intérêts **(3.27%)** que la banque vous doit et affichez les
- Mauvaise nouvelle, des prélèvements sociaux **(17,20%)** sont dus sur ces intérêts…Affichez les

© Boris SAUVAGE - 2023

Développement JAVA

STRUCTURES DE CONTRÔLE

Les structures de contrôle

Introduction

Comme la plupart des langages impératifs, Java propose des structures de contrôle

- If-else
- While / Do-while
- For / Foreach
- Switch
- Break / Libellé / Continue / Return

If-else

- Permet d'exécuter un bloc d'instructions selon l'évaluation d'une condition

Exemple :

```
int age = 30;
if (age > 18) {
   System.out.println("Adult person");
}
```

```
if (age > 18) {
   System.out.println("Adult person");
} else if (age > 12 && age < 18) {
   System.out.println(" Adolescence");
} else {
   System.out.println("Juvenil person");
}
```

```
if (age > 18) {
   System.out.println("Adult person");
} else {
   System.out.println("Juvenil person");
}
```

```
Adult person
```

While / Do-While

- Permet de définir un bloc d'instructions à répéter tant qu'une condition est vraie

Exemple : while

```
int nb = 1;
while (nb < 6){
   System.out.print(nb++ + " ");
}
```

```
1 2 3 4 5 6
```

Exemple : do-while

```
int nb = 1;
do{
   System.out.print(nb++ + " ");
} while (nb < 6);
```

```
1 2 3 4 5 6
```

Boucle For

- Permet de réaliser une itération, tant qu'une condition est vraie

Exemple :

```
for (int index = 1; index <= 6; index++) {
    System.out.print(index + " ");
}
```

```
1 2 3 4 5 6
```

- Peut-être composé d'une initialisation, d'une condition et d'une incrémentation

```
int index = 1 ;
for ( ; index <= 6; index++) {
    System.out.print(index + " ");
}
```

```
int index = 1 ;
for ( ; index <= 6; ) {
    System.out.print(index++ + " ");
}
```

```
for ( ;; ) {
}
```

Boucle For-each

- Permet d'utiliser la boucle For avec une syntaxe succinte

Exemple :

```
short[] arrayOfShort = {1, 2, 3, 4, 5, 6};

for (int value : arrayOfShort) {
   System.out.print(value + " ");
}
```

```
1 2 3 4 5 6
```

- La variable désignant la collection doit être un tableau ou un type **Iterable**

Le Switch

- Permet d'effectuer une sélection parmi plusieurs valeurs
- Selon la version de Java, la syntaxe est différente

- **Java 1 à 7 :** Uniquement les entiers sont utilisables
- **Java 8 (LTS) :** Les Strings et les énumérations sont désormais utilisables
- **Java 12 :** Le switch peut retourner une valeur avec return et break
- **Java 13 :** Le mot-clé break dans le switch est remplacé par yield
- **Java 17 (LTS) :** L'utilisation d'objets et la vérifications de type est possible
- **…**

© Boris SAUVAGE - 2023

Java 1-7 : Switch historique

- Permet d'effectuer un switch avec plusieurs instructions

Exemple :

```
int value = 5;
switch (value) {
   case 1:
      System.out.println("One");
      break;
   case 5:
      System.out.println("five");
      break;
   default:
      System.out.println("Unknown");
}
```

```
five
```

© Boris SAUVAGE - 2023

Java 8 (LTS) : Switch

- Permet d'effectuer un switch avec plusieurs instructions

Exemple :

```
String day = "Tuesday";
switch (day) {
   case "Monday":
   case "Tuesday":
   case "Wednesday":
   case "Thursday":
   case "Friday":
      System.out.println("Week day");
      break;
   case "Saturday":
      System.out.println("Weekend");
      break;
   default:
      System.out.println("Unknown");
}
```

```
Weekend
Weekend
```

```
DAYS days = DAYS.TUESDAY;
switch (days) {
   case MONDAY:
   case TUESDAY:
   case WEDNESDAY:
   case THURSDAY:
   case FRIDAY:
      System.out.println("Week day");
      break;
   case SATURDAY:
   case SUNDAY:
      System.out.println("Weekend");
      break;
   default:
      System.out.println("Unknown");
}
```

Java 13 : Switch, yield, return

- Permet d'effectuer un switch avec plusieurs instructions

Exemple :

```
int position = 4;
int score = 50;
String result = switch (position) {
   case 1 -> {
      score = score + 10;
      yield "THE WINNER with " + score + " points ";
   }
   case 2, 3 -> {
      score = score + 5;
      yield "THE WINNER in 2nd and 3rd place with " + score + " points ";
   }
   default -> "Hum...Looser !";
};
System.out.println(result);
```

Hum…Looser !

Java 17 : Switch

- Permet d'effectuer un switch avec plusieurs instructions

Exemple :

```
String day = "Saturday";
switch (day) {
    case "Monday", "Tuesday", "Wednesday", "Thurday" -> System.out.println("Week");
    case "Saturday", "Sunday" -> System.out.println("Weekend");
    default -> System.out.println("Unknown day");
}
```

```
Weekend
```

Java 17 : Pattern Matching

- Avec la version de Java 17, il est possible de vérifier un type :

```
public static void main(String[] args) {
   String o = "Example value";
   System.out.println(formatterPatternSwitch(o));
}

static String formatterPatternSwitch(Object o) {
   return switch (o) {
      case Integer i -> String.format("int %d", i);
      case Long l    -> String.format("long %d", l);
      case Double d  -> String.format("double %f", d);
      case String s  -> String.format("String %s", s);
      default        -> o.toString();
   };
}
```

```
String Example value
```

Java 17 : Guarded Patterns

- Avec la version de Java 17, il est possible de vérifier une condition :

```
public static void main(String[] args) {
   User user = new User("Boris", "S");
   System.out.println(formatterPatternSwitch(user));
}

static String formatterPatternSwitch(Object o) {
   return switch (o) {
      case Integer i -> "It is an integer";
      case String s -> "It is a string";
      case User u && u.getFirstname().equals("Boris") -> "It's Boris";
      default -> "It is none of the known data types";
   };
}
```

```
It’s Boris
```

Java 17 : Null cases

- Avec la version de Java 17, il est possible de vérifier un cas null :

```
public static void main(String[] args) {
   User user = null;
   System.out.println(formatterPatternSwitch(user));
}

static String formatterPatternSwitch(Object o) {
   return switch (o) {
      case null  -> "It’s Null";
      case User u && u.getFirstname().equals("Boris") -> "It's Boris";
      default -> "It is none of the known data types";
   };
}
```

```
It’s Null
```

Break & continue

- il est possible d'arrêter une itération sans exécuter le reste avec **break**

```
int sum = 0;

for (int i = 0; i < 10; i++) {
        sum += i * 30;
        if (sum > 500) {
            break;
        }
}
System.out.println(sum);
```

```
630
```

```
int[] values = {1, 2, 3, 4, 5};
int sum = 0;

  for (int i = 0; i < values.length; i++) {
     if (values[i] % 2 == 0) {
        continue;
     }
     sum += values[i];
  }
  System.out.println(sum);
}
```

```
9
```

Break & Libellé

- il est possible de définir un libellé avant une expression **for** ou **while**
  - Permet d'arrêter une itération de niveau supérieur

```
int[][] values = {{1, 2}, {3, 4}, {5, 6}};
search:
for (int i = 0; i < values.length; i++) {
   for (int j = 0; j < values[i].length; j++) {
      if (values[i][j] == 4) {
         System.out.println("Le chiffre 4 a été trouvé à l'index [" + i + "][" + j + "]");
         break search;
      } else if (values[i][j] == 6) {
         System.out.println("Le chiffre 6 a été trouvé à l'index [" + i + "][" + j + "]");
         break search;
      }
   }
}
```

```
Le chiffre 4 a été trouvé à l'index [1][1]
```

Return

- Permet d'arrêter le traitement d'une méthode, et de retourner une valeur si besoin

```
public static void main(String[] args) {
   example();
}

static void example() {
   for (int i = 1; i < 10; i++) {
      if (i == 6) {
         return;
      }
      System.out.println(i);
   }
}
```

```
1
2
3
4
5
```

sans retour

```
public static void main(String[] args) {
   String result = example();
   System.out.println(result);
}

static String example() {
   for (int i = 1; i < 10; i++) {
      if (i == 6) {
         return "i = 6";
      }
   }
   return null;
}
```

```
i = 6
```

avec un retour de valeur

Exercice

- Demander à l’utilisateur de saisir un mot ou une phrase
- Indiquez ensuite si la saisie est un Palindrome ou non

REMARQUE :

- La classe **Scanner** et la méthode **charAt()** seront vos meilleurs alliés

Développement JAVA

LES TABLEAUX

Les tableaux

- Représentent des collections de valeurs ou d'objets

- Déclaration d'une variable de type tableau :

```
int[] myArray;
```

```
int myArray[];

int []myArray;
```

- Les tableaux sont eux-mêmes des objets
  - Le mot-clé new est utilisable
  - Une variable de type tableau peut avoir la valeur **null**

© Boris SAUVAGE - 2023

Déclaration avec initialisation

- Une variable de type tableau est initialisable avec une liste de valeurs

```
int[] integerArray = {1, 2, 3, 4, 5};
String[] stringArray = {"Hello", "World", "!"};
```

- La taille d'un tableau est fixe

- L'écriture avec le mot-clé new est possible

```
int[] integerArray = new int[] {1, 2, 3, 4, 5};
String[] stringArray = new String[] {"Hello", "World", "!"};
```

© Boris SAUVAGE - 2023

Déclaration sans initialisation

- Une variable de type tableau peut-être utilisable sans valeurs
  - Comme la taille d'un tableau est fixe, il faut la préciser

```
boolean[] booleans = new boolean[3];

char[] chars = new char[3];

byte[] bytes = new byte[3];
short[] shorts = new short[3];
int[] ints = new int[3];
long[] longs = new long[3];

float[] floats = new float[3];
double[] doubles = new double[3];

Object[] objects = new Object[3];
```

false	false	false
`\0`	`\0`	`\0`
0	0	0
0.0	0.0	0.0
null	null	null

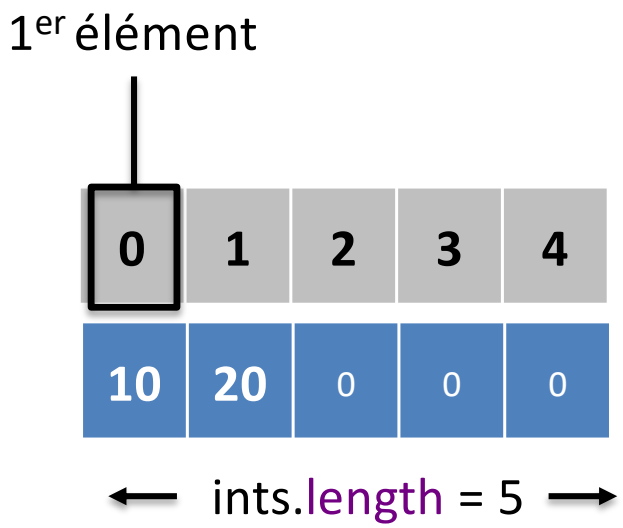
- Dans ce cas, les éléments sont tous initialisés avec une valeur par défaut

Les tableaux

Manipulation

- Exemple avec une déclaration, puis une initialisation

```
int[] ints = new int[5];
ints[0] = 10; // initialisation du 1er élément
ints[1] = 20; // initialisation du 2nd élément
```




```
// déclaration + initialisation des éléments
int[] ints = {10, 20, 30, 40, 50};
```

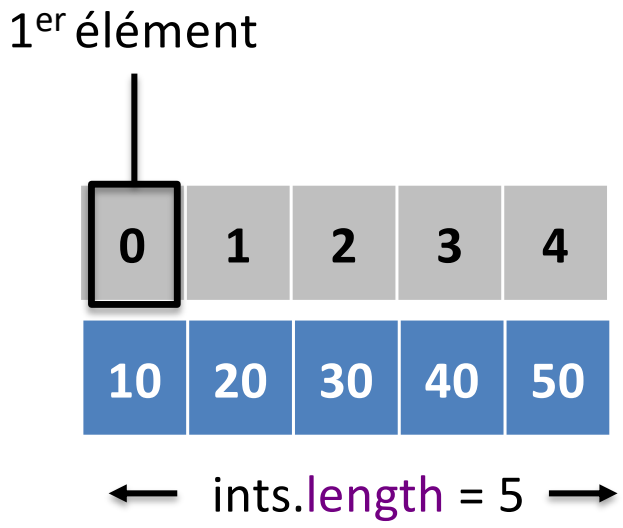
10	20	30	40	50

Accès aux éléments

- L'accès aux éléments se fait en donnant l'indice d'un élément entre crochets
  - Le premier élément d'un tableau a l'indice **0**
  - Le dernier élément d'un tableau a donc comme indice la taille du tableau moins un

```
int[] ints = {10, 20, 30, 40, 50};

for (int i = 0; i < ints.length; i++) {
    System.out.println(ints[i]);
}
```

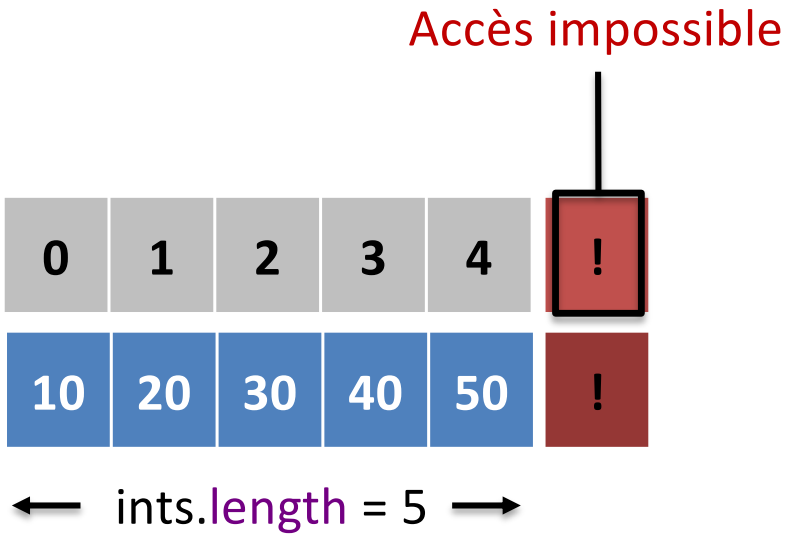
```
for (int i : ints) {
    System.out.println(i);
}
```



Accès aux mauvais éléments

- L'accès à un indice de tableau trop grand ou négatif produira une erreur :
  - java.lang.ArrayIndexOutOfBoundsException

```
int[] ints = {10, 20, 30, 40, 50};

System.out.println(ints[5]);
```




```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
at com.training.app.Example.main(Example.java:7)
```

Développement JAVA

MULTI-DIMENSIONNEL

Les tableaux multidimensionnels

- Représentent un tableau de tableaux
  - Chaque élément d'un tableau est lui-même un tableau

- Déclaration d'un tableau à 2 dimensions :

```
int[][] ints = new int[5][10];
```

  - Ce tableau peut contenir 50 éléments

```
int myArray[][];

int [][]myArray;
```

- Déclaration d'un tableau à 3 dimensions :

```
int[][][] ints = new int[5][10][10];
```

  - Ce tableau peut contenir 500 éléments

```
int myArray[][][];

int [][][]myArray;
```

© Boris SAUVAGE - 2023

Déclaration avec initialisation 1/2

- Il est possible d'initialiser un tableau à plusieurs dimensions

- Déclaration et initialisation d'un tableau à **2 dimensions :**

```
int[][] ints = {  {1 , 2}  ,  {3 , 4}  ,  {5 , 6}  };
```

  – Ce tableau contient **6 éléments**

```
System.out.println(  ints[0][1]  );
```

```
2
```

- L'écriture suivante est possible :

```
int[][] ints = new int[2][];

ints[0] = new int[10];
ints[1] = new int[5];
```

Déclaration avec initialisation 2/2

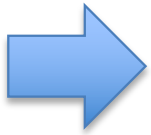
- Il est possible d'initialiser un tableau à plusieurs dimensions

- Déclaration et initialisation d'un tableau à **3 dimensions :**

```
int[][][] ints = {  { {1 , 2}  ,  {3 , 4} } , { {5 , 6}  ,  {7 , 8} } };
```

  – Ce tableau contient **8 éléments**

```
System.out.println(  ints[0][1][0]  );
```



```
3
```

- L'écriture suivante est possible :

```
int[][] ints = new int[2][];

ints[0] = new int[10];
ints[1] = new int[5];
```

Vue sur un tableau à 2 dimensions

```
int[][] ints = {  {1, 2, 3} ,   {4, 5, 6, 9} ,  {7}  };
```

	Colonne 1	Colonne 2	Colonne 3	Colonne 4
Ligne 1	1 [0] [0]	2 [0] [1]	3 [0] [2]	
Ligne 2	4 [1] [0]	5 [1] [1]	6 [1] [2]	9 [1] [3]
Ligne 3	7 [2] [0]			

```
System.out.println("Longueur de la ligne 1: " + ints[0].length);
System.out.println("Longueur de la ligne 2: " + ints[1].length);
System.out.println("Longueur de la ligne 3: " + ints[2].length);
```

```
...3
...4
...1
```

Conversion en chaîne de caractères

- L'affichage d'un tableau sur la sortie standard est possible avec la classe **Arrays** :

```
int[] a = {1, 2, 3, 4, 5};
System.out.println( java.util.Arrays.toString( a ) );

int[][] b = {{1, 2}, {3, 4}, {5, 6}};
System.out.println( java.util.Arrays.deepToString( b ) );
```

```
[1, 2, 3, 4, 5]
[[1, 2], [3, 4], [5, 6]]
```

- Par défaut, un tableau est converti par une String sous une forme particulière
  - Un tableau est un objet Java

```
int[] ints = {1, 2, 3, 4, 5};
System.out.println(ints);
```

```
[I@5305068a
```

Égalité de 2 tableaux 1/2

- Pour comparer 2 tableaux, il faut utiliser la classe **Arrays** (méthode **equals**)

```
int[] a = {1, 2, 3, 4, 5};
int[] b = {1, 2, 3, 4, 5};
System.out.println( java.util.Arrays.equals( a, b ) );

String[] c = {"Boris", "S"};
String[] d= {"Boris", "S"};
System.out.println( java.util.Arrays.equals( c, d ) );
```

```
true

true
```

- Pour comparer des tableaux multi-dimensionnels, il faut utiliser **deepEquals()**

```
int[][] a = {{1, 2}, {3, 4}, {5, 6}};
int[][] b = {{1, 2}, {3, 4}, {5, 6}};
System.out.println( java.util.Arrays.deepEquals( a, b ) );
```

```
true
```

© Boris SAUVAGE - 2023

Égalité de 2 tableaux 2/2

- L'utilisation de l'opérateur **==** ne doit pas être utilisées
  - Cet opérateur compare la référence en mémoire des objets
  - Il indique **true** uniquement si les deux variables référencent le même objet

```
int[] a = {1, 2, 3, 4, 5};
int[] b = {1, 2, 3, 4, 5};
System.out.println(a == a);
System.out.println(a == b);
```

```
true
false
```

- L'utilisation de la méthode **equals()** ne fonctionne pas non plus
  - Par défaut, elle a le même comportement que l'opérateur ==

```
int[] a = {1, 2, 3, 4, 5};
int[] b = {1, 2, 3, 4, 5};
System.out.println(a .equals( a ));
System.out.println(a .equals( b ));
```

```
true
false
```

Trier un tableau

- La classe java.util.Arrays propose une méthode pour trier un tableau

- Trier un tableau d'entiers est possible avec **Arrays.sort()**

```
int[] ints = {1, 5, 4, 3, 2};
java.util.Arrays.sort( ints );
System.out.println( java.util.Arrays.toString( ints ));
```

```
[1, 2, 3, 4, 5]
```

- Trier un tableau de String est également possible avec **Arrays.sort()**

```
String[] strings = {"Lille", "Paris", "Tours", "Angers", "Orléans"};
java.util.Arrays.sort(strings);
System.out.println(java.util.Arrays.toString(strings));
```

```
[Angers, Lille, Orléans, Paris, Tours]
```

Rechercher dans un tableau

- La classe java.util.Arrays propose une méthode pour rechercher dans un tableau
- Rechercher dans un tableau d'entiers avec **Arrays.binarySearch()**

```
int[] ints = {10, 20, 30, 40, 50};
System.out.println(java.util.Arrays.binarySearch(ints, 20));
System.out.println(java.util.Arrays.binarySearch(ints, 45));
```

```
1
-5
```

- Rechercher dans un tableau de Strings avec **Arrays.binarySearch()**

```
String[] strings = {"Angers", "Lille", "Orléans", "Paris", "Tours"};
System.out.println(java.util.Arrays.binarySearch(strings, "Lille"));
System.out.println(java.util.Arrays.binarySearch(strings, "Bourges"));
```

```
1
-2
```

- **binarySearch()** ne fonctionne correctement que pour un tableau trié

Copie d'un tableau

- La classe java.util.Arrays propose une méthode pour copier un tableau
  - La taille d'un tableau n'est pas modifiable

- Copier un tableau avec **Arrays.copyOf()**

```
int[] ints = {10, 20, 30, 40, 50};
ints = java.util.Arrays.copyOf( ints, ints.length + 1 );
System.out.println( java.util.Arrays.toString( ints ) );
```

[10, 20, 30, 40, 50, 0]

- Copier une partie d'un tableau avec **Arrays.copyOfRange()**

```
int[] ints = {10, 20, 30, 40, 50};
ints = java.util.Arrays.copyOfRange( ints, 3,4 );
System.out.println( java.util.Arrays.toString( ints ) );
```

[30, 40]

Conversion d'un tableau en liste

- La classe java.util.Arrays propose une méthode pour convertir un tableau en liste
  - La plupart des API Java utilisent des collections plutôt que des tableaux

- Transformer un tableau en liste avec **Arrays.asList()**

```
String[] strings = {"The", "Cat"};
java.util.List<String> stringlist = java.util.Arrays.asList( strings );
stringlist.set(1, "Dog");
System.out.println(java.util.Arrays.toString( stringlist ));
```

[The, Dog]

- Il n'est pas possible d'ajouter un élément car **Arrays.asList()** retourne une liste fixe

```
String[] strings = {"The", "Cat"};
java.util.List<String> stringlist = java.util.Arrays.asList(stringlist );
stringlist.add("!");
```

Exception in thread "main" java.lang.UnsupportedOperationException

Exercice

- Écrire un algorithme qui vous demande de saisir une note et un prénom pour chacun de vos élèves
- La saisie s’arrête dès que l’utilisateur saisit -1
- Veuillez ensuite afficher la note de chaque élèves

© Boris SAUVAGE - 2023

Développement JAVA

POO JAVA

Introduction

- Java est un langage orienté objet
  - Tout est objet, défini par des classes
- Basée sur une approche de conception et de développement de logiciels
- Représenter les éléments du monde réel par des « objets »
- De nombreux principes de POO seront abordés :
  - Classes et Objets
  - Encapsulation
  - Héritage
  - Polymorphisme
  - Exceptions

```
public class Livre {
}
```

© Boris SAUVAGE - 2023

Différentes approches

Au niveau de la conception de logiciels, il existe de nombreuses approches :

Approche procédurales : **Que doit faire mon programme ?**

- Organise la conception de logiciels autour de la logique et des procédures (fonctions)
- Utilise des procédures pour indiquer à l'ordinateur ce qu'il doit faire en une série d'étapes

Approche orientée-objet : **De quoi doit être composé mon programme ?**

- Organise la conception de logiciels autour d'objets ou de données
- Utilise des objets favorisant : Réutilisation, Efficacité, Évolutivité, Développement collaboratif)

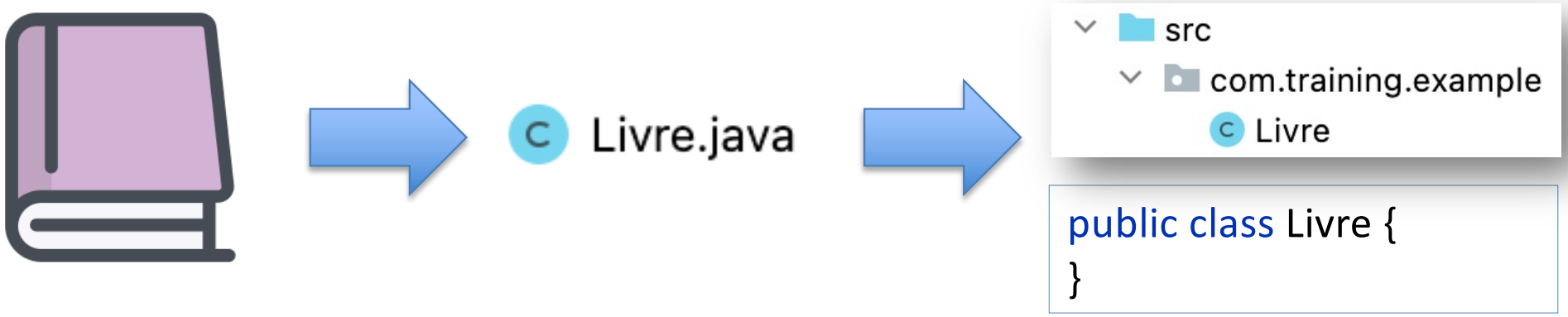
© Boris SAUVAGE - 2023

Développement JAVA

LES CLASSES

Les classes

- Programmer en Java revient à **déclarer** des classes
- Les classes sont des représentation abstraite pour les besoins de notre application
- Une classe est déclarée dans son propre fichier .java
  - Le nom du fichier doit être le même que le nom de la classe

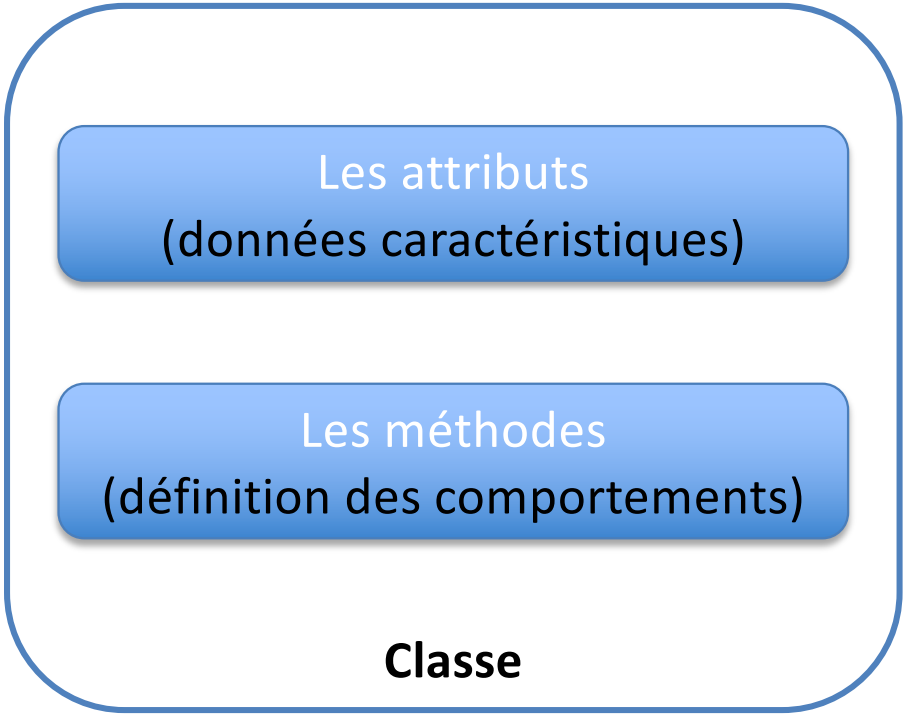


Anatomie d'une classe 1/2

- Déclarée par le mot-clé **class** suivi du nom de la classe
- Le contenu de la classe est défini par un bloc ouvert avec des accolades

Partie statique

Partie dynamique




```
public class Auteur {

// Attributs
...

// Méthodes
...


}
```

Anatomie d’une classe 2/2

- La déclaration d’une classe peut contenir de nombreux éléments :
  - des constantes
  - des classes internes
  - des énumérations
  - des attributs
  - des constructeurs
  - des méthodes

- Par convention, l’ordre des éléments ci-dessus est utilisé dans la déclaration de classe

© Boris SAUVAGE - 2023

Exemple de la classe Auteur 1/3

- Un auteur peut-être caractérisé par :
  - Un nom
  - Un prénom
  - Un age

```
public class Auteur {
    String prenom;
    String nom;
    int age;
}
```

Lise
Bourbeau
81

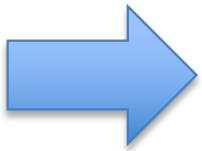
Miguel
Ruiz
69

- Ces informations (prenom, nom, age) sont des attributs d'instance

Exemple de la classe Auteur 2/3

- La classe Auteur représente un modèle pour créer des objets en mémoire

```
public class Auteur {
    String prenom;
    String nom;
    int age;
}
```



Lise
Bourbeau
81

Miguel
Ruiz
69

- Cette création d’un objet s’appelle l’instanciation d’un objet

- Un **objet** est une **instance**

Exemple de la classe Auteur 3/3

- L'instanciation de la classe Auteur va créer un objet en mémoire :

```
public class Main {

    public static void main(String[] args) {
        Auteur auteur1 = new Auteur();
        auteur1.prenom = "Lise";
        auteur1.nom = "Bourbeau";
        auteur1.age = 81;

        Auteur auteur2 = new Auteur();
        auteur2.prenom = "Miguel";
        auteur2.nom = "Ruiz";
        auteur2.age = 69;
    }
}
```

```
public class Auteur {
    String prenom;
    String nom;
    int age;
}
```

Développement JAVA

LES CLASSES EN MÉMOIRE

Instance d’un objet en mémoire

- L’instanciation de la classe Auteur va créer un objet en mémoire :

```
Auteur auteur1 = null;

auteur1 = new Auteur();

auteur1.prenom = "Lise";
auteur1.nom = "Bourbeau";
auteur1.age = 81;
```

Création d’une variable du type de la classe

Instanciation de la classe Auteur = Création d’un objet

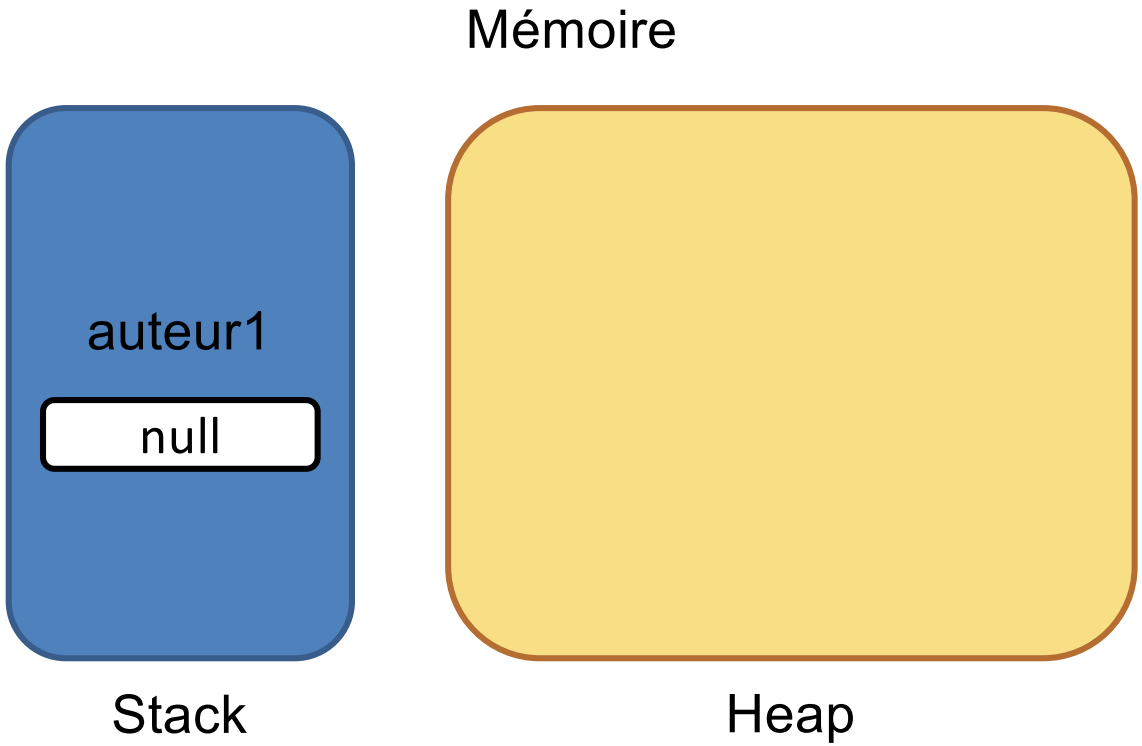
Attribution de valeurs aux attributs de l’objet

- La variable auteur1 contient la référence de l’objet créé

Notion de référence 1/4

- Création d'une variable auteur1 de type Auteur :

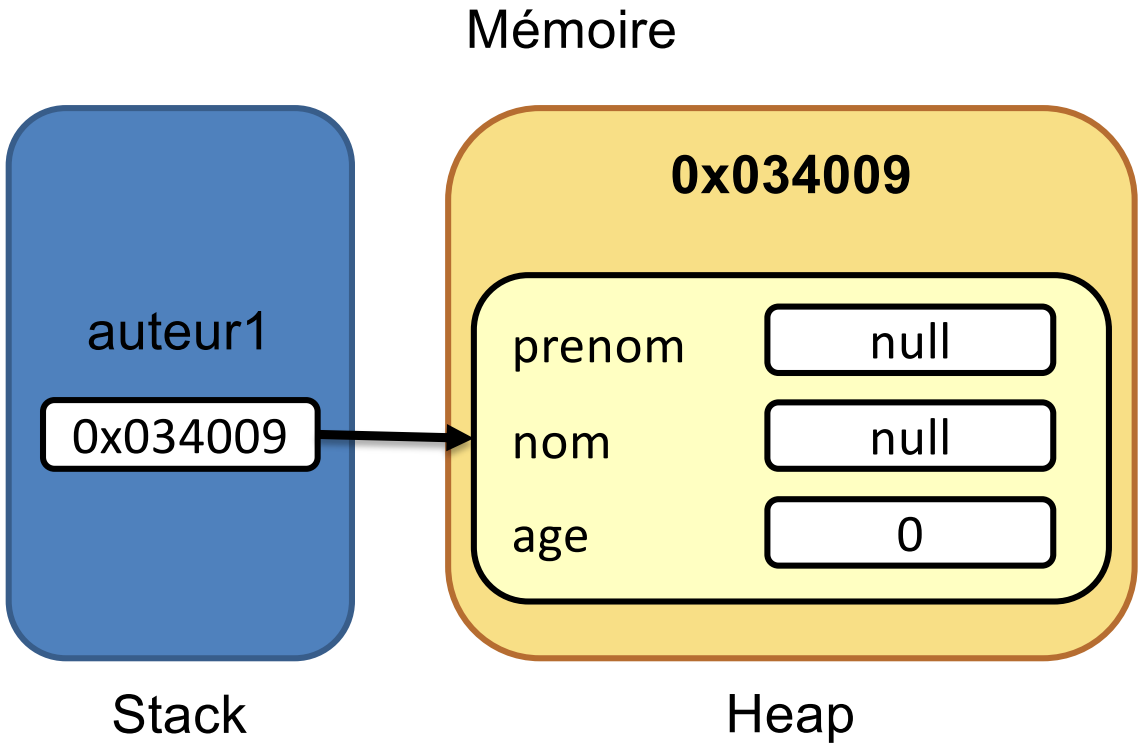
```
Auteur auteur1 = null;

auteur1 = new Auteur();

auteur1.prenom = "Lise";

auteur1.nom = "Bourbeau";

auteur1.age = 81;
```



Notion de référence 2/4

- Instanciation de la classe Auteur (création d'un objet en mémoire)
  - L'objet auteur1 possède l'adresse de son emplacement en mémoire 0x034009

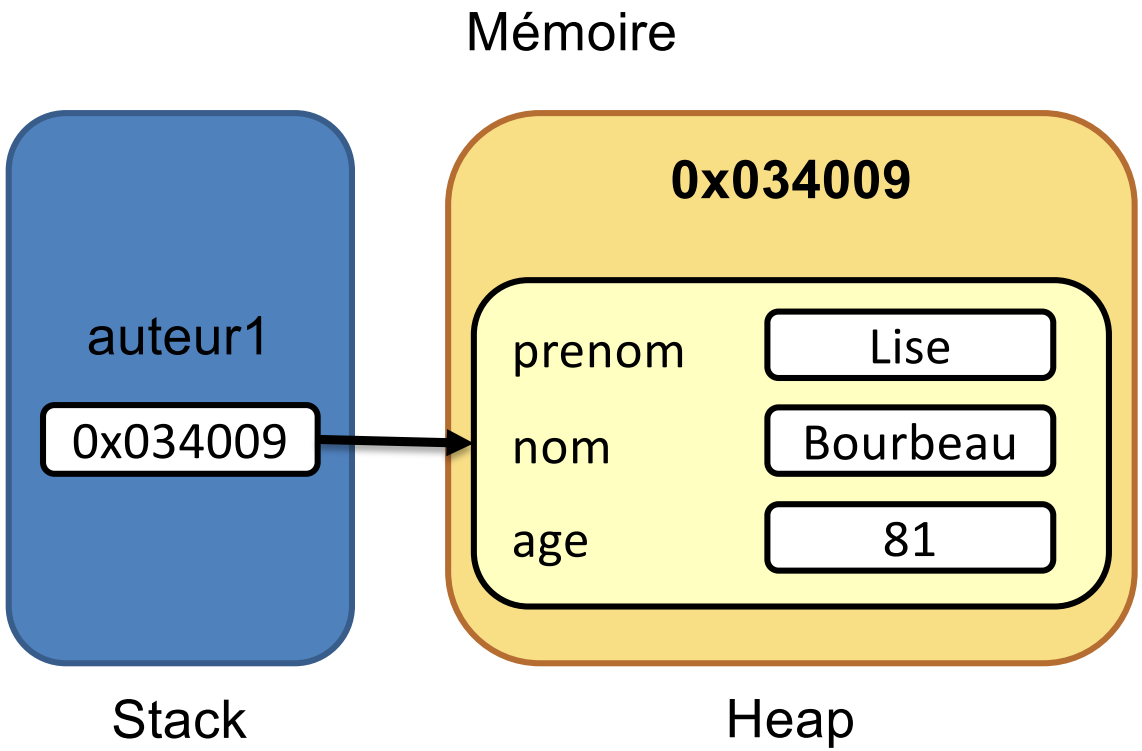
```
Auteur auteur1 = null;

auteur1 = new Auteur();

auteur1.prenom = "Lise";

auteur1.nom = "Bourbeau";

auteur1.age = 81;
```



Notion de référence 3/4

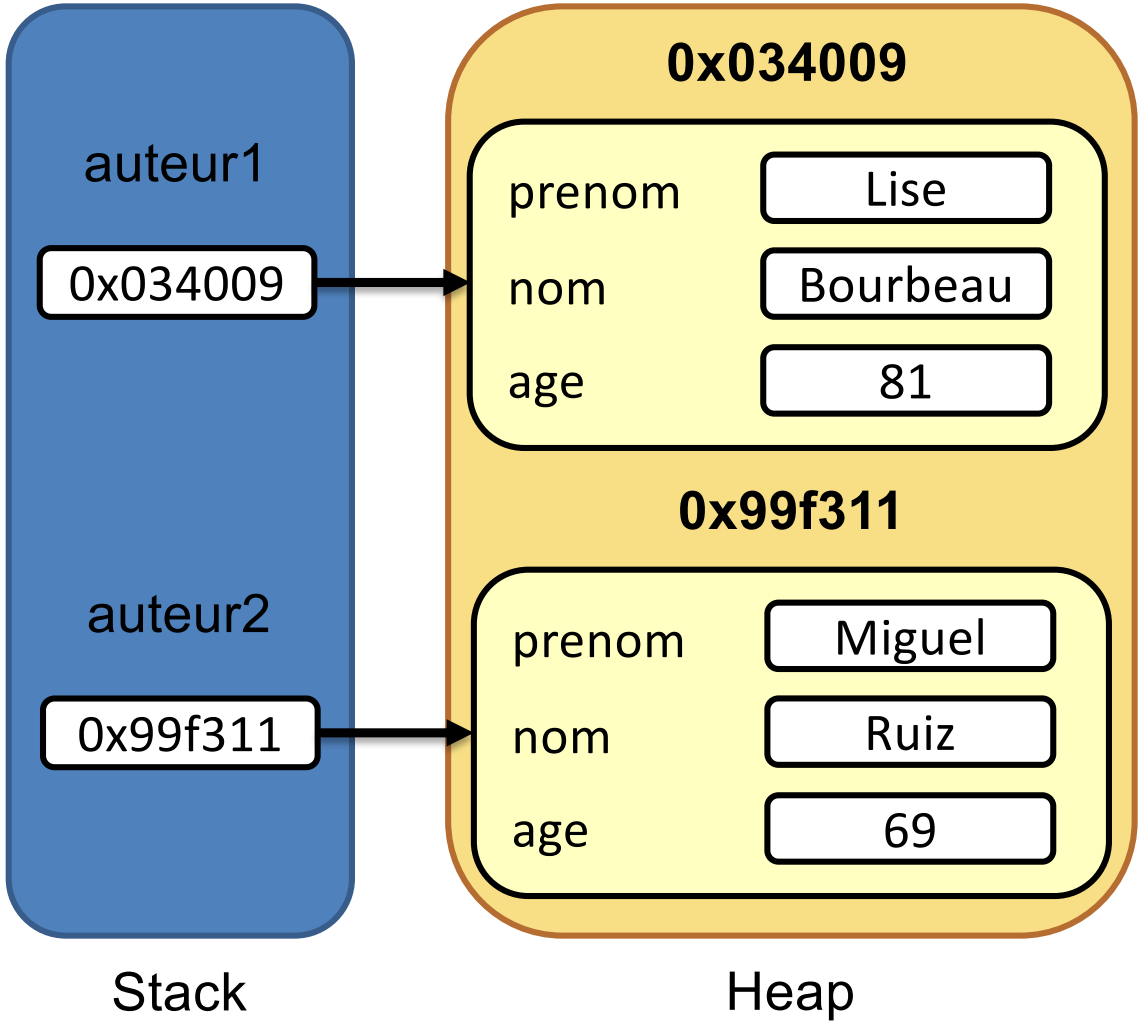
- Utilisation de la référence auteur1 pour accéder et définir les attributs dans l'objet

```
Auteur auteur1 = null;

auteur1 = new Auteur();

auteur1.prenom = "Lise";

auteur1.nom = "Bourbeau";

auteur1.age = 81;
```



Notion de référence 4/4

```
Auteur auteur1 = new Auteur();

auteur1.prenom = "Lise";
auteur1.nom = "Bourbeau";
auteur1.age = 81;


Auteur auteur2 = new Auteur();

auteur2.prenom = "Miguel";
auteur2.nom = "Ruiz";
auteur2.age = 69;
```

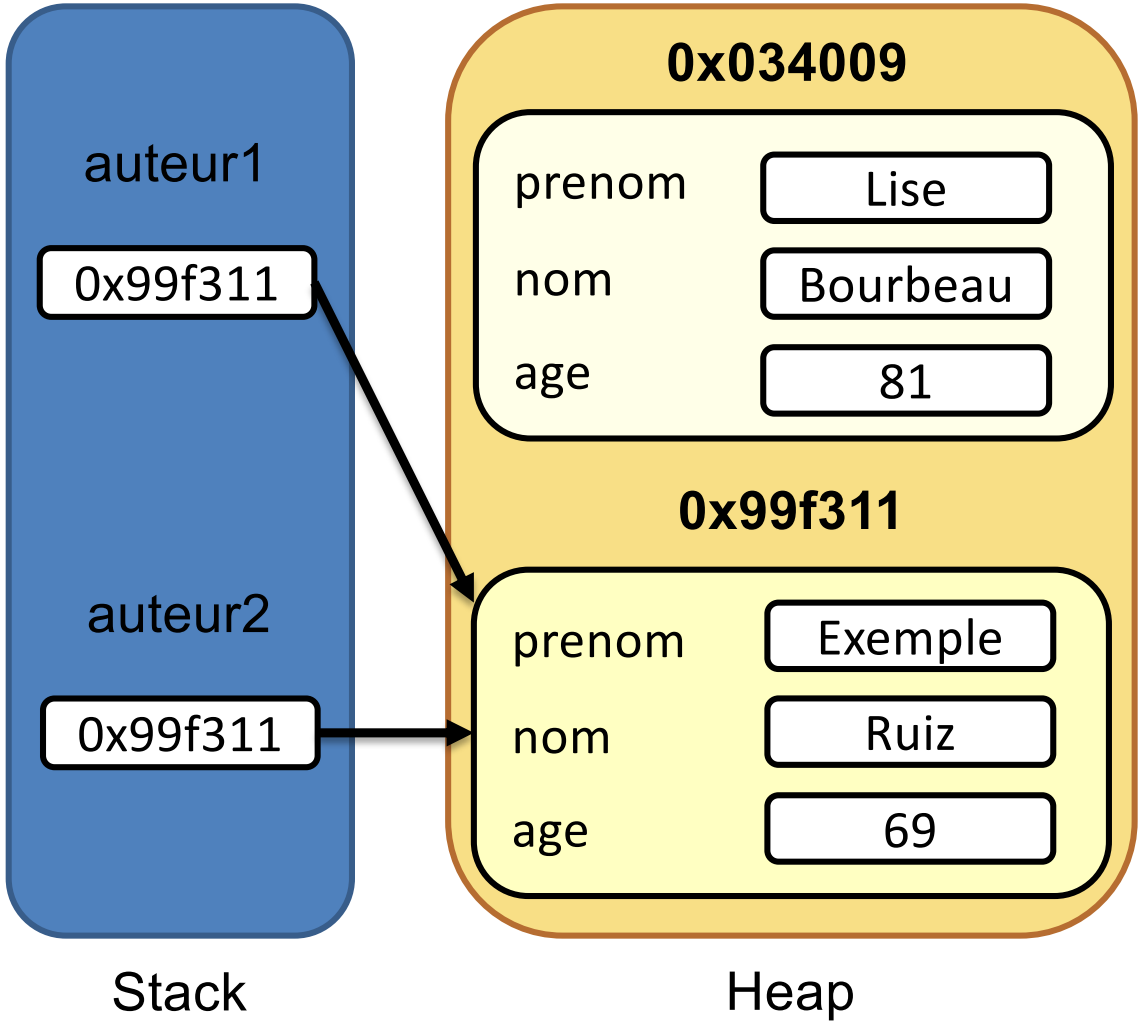


Développement JAVA

ALLEZ PLUS LOIN EN MÉMOIRE

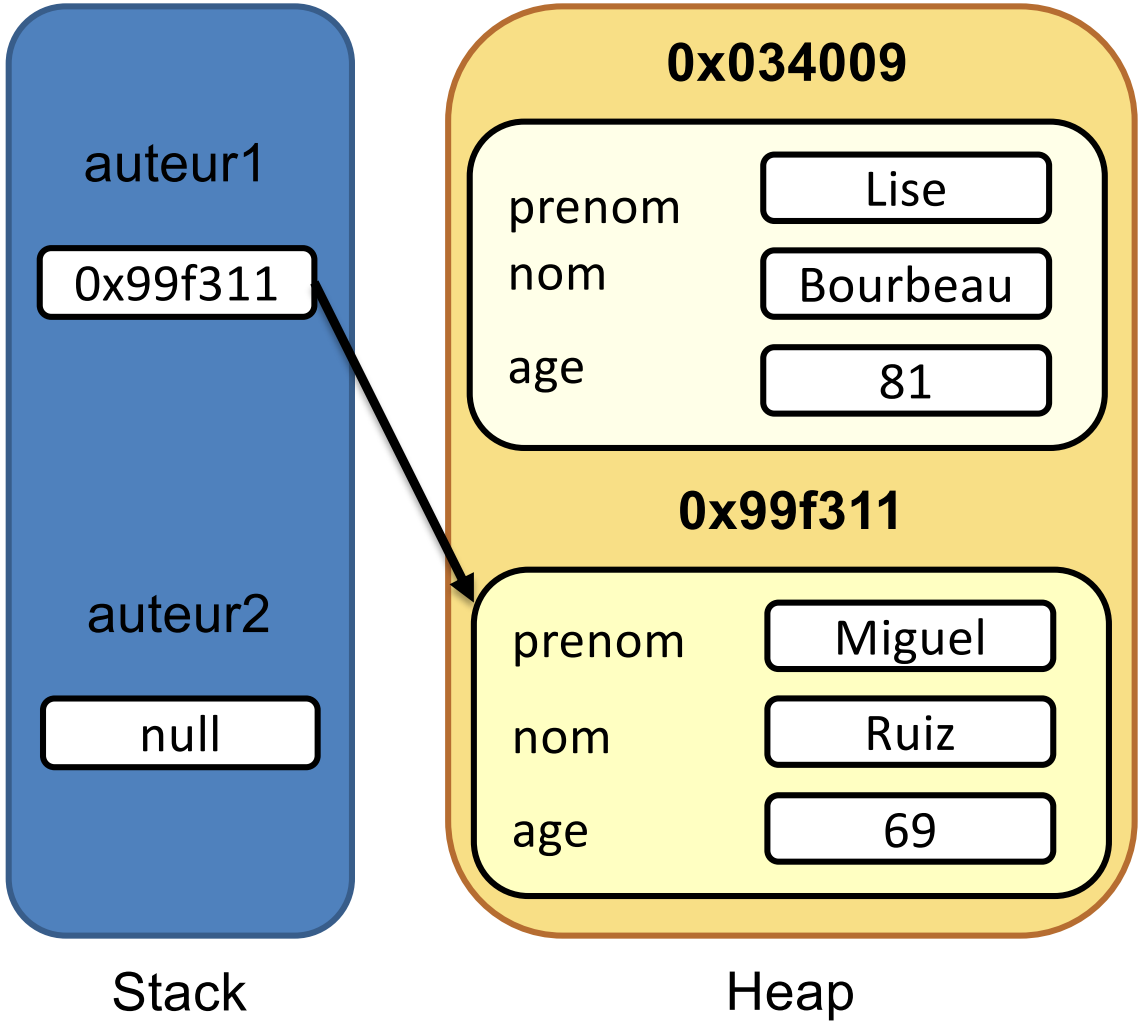
Partage d’une référence

```
Auteur auteur1 = new Auteur();
auteur1.prenom = "Lise";
auteur1.nom = "Bourbeau";
auteur1.age = 81;

Auteur auteur2 = new Auteur();
auteur2.prenom = "Miguel";
auteur2.nom = "Ruiz";
auteur2.age = 69;

auteur1 = auteur2;
auteur2.prenom = "Exemple";
```



Suppression d'une référence

```
Auteur auteur1 = new Auteur();
auteur1.prenom = "Lise";
auteur1.nom = "Bourbeau";
auteur1.age = 81;

Auteur auteur2 = new Auteur();
auteur2.prenom = "Miguel";
auteur2.nom = "Ruiz";
auteur2.age = 69;

auteur1 = auteur2;
auteur2 = null;
```



Développement JAVA

CONSTRUCTEUR & CLASSE

Les constructeurs

- Pour créer un **objet** à partir d'une classe, on utilise l'opérateur **new**
  - L'opérateur **new** fait appel au constructeur de la classe

Un constructeur porte le nom de sa classe

Un constructeur n'a pas de type de retour

```
public static void main(String[] args) {
        Auteur auteur1 = new Auteur();
}
```

```
public class Auteur {

    String prenom;
    String nom;
    int age;

    public Auteur() {
    }

}
```

Constructeur par défaut

- Pour chaque nouvelle classe, il existe un constructeur présent par défaut
  - Il est invisible dans le code de la classe, mais est présent pour Java

```
public class Auteur {

    String prenom;
    String nom;
    int age;

    public Auteur() {
    }

}
```

=

```
public class Auteur {

    String prenom;
    String nom;
    int age;

    public Auteur() {
    }

}
```

Constructeur & surcharge 1/2

- Il est possible de surcharger le constructeur par défaut
  - Le constructeur peut ainsi contenir des instructions

```
public static void main(String[] args) {
      Auteur auteur1 = new Auteur();
}
```

```
public class Auteur {

   String prenom;
   String nom;
   int age;

   public Auteur() {
   }
}
```

```
public class Auteur {

   String prenom;
   String nom;
   int age;

   public Auteur() {
      this.prenom = "John";
      this.nom = "Doe";
      this.age = 30;
   }

}
```

Constructeur & surcharge 2/2

- Il est possible de surcharger le constructeur par défaut
  - Le constructeur peut ainsi posséder des arguments

```
public class Auteur {

    String prenom;
    String nom;
    int age;

    public Auteur(String prenom, String nom, int age) {
        this.prenom = prenom;
        this.nom = nom;
        this.age = age;
    }
}
```

```
public static void main(String[] args) {
    Auteur auteur1 = new Auteur("Lise", "Bourbeau", 81);
}
```

Mot-clé this

Le mot-clé **this** permet de désigner l'objet courant

- **this** permet d'accéder aux attributs et méthodes de l'objet courant

```
public class Auteur {

    String prenom;

    public Auteur(String prenom) {
        this.prenom = prenom;
    }
}
```

```
public class Auteur {

    String prenom;

    public Auteur(String prenom) {
        prenom = prenom;
    }
}
```

Fonctionnement de la surcharge 1/3

- Sans écrire un constructeur, le compilateur créer un constructeur implicite par défaut

```
public static void main(String[] args) {
        Auteur auteur1 = new Auteur();
}
```

```
public class Auteur {
    String prenom;
    String nom;
    int age;
}
```

- Avec une surcharge, le constructeur implicite ne sera pas créer par le compilateur

```
public static void main(String[] args) {
        Auteur auteur1 = new Auteur();
}
```

java: no suitable constructor found for Auteur(no arguments)

```
public class Auteur {
    …

    public Auteur(String nom) {
        this.nom = nom;
    }
```

Fonctionnement de la surcharge 2/3

- Java différencie les différents constructeurs en se basant :
  - Sur le nombre des paramètres
  - Sur le type des paramètres

```java
Auteur auteur1 = new Auteur();
```

```java
Auteur auteur2 = new Auteur("Boris");
```

```java
Auteur auteur3 = new Auteur("Lise", "Bourbeau", 81);
```

```java
public class Auteur {
    String prenom;
    String nom;
    int age;

    public Auteur() { }


    public Auteur(String prenom) {
        this.prenom = prenom;
    }

    public Auteur(String prenom, String nom, int age) {
        this.prenom = prenom;
        this.nom = nom;
        this.age = age;
    }
}
```

Fonctionnement de la surcharge 3/3

- On **ne peut pas** créer 2 constructeurs ayant le même nombre et types de paramètres

```
public class Main {


  public static void main(String[] args) {

      Auteur auteur2 = new Auteur("Lise");

      Auteur auteur3 = new Auteur("Miguel");
   }

}
```

```
public class Auteur {
   String prenom;
   String nom;
   int age;

   public Auteur() { }

   public Auteur(String nom) {
      this.nom = nom;
   }

   public Auteur(String prenom) {
      this.prenom = prenom;
   }
}
```

java: constructor Auteur(java.lang.String) is already defined in class com.training.example.Auteur

Exercice

- Reprendre l'algorithme précédent.
  - Écrire un algorithme qui vous demande de saisir une note, un prénom et un age pour chacun de vos élèves. La saisie s'arrête dès que l'utilisateur saisit -1. Veuillez ensuite afficher la note de chaque élèves
- Déclarer une classe Student
  - contenant un prénom, un nom, une note, un age
- Revisitez votre exercice avec la programmation orientée objet

© Boris SAUVAGE - 2023

Développement JAVA

MÉTHODES

Les méthodes 1/2

- Permettent de définir le comportement des objets
- Définie par une **signature** suivie d'un bloc de code (le **corps** de méthode)
  - Comporte une portée, un type de retour, un identifiant, une liste de paramètres et un bloc de code

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

```
public class Auteur {

    public void ecrireLivre(String titre) {
        System.out.println("Écriture du livre " + titre);
    }
}
```

```
Auteur auteur2 = new Auteur();
auteur2.ecrireLivre( " Formation Java " );
```

Écriture du livre Formation Java

Les méthodes 2/2

- Les méthodes peuvent également avoir accès aux attributs de l'objet

```
public class Auteur {

   String prenom;

   public Auteur(String prenom) {
     this.prenom = prenom;
   }

   public void ecrireLivre(String titre) {
      System.out.println( "Écriture du livre " + titre + " par " + this.prenom );
   }
}
```

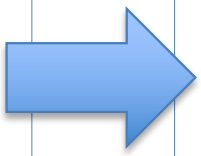
```
Auteur auteur2 = new Auteur("Boris");
auteur2.ecrireLivre( " Formation Java " );
```

Écriture du livre Formation Java par Boris

Exemple de la classe Auteur 1/3

- Un auteur peut faire les actions suivantes :
  - écrire un livre
  - Se présenter

```
public class Auteur {
    String prenom;
    String nom;
    int age;
    String[] livres;
}
```



```
public class Auteur {
    String prenom;
    String nom;
    int age;
    String[] livres;

    public void ecrireLivre(String titre) {}

    public void sePresenter() {}
}
```

- Ces comportements sont des méthodes d’instance

Exemple de la classe Auteur 2/3

- Les méthodes comportent des instructions, utilisant souvent les attributs :

```
public class Auteur {
   String prenom;
   String nom;
   int nbLivres;
   int age;
   String[] livres;

   public Auteur(String prenom, String nom) {
      this.prenom = prenom;
      this.nom = nom;
      this.livres = new String[1000];
   }

   public void ecrireLivre(String titre) {
      System.out.println(this.prenom + " écrit son livre «" + titre + "»");
      this.livres[this.nbLivres++] = titre;
   }

   public void sePresenter() {
      System.out.println(this.prenom + " " + this.nom + " est une écrivaine spécialiste du développement personnel de " + this.age + " ans.");
   }
}
```

Exemple de la classe Auteur 3/3

- L'instanciation de la classe Auteur va créer un objet en mémoire :

```
public class Main {

    public static void main(String[] args) {
        Auteur auteur1 = new Auteur("Lise", "Bourbeau", 81);
        auteur1.ecrireLivre("Écoute Ton Corps");
        auteur1.sePresenter();
    }
}
```

```
Lise écrit son livre «Écoute Ton Corps »
Lise Bourbeau est une écrivaine spécialiste du développement personnel de 81 ans.
```

Méthodes et portée 1/3

- Une méthode est définie par une portée

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

- Cette portée est la partie du programme où la méthode est **visible/utilisable**

<u>Exemples des portées en Java :</u>

```
public void maMethode() {
	System.out.println(" maMethode publique ");
}
```

```
protected void maMethode() {
	System.out.println(" maMethode protégée ");
}
```

```
private void maMethode() {
	System.out.println(" maMethode privée ");
}
```

```
void maMethode() {
	System.out.println(" maMethode par défaut ");
}
```

© Boris SAUVAGE - 2023

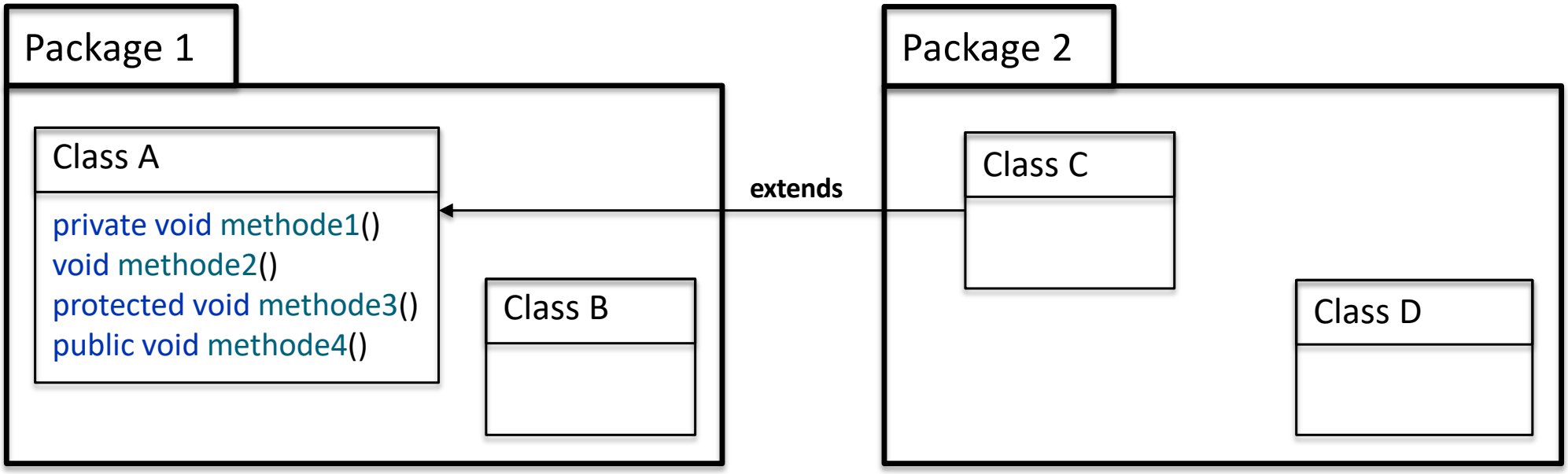
Méthodes et portée 2/3

Les niveaux d’accès sont suivants :

- **Public :** visible par toutes les classes
- **Private :** accessible uniquement depuis l'intérieur même de la classe
- **Protected :** accessible par les classes d'un package et par les sous-classes de cette classe
- **Par défaut :** accessible depuis les classes du même package

© Boris SAUVAGE - 2023

Méthodes et portée 3/3



	Classe A	Classe B	Classe C	Classe D
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

© Boris SAUVAGE - 2023

Méthodes et paramètres 1/2

- Une méthode peut contenir des paramètres

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

- <u>Exemple d'une méthode avec des paramètres :</u>

```
public static void main(String[] args) {
    Exemple e = new Exemple();
    e.afficherAddition(20,80);
}
```

```
public class Exemple {

    void afficherAddition(int a, int b) {
        System.out.println(a + b);
    }
}
```

© Boris SAUVAGE - 2023

Méthodes et paramètres 2/2

- Une méthode peut contenir des paramètres déclarés **final**
  - Le mot clé **final** indique que la valeur d'une variable ne peut pas changer

```
public class Voiture {

   void accelerer(final float deltaVitesse) {
      deltaVitesse = 0.0f;      // erreur de compilation
   }
}
```

- Java autorise uniquement le passage par copie
  - Assigner une nouvelle valeur a donc un impact restreint à la méthode

- **final** empêche les erreurs de réassignation d'un paramètre

© Boris SAUVAGE - 2023

Méthodes sans retour (void)

- Une méthode est définie par un type de retour

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

- Le type de retour précise si la méthode renvoie une valeur, ou non

Exemple d'une méthode sans retour : **void**

```
public static void main(String[] args) {
    Exemple e = new Exemple();
    e.afficherAddition(20,80);
}
```

```
public class Exemple {

    void afficherAddition(int a, int b) {
        System.out.println(a + b);
    }
}
```

© Boris SAUVAGE - 2023

Méthodes avec un retour (return)

- Une méthode est définie par un type de retour

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

- Le type de retour précise si la méthode renvoie une valeur, ou non

Exemple d'une méthode avec un retour : **un type entier est retourné**

```
public static void main(String[] args) {

    Exemple e = new Exemple();
    int result = e.addition(2,8);

}
```

```
public class Exemple {

    int addition(int a, int b) {
        return a + b;
    }

}
```

Méthodes et identifiant

- Une méthode est définie par un identifiant (= un nom de méthode)

```
[portée] [type de retour] [identifiant] ([liste des paramètres]) {
  [code]
}
```

- Les noms de méthodes doivent contenir un verbe
- La première lettre est obligatoirement une minuscule
- La notation camelCase doit être utilisée pour le nommage
- Le choix de la langue anglaise est préférable

Exemple de plusieurs méthodes :

```
User[] findAllUsers() {
      return users;
}
```

```
public User findUserById(Long id) {
      return user;
}
```

© Boris SAUVAGE - 2023

Destructeur

- Au sein d'un projet Java, la mémoire est automatiquement libérée
- Le Garbage Collector s'exécute dès que la mémoire dépasse un certain seuil
- Les objets instanciés seront donc automatiquement détruits

Plusieurs bonnes pratiques :

- limiter le nombre d'objets avec la même valeur
- Essayer de recycler vos objets

© Boris SAUVAGE - 2023

Installation

Exercice

- Réaliser le projet Banque interactive

Développement JAVA

MÉTHODES DE CLASSE

Méthodes de classe – Fonction 1/4

- Les méthodes définissent les comportement d'un objet
  - Elles peuvent accéder aux attributs de l'instance

- **Les méthodes de classe** ne peuvent pas accéder aux attributs d'un objet

Elles sont souvent utilisées comme des méthodes utilitaires
  - Pas d'instance nécessaire
  - Aussi connues en tant que fonctions

© Boris SAUVAGE - 2023

Méthodes de classe – Fonction 2/4

- Pour déclarer une méthode de classe, on utilise le mot clé **static**

```
public class MathUtil {

  public static int additionner(int... valeurs) {
      int resultat = 0;
      for (int valeur : valeurs) {
         resultat += valeur;
      }
      return resultat;
   }
}
```

```
public class Test {


   public static void main(String[] args) {

      int resultat = MathUtil.additionner(1, 2, 3, 4);

   }

}
```

© Boris SAUVAGE - 2023

Méthodes de classe – Fonction 3/4

- Certaines classes de l'API Java ne contiennent que des méthodes de classe
- Voici quelques classes utilitaires disponibles en Java :
  - java.lang.Math
  - java.lang.System
  - java.util.Arrays
  - java.util.Collections

```
public static void main(String[] args) {
   System.out.println();
   Math.random();
   Arrays.toString(new int[]{1,2,3});
}
```

© Boris SAUVAGE - 2023

Méthodes de classe – Fonction 4/4

- Il est possible d'invoquer une méthode de classe avec une instance de cette classe :

```
public static void main(String[] args) {

    int resultat = new MathUtil().additionner(1, 2, 3, 4);
}
```

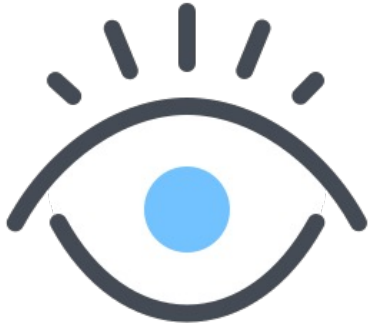
Développement JAVA

PORTÉE DES NOMS

Portée des noms 1/3

- Le nom d'une variable est limité au bloc de code dans lequel il a été déclaré

```
public int exempleTraitement(int valeurMax) {
   int resultat = 0;
   // la variable i n'est accessible que dans la boucle for
   for (int i = 0; i < 10; ++i) {
      // la variable k n'est accessible que dans la boucle for
      for (int k = 0; k < 10; ++k) {
         // la variable m n'est accessible que dans ce bloc
         int m = resultat + i * k;
         if (m > valeurMax) {
            return valeurMax;
         }
         resultat = m;
      }
   }
   return resultat;
}
```

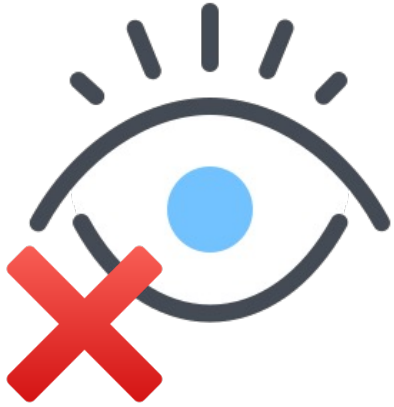


Portée des noms 2/3

- **Impossible** de nommer une variable ayant le même nom qu'un paramètre/variable
  - Accessible dans le bloc de code courant

```
public int exempleTraitement(int valeurMax) {
        int valeurMax = 2;    // ERREUR DE COMPILATION
}
```

```
public int exempleTraitement(int valeurMax) {
   int resultat = 0;
   for (int i = 0; i < 10; ++i) {
      resultat += i;
      if (resultat > 10) {
         int resultat = -1; // ERREUR DE COMPILATION
         return resultat;
      }
   }
   return resultat;
}
```



Portée des noms 3/3

- **Possible** de réutiliser un nom de variable dans 2 blocs de code successifs
  - Peut-être utile pour des variables temporaires

```
public int exempleTraitement(int valeurMin , int valeurMax) {

  for (int i = 0; i < valeurMax; ++i) {
  ...
  }

  // on peut de nouveau utiliser la variable i
  for (int i = 0; i < valeurMin; --i) {
  ...
  }

}
```

Développement JAVA

ENCAPSULATION

Le principe d'encapsulation

- Un objet ne doit pas exposer sa **représentation interne** au **monde extérieur**

représentation interne d'un objet:
- Les attributs
- Les méthodes

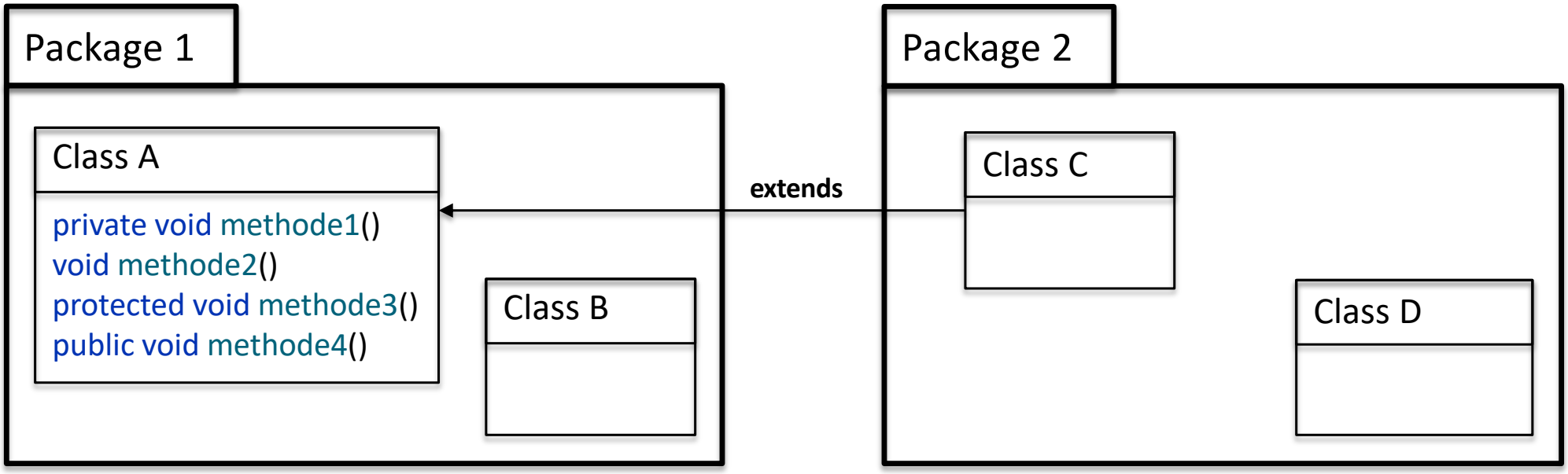
monde extérieur :
- Les autres classes dans le même package
- Les classes des autres packages

- Le principe d'encapsulation s'applique à différents niveaux :
  - Les classes
  - Les attributs
  - Les méthodes

- Les mots-clés **public**, **protected**, **private** et **Ø** sont utilisés pour appliquer ce principe

© Boris SAUVAGE - 2023

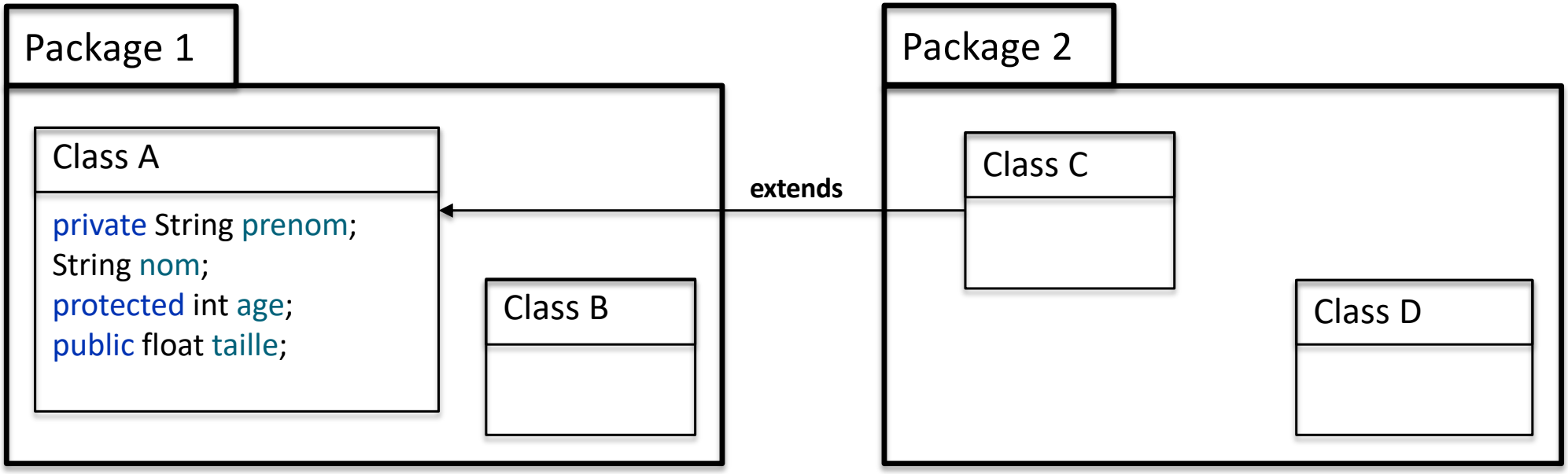
L'encapsulation des méthodes



	Classe A	Classe B	Classe C	Classe D
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

© Boris SAUVAGE - 2023

L'encapsulation des attributs



	Classe A	Classe B	Classe C	Classe D
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

© Boris SAUVAGE - 2023

Les accesseurs et les mutateurs 1/2

- L'encapsulation cache les attributs et les méthodes d'une classe
  - permet un meilleur contrôle et une meilleure maintenance de l'état de la classe

- Il existe des méthodes qui permettent d’accéder au contenu des propriétés privé
  - Les accesseurs (getters)
  - Les mutateurs (setters)

- Les accesseurs sont utilisés pour récupérer la valeur d’un attribut

- Les mutateurs sont utilisés pour définir ou mettre à jour la valeur d’un attribut

© Boris SAUVAGE - 2023

Les accesseurs et les mutateurs 2/2

- Exemple d'une classe avec des accesseurs et des mutateurs :

```
public class Person {

   private String name; // Attribut privé de classe

   // Getter pour récupérer la valeur de l'attribut name
   public String getName() {
      return name;
   }

   // Setter pour définir ou mettre à jour la valeur de l'attribut name
   public void setName(String name) {
      this.name = name;
   }
}
```

Remarque : Les getters & setters respectent la convention **get**Attribut() et **set**Attribut()

Développement JAVA

HÉRITAGE

Héritage et composition

- Une application Java est composée d'un ensemble d'objets
  - La POO réside dans les relations que ces objets entretiennent les uns avec les autres

- Ces relations faites par les développeurs constituent l'architecture d'une application

- Il existe 2 relations fondamentales en programmation objet :

Est un (*is-a*) : permet d'identifier une **relation d'héritage**

A un (has-a) : permet d'identifier une **relation de composition**

© Boris SAUVAGE - 2023

Héritage (Est un) 1/3

- Dans une Bibliothèque, nous souhaitons gérer des documents de diverses natures :

```
public class Livre {

    private int reference;
    private String titre;
    private Auteur auteur;

}
```

```
public class Revue {
    private int reference;
    private String titre;
    private int mois;
    private int annee;

}
```

- Ici, les objets **Livre** et **Revue** partagent des attributs communs

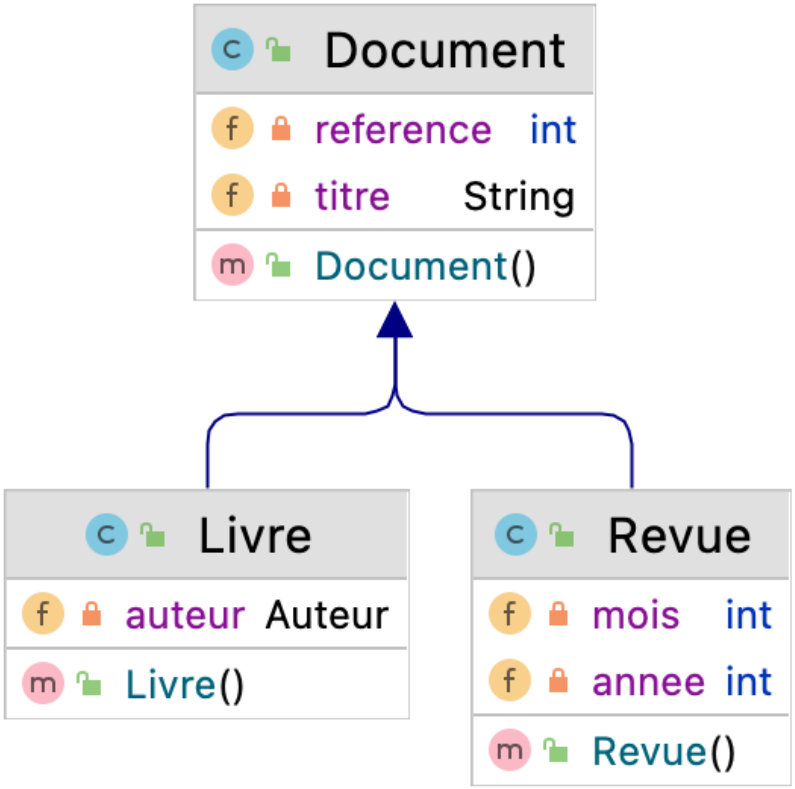
- Ainsi, il est possible de factoriser les notions communes dans une classe

© Boris SAUVAGE - 2023

Héritage (Est un) 2/3

- **Livre** et **Revue** héritent de la classe **Document**

- L'héritage est indiqué par **extends**
  - **Document** est la classe mère ou parente
  - **Livre** et **Revue** sont les classes filles ou enfant



Héritage (Est un) 3/3

```
public class Document {
    private int reference;
    private String titre;
}
```

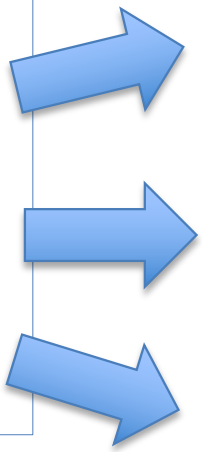
```
public class Livre extends Document {
    private Auteur auteur;
}
```

```
public class Revue extends Document {
    private int mois;
    private int annee;
}
```

Héritage et Méthodes

- Il est possible de déclarer des méthodes dans la classe *Document*
  - Ainsi, les classes *Livre* et *Revue* peuvent en bénéficier

```
public class Document {
   …

   public void etreLu(){
         System.out.println("Je suis lu");
   }

}
```



```
Document doc = new Document();
doc.etreLu();

Livre livre = new Livre();
livre.etreLu();

Revue revue = new Revue();
revue.etreLu();
```

Héritage multiple

- **Une classe hériteras au maximum d'une seule classe mère**

```
public class Article {
    private int reference;
    private double prix;
}
```

```
public class Document {
    private int reference;
    private String titre;
}
```

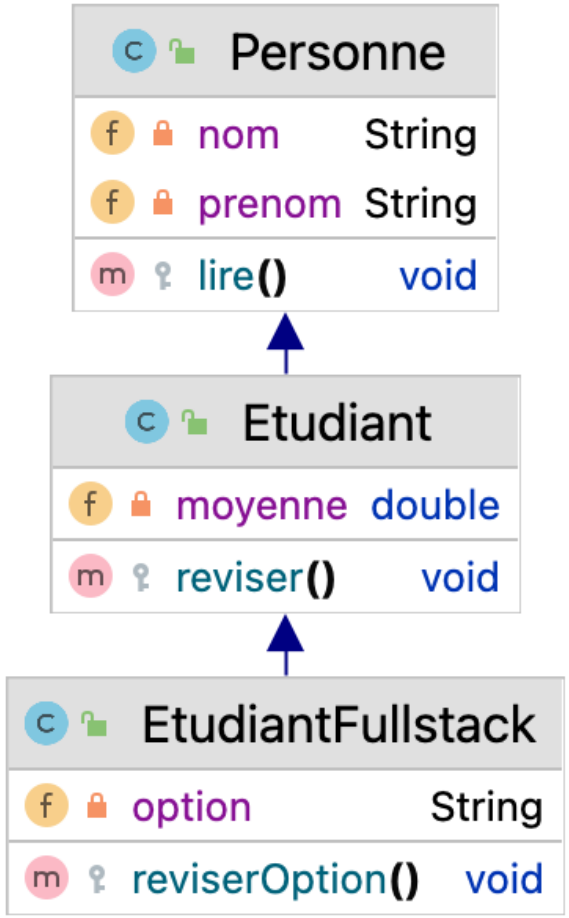
```
public class Livre extends Document , Article {
    private Auteur auteur;
}
```

Héritage en cascade 1/2

- **Etudiant** hérite de Personne
- **EtudiantFullstack** hérite de Etudiant **et** de Personne

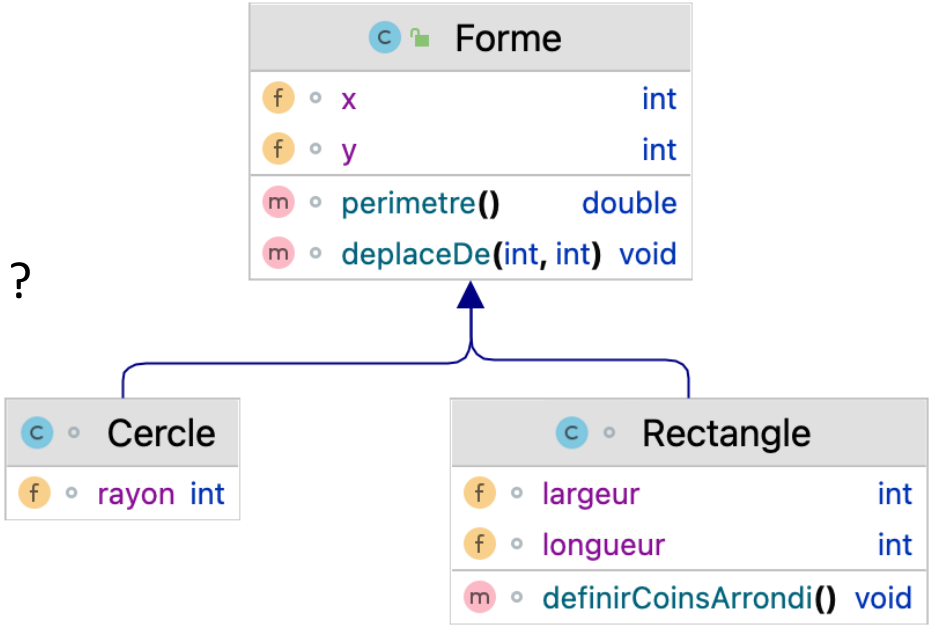
```
Etudiant etudiant = new …
etudiant.lire();
etudiant.reviser();
etudiant.reviserOption();
```

```
EtudiantFullstack etudiantF = new …
etudiantF.lire();
etudiantF.reviser();
etudiantF.reviserOption();
```

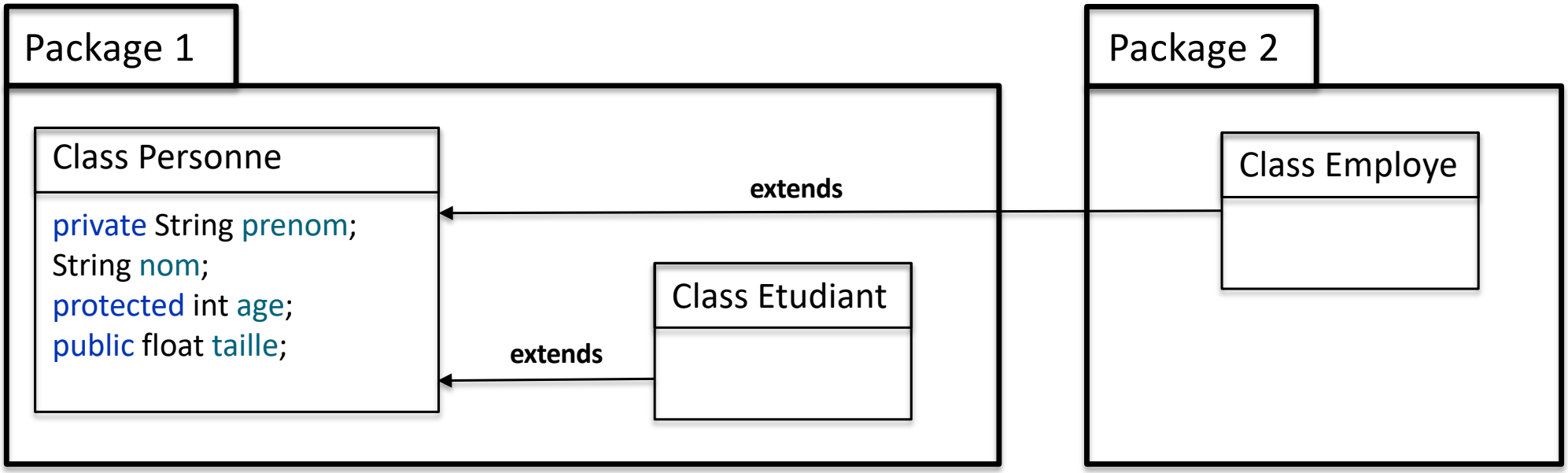


Héritage en cascade 2/2

- Quel nombre d'attributs possède Cercle ?
- Quel nombre d'attributs possède Rectangle ?
- Combien de méthodes Cercle possède -t-elle ?
- Combien de méthodes Rectangle a-t-elle ?
- Rectangle peut-il invoquer deplaceDe() ?
- Cercle peut-il invoquer definirCoinsArrondi() ?



Héritage et encapsulation



Visibilité	Personne	Etudiant	Employe
prenom	✓		
nom	✓	✓	
age	✓	✓	✓
taille	✓	✓	✓

© Boris SAUVAGE - 2023

Héritage et Constructeur 1/3

- Il est possible de déclarer un constructeur dans la classe *Document* et d'appeler ce constructeur depuis les constructeurs de *Livre* et *Revue*

```
public class Document {
    private int reference;
    private String titre;

    public Document(){
        this.reference = 123;
        this.titre = "Exemple";
    }
}
```

```
public class Livre extends Document {

    public Livre(){
        super();
    }
}
```

```
public class Revue extends Document {

    public Revue(){
        super();
    }
}
```

super() fait appel au constructeur par défaut de la classe mère

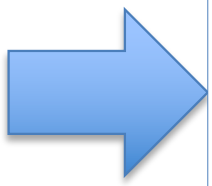
© Boris SAUVAGE - 2023

Héritage et Constructeur 2/3

- Il est possible de surcharger le constructeur de la classe *Document,* à condition d'adapter l'appel du **super()**

```
public class Document {
    private int reference;
    private String titre;

    public Document(String titre){
        this.reference = 123;
        this.titre = titre;
    }
}
```

super() fait appel au constructeur surchargé de la classe mère



```
public class Livre extends Document {

    public Livre(String titre){
        super(titre);
    }
}
```

```
public class Revue extends Document {

    public Revue(){
        super("Science et Vie");
    }
}
```

© Boris SAUVAGE - 2023

Héritage et Constructeur 3/3

- Tester l'exemple suivant.

```
public class Animal {

    private int nbPattes;

    public Animal(int nbPattes){
        this.nbPattes = nbPattes;
    }
}
```

```
public class Chat extends Animal {

}
```

- Comment résoudre le résultat ?

© Boris SAUVAGE - 2023

Enchaînement des constructeurs 1/2

- Lors d'un héritage en cascade, il y a un enchainement des constructeurs

```
public class Personne {

  public Personne(){
        System.out.println("Je suis personne");
  }
}
```

```
public class Etudiant extends Personne {

  public Etudiant(){
        System.out.println("Je suis étudiant");
  }
}
```

```
public class EtudiantFullStack extends Etudiant {

  public EtudiantFullStack(){
        System.out.println("Je suis étudiant fullstack");
  }
}
```

```
EtudiantFullStack ef = new EtudiantFullStack();
```

```
Je suis personne
Je suis étudiant
Je suis étudiant fullstack
```

Enchaînement des constructeurs 2/2

- Lors d'un héritage en cascade, il y a un enchainement des constructeurs

```
public class Personne {

  public Personne(){
        super();
        System.out.println("Je suis personne");
  }
}
```

```
public class Etudiant extends Personne {

  public Etudiant(){
        super();
        System.out.println("Je suis étudiant");
  }
}
```

```
public class EtudiantFullStack extends Etudiant {

  public EtudiantFullStack(){
        super();
        System.out.println("Je suis étudiant fullstack");
  }
}
```

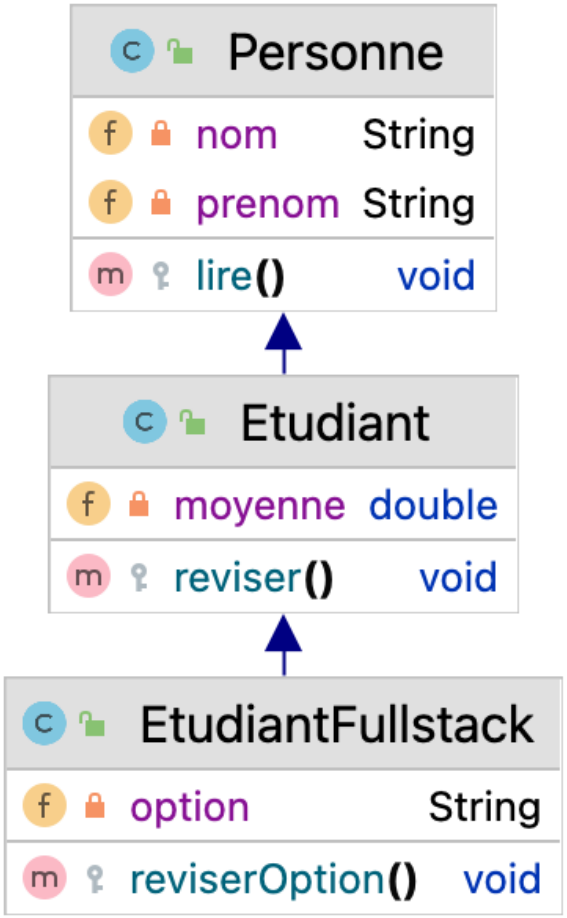
Appel implicite à super

Exemple d'utilisation

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

   public Personne(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
   }
}
```

```java
public class Etudiant extends Personne {
   private double moyenne;

   public Etudiant(String nom, String prenom, double moyenne) {
      super(nom, prenom);
      this.moyenne = moyenne;
   }
}
```

```java
public class EtudiantFullstack extends Etudiant {
   private String option;

   public EtudiantFullstack(String nom, String prenom, double moyenne, String option) {
      super(nom, prenom, moyenne);
      this.option = option;
   }
}
```



Développement JAVA

POLYMORPHISME

Polymorphisme : Introduction

- N'importe quel objet en Java qui peut passer d'une relation « est un » peut être considéré polymorphe

- Le polymorphisme est le fait de référencer une classe **fille** avec une référence déclaré de type une classe **mère**

Exemple :

```
Animal chien = new Chien();
```

Référence déclaré en tant qu'Animal().

Objet crée en tant que Chien().

```
public class Animal {
}
```

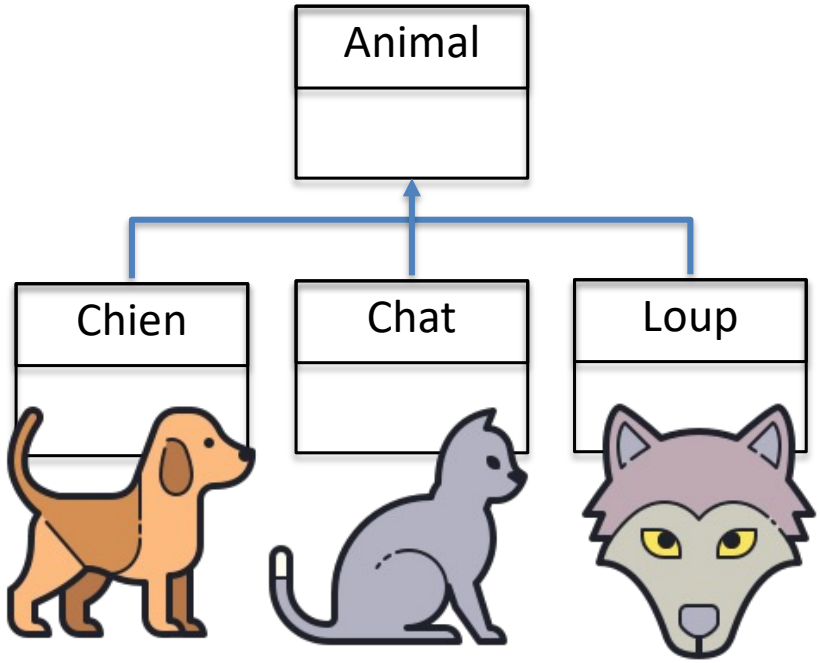
```
public class Chien {
}
```

Polymorphisme : exemple 1/3

- Le type de la référence peut être la classe mère de l'objet instancié

<u>Exemple :</u> Utilisation d'un tableau polymorphique

```
Animal[] animals = new Animal[3];
animals [0] = new Chien();
animals [1] = new Chat();
animals [2] = new Loup();
```



Polymorphisme : exemple 2/3

- Les classes filles peuvent redéfinir les comportements de la classe mère
  - L'annotation **@Override** est utilisée pour indiquer cette **redéfinition**

```java
public class Animal {

    public void mange() {
        System.out.println("En tant qu'Animal, je mange");
    }

    public void chasse() {
        System.out.println("En tant qu'Animal, je chasse");
    }

}
```

```java
public class Chat extends Animal {

    public class Chien extends Animal {

        public class Loup extends Animal {

            @Override
            public void mange() {
                System.out.println("En tant que Loup, je mange");
            }

            @Override
            public void chasse() {
                System.out.println("En tant que Loup, je chasse");
            }
        }
    }
}
```

Polymorphisme : exemple 3/3

- Avec le polymorphisme, les classes sont regroupées et utilisables sous un même type

```java
public static void main(String[] args) {

    Animal[] animals = new Animal[4];
    animals[0] = new Chat();
    animals[1] = new Chien();
    animals[2] = new Loup();
    animals[3] = new Animal();

    for (Animal animal : animals) {
      System.out.println("######");
      animal.chasse();
      animal.mange();
    }

}
```

```
######
En tant que Chat, je chasse
En tant que Chat, je mange
######
En tant que Chien, je chasse
En tant que Chien, je mange
######
En tant que Loup, je chasse
En tant que Loup, je mange
######
En tant qu'Animal, je chasse
En tant qu'Animal, je mange
```

Développement JAVA

SURCLASSEMENT VS SUBSTITUTION

Surclassement vs Substitution

- Un objet peut être manipulé grâce à un type :
  - Le type de la classe mère
  - Le type de la classe fille

- Les conversions de types d'un objet sont courantes en Java
  - Il existe deux conversion possibles

Surclassement (UpCasting)

Changement d'un objet de type enfant en un objet de type parent

Substitution (DownCasting)

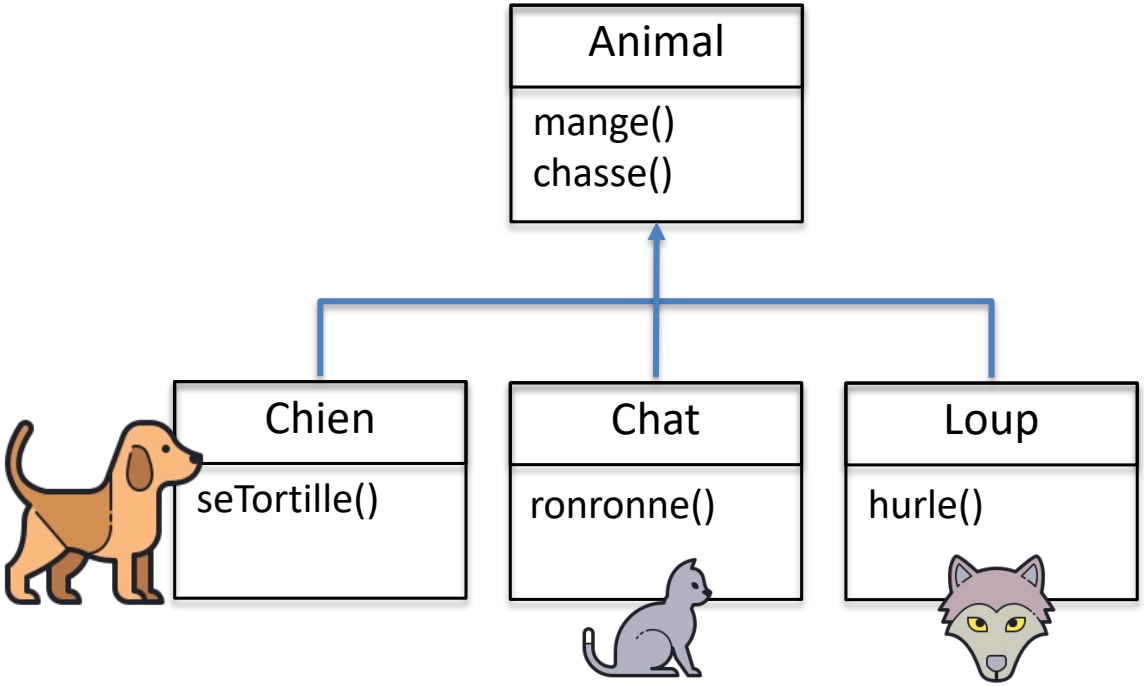
Changement d'un objet de type parent en un objet de type enfant

© Boris SAUVAGE - 2023

Substitution (DownCasting)

```
Animal tommy = new Chien();

tommy.mange();
tommy.chasse();
tommy.seTortille();
```

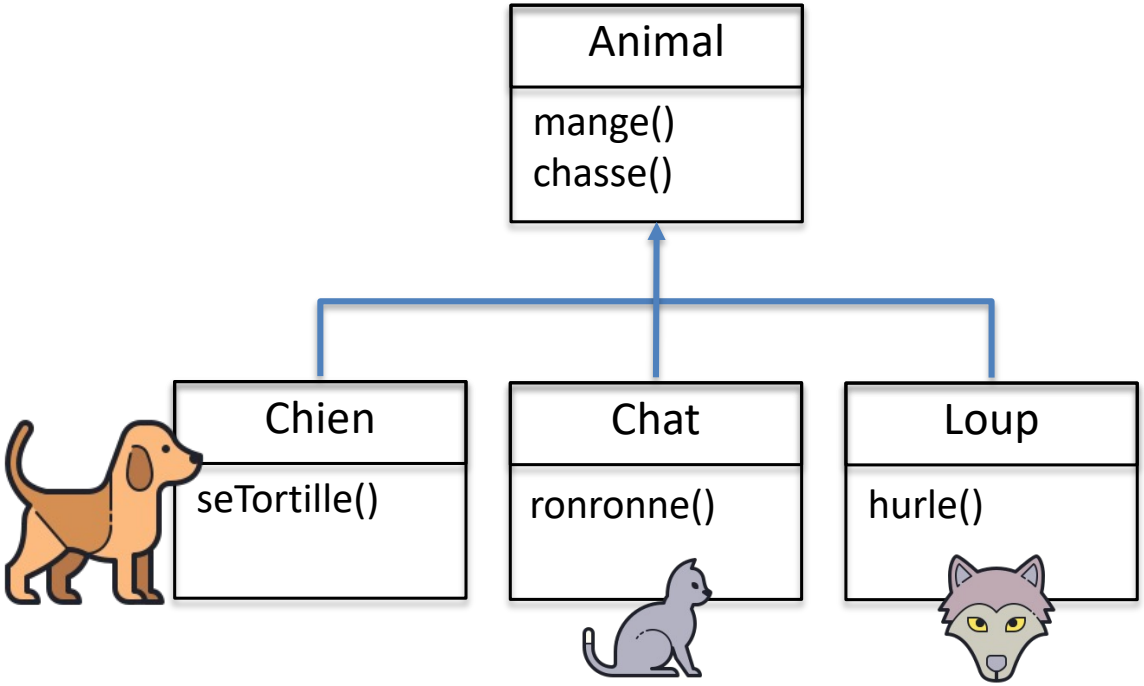
```
Chien tomChien = (Chien) tommy;

tomChien.mange();
tomChien.chasse();
tomChien.seTortille();
```



Surclassement (UpCasting)

```
Chien tomChien = new Chien();

tomChien.mange();
tomChien.chasse();
tomChien.seTortille();
```

```
Animal tommy = tomChien;

tommy.mange();
tommy.chasse();
tommy.seTortille();
```



Surclassement vs Substitution

- Un objet est surclassé implicitement

```
Chien tomChien = new Chien();

Animal tommy = tomChien;
```

- Un objet peut-être substituer de plusieurs manières :

```
Animal tommy = new Chien();

Chien tomChien = (Chien) tommy;
tomChien.seTortille();
```

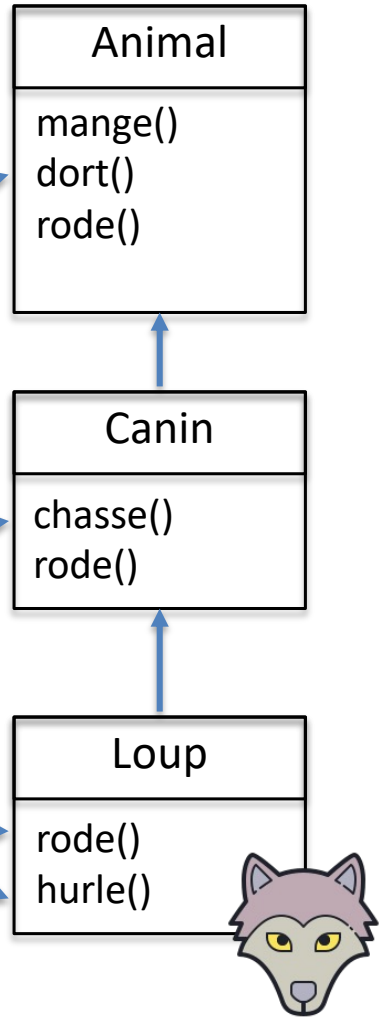
```
Animal tommy = new Chien();

((Chien) tommy).seTortille();
```

© Boris SAUVAGE - 2023

Polymorphisme et Réécriture

- Quelle est la méthode appelée ?
  - Le type le plus spécialisé l'emporte toujours

```
Loup loup = new Loup();

loup.hurle();

loup.dort();

loup.chasse();

loup.rode();
```



Développement JAVA

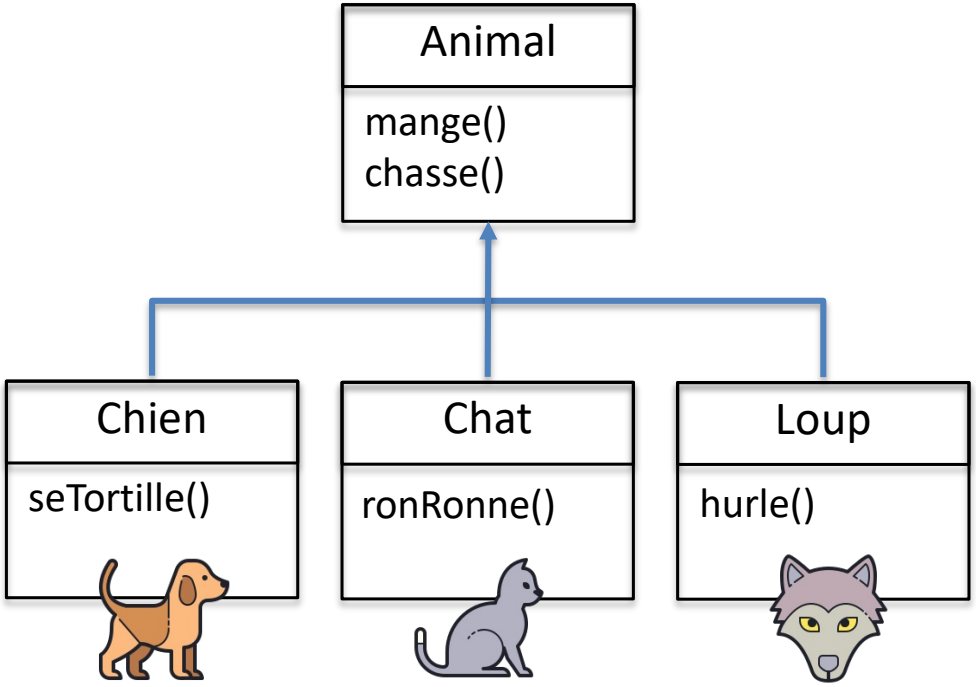
CLASSE ABSTRAITE

Classe abstraite 1/5

- Avec l'héritage, Il est possible de mutualiser du code dans une classe parente

- Parfois, cette classe est abstraite
  - Réellement, Comment mange un Animal ?
  - Comment chasse-t-il ?
  - ..

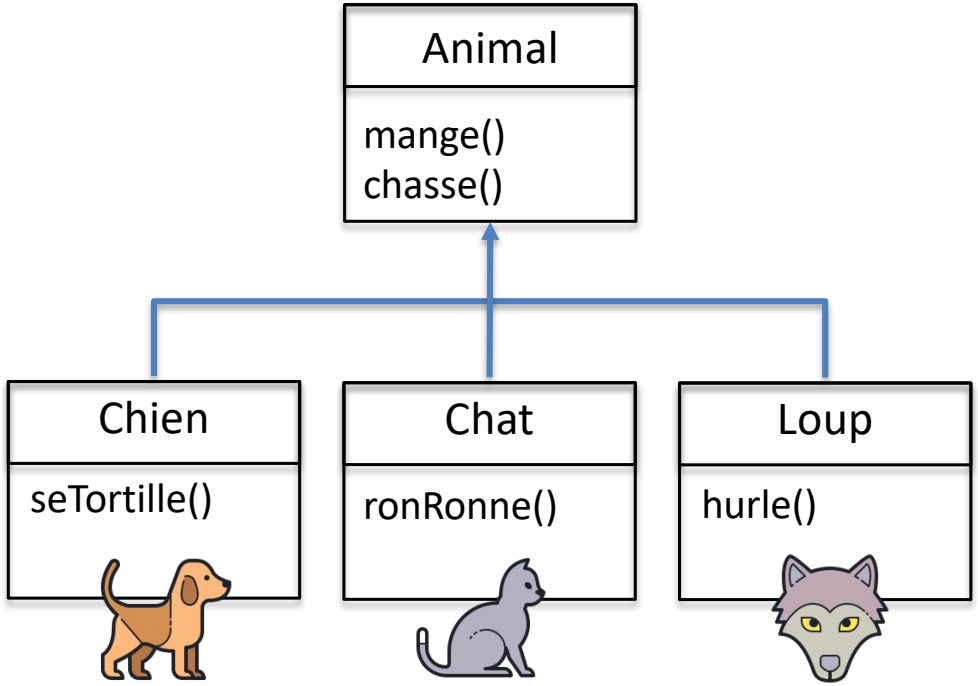
La classe Animal représente une abstraction



Classe abstraite 2/5

- Cette abstraction se matérialise par une classe abstraite

```
public abstract class Animal {

    private String espece;
    private float taille;


    protected abstract void mange();

    protected abstract void chasse();

}
```



Classe abstraite 3/5

- Les classes filles peuvent hériter de la classe abstraite
  - Elles sont **obligées** d'implémenter les comportements **mange() et chasse()**

```java
public abstract class Animal {

    private String espece;
    private float taille;



    protected abstract void mange();



    protected abstract void chasse();

}
```

```java
public class Chat extends Animal {
```

```java
public class Chien extends Animal {
```

```java
public class Loup extends Animal {


    @Override
    public void mange() {
        System.out.println("En tant que Loup, je mange");
    }

    @Override
    public void chasse() {
        System.out.println("En tant que Loup, je chasse");
    }
}
```

Classe abstraite 4/5

- Une classe abstraite n'est pas instanciable directement
  - Est utilisable par les classes filles

```
public static void main(String[] args) {

    Animal[] animals = new Animal[4];
    animals[0] = new Chat();
    animals[1] = new Chien();
    animals[2] = new Loup();
    animals[3] = new Animal();

    for (Animal animal : animals) {
        System.out.println("######");
        animal.chasse();
        animal.mange();
    }
}
```

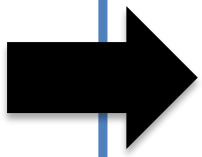
```
######
En tant que Chat, je chasse
En tant que Chat, je mange
######
En tant que Chien, je chasse
En tant que Chien, je mange
######
En tant que Loup, je chasse
En tant que Loup, je mange
######
```

© Boris SAUVAGE - 2023

Classe abstraite 5/5

- Une classe abstraite peut contenir des méthodes qui ne sont pas abstraites

```java
public abstract class Animal {

    private String espece;
    private float taille;


    protected abstract void mange();
    protected abstract void chasse();

    protected void regarde(){
        System.out.println("regarde...");
    }

}
```



```java
public static void main(String[] args) {

    Animal loup = new Loup();
    animal.mange();
    animal.chasse();
    animal.regarde();
}
```

Développement JAVA

INTERFACE

Interface 1/5

- **Rappel :** Impossible d'hériter de plusieurs classes
  - Le multi-héritage n'existe pas en java

```
public abstract class Animal {

 …

}
```

```
public abstract class Calin {

    abstract void faireUnCalin();

}
```

```
public class Chat extends Animal , Calin {


}
```

Interface 2/5

- Une interface constitue un ensemble d'exigences pour les classes qui s'y conforment
  - Le comportement **faireUnCalin()** doit obligatoirement être implémenté dans **Chat**

```
public abstract class Animal {

 ...

}
```

```
public interface Calin {

    abstract void faireUnCalin();

}
```

```
public class Chat extends Animal implements Calin {

   @Override
   public void faireUnCalin() {
}
```

- La classe Chat **hérite** de la classe Animal et **implémente** l'interface Calin

Interface 3/5

- Un objet peut être référencer par le type de l'interface Calin

```java
public abstract class Animal {
   private String espece;
   private float taille;

   protected abstract void mange();
   protected abstract void chasse();

   protected void regarde(){
      System.out.println("regarde…");
   }
}
```

```java
public interface Calin {

    abstract void faireUnCalin();
}
```

```java
public static void main(String[] args) {


      Animal loup = new Loup();
      animal.mange();
      animal.chasse();
      animal.regarde();




      Calin loupCalin = loup;
      loupCalin.faireUnCalin();

}
```

Interface 4/5

- Une interface permet de déclarer des méthodes et aussi des constantes :
  - Les méthodes implicitement sont abstraites et publiques
  - Les attributs sont implicitement finaux, statiques et publiques

- Depuis Java 8 , une interface peux définir des méthodes default
  - n’ont pas besoin d’être implémentées par les classes existantes

- Depuis Java 9 , une interface peux définir des méthodes private
  - peuvent être utilisées pour implémenter des logiques internes de l’interface

© Boris SAUVAGE - 2023

Interface 5/5

- Exemple d’interface :

```
public interface FileManager {

    public static final String FILE_EXTENSION = ".txt";

    public void save(String fileName, String content);

    private String generatePath(String fileName) {
        return "/path/to/directory/" + fileName + FILE_EXTENSION;
    }

    default void saveWithFilePath(String fileName, String content) {
        String path = generatePath(fileName);
        save(path, content);
    }

}
```

© Boris SAUVAGE - 2023

Développement JAVA

MOT CLÉ FINAL

Empêcher la redéfinition de valeur

- Une variable déclarée **final** ne peut changer de valeur
  - Le modificateur **final** doit être ajouté devant le type

```
public static void main(String[] args) {

 final String personne = "Boris Sauvage";

personne = "John Doe";
```

Empêcher l'héritage

- Une classe déclarée **final** ne peut pas être héritée/dérivée
  - Le modificateur **final** doit être ajouté

```
public final class Animal {
    ...
}
```

```
public class Chat extends Animal {


}
```

Empêcher la redéfinition

- Une classe peut protéger une méthode de sa redéfinition dans les sous-classes
  - Le modificateur **final** doit être ajouté dans la signature de la méthode

```
public class Animal {

    void final exemple(){ … }
}
```

```
public class Chat extends Animal {

    @Override
    void exemple(){ … }
}
```

Développement JAVA

LA CLASSE OBJECT

Héritage et classe Object

- Toutes les classes java héritent implicitement de la classe Object
  - La classe Object est l'ancêtre de toutes les classes

```
public class Object {
    …
}
```

```
public class Animal extends Object {


}
```

Classe Object et méthodes

- Cette classe Object possède des méthodes communes à toutes les classes
  - Toute les classes peuvent redéfinir ces méthodes car elles héritent de la classe Object

- Les méthodes de la classe Object sont :
  - getClass()
  - equals()
  - hashCode()
  - toString()
  - finalize()
  - clone()

```
public class Object {

        ...
}
```

Méthode getClass

- Permet d'accéder à l'objet représentant la classe de l'instance
  - Utilisée dans des usages avancées impliquant la réflexivité

```
public static void main(String[] args) {

    Task t1 = new Task(1L, "Tondre");

    Class<Task> taskClass = (Class<Task>) t1.getClass();
    System.out.println(taskClass.isInstance(t1));
    System.out.println(taskClass.isInterface());
    System.out.println(taskClass.getDeclaredFields().length);
    System.out.println(taskClass.getConstructors().length);
}
```

```
true
false
2
2
```

```
public class Task extends Object {

    private Long id;
    private String description;

    public Task(Long id, String description) {
        this.id = id;
        this.description = description;
    }

    public Task(String description) {
        this.description = description;
    }

}
```

Méthode equals 1/2

- La comparaison d'objets se fait grâce à la **méthode equals** héritée Object

```
public static void main(String[] args) {
   Task t1 = new Task(1L, "Tondre");
   Task t2 = new Task(2L, "Ramasser les feuilles");

   if(t1.equals(t2)){
      System.out.println("t1 est identique à t2");
   }

}
```

- L'implémentation par défaut de **equals** compare les références entre elles

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Méthode equals 2/2

- La redéfinition de **equals** permet de définir une égalité sur les attributs de notre choix

```
public class Task extends Object {

    private Long id;
    private String description;
    private LocalDateTime createdAt;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Task task = (Task) o;
        return id.equals(task.id);
    }
```

© Boris SAUVAGE - 2023

Méthode hashCode() 1/3

- La méthode **hashCode()** permet de renvoyer la valeur de hachage de l'objet

```
public static void main(String[] args) {
    Task t1 = new Task(1L, "Tondre");
    Task t2 = new Task(2L, "Ramasser les feuilles");

    System.out.println(t1.hashCode());
    System.out.println(t2.hashCode());
}
```

```
317574433
885284298
```

- L'implémentation par défaut renvoie la référence de l'objet sous la forme d'un **int**

```
public native int hashCode();
```

  – Cette méthode est **native** car c'est la JVM qui peut obtenir l'adresse mémoire de l'objet

Méthode hashCode() 2/3

- La redéfinition de **hashCode()** est nécessaire par soucis de performance
  - il est nécessaire de redéfinir la **hashCode** si **equals** est redéfinie

- Deux objets égaux doivent avoir le même hash code
  - La définition de **hashCode** et **equals** doivent utiliser les même attributs de la classe

- **hascode()** est utilisée par les collections pour optimiser le classement et la recherche

<u>Recherche par clé d'un élément dans une HashMap :</u>

Les collections utilisent les valeurs **hashCode()** pour regrouper les éléments par plage de valeurs

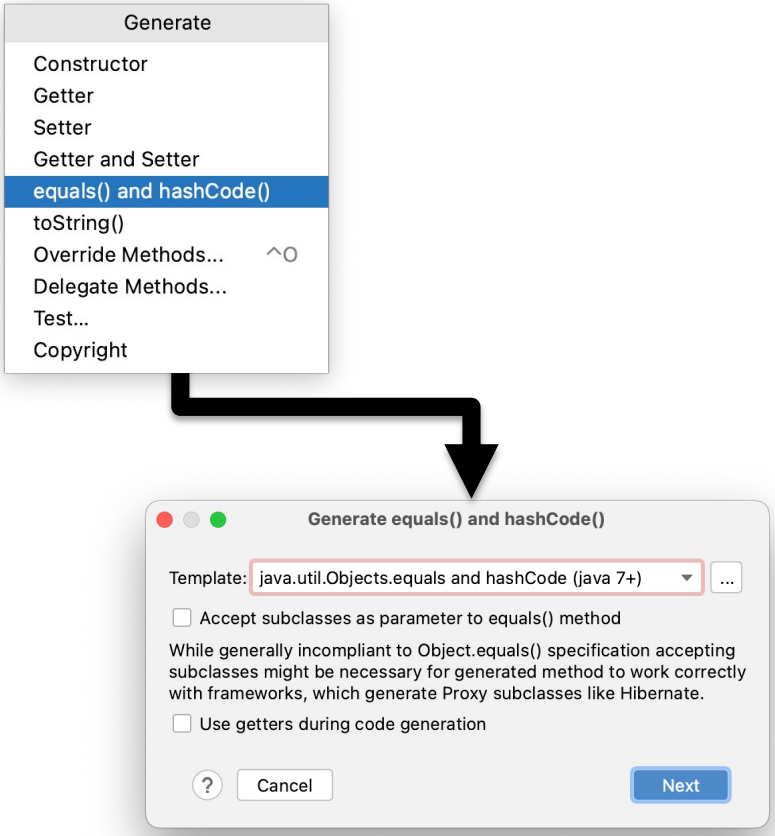
Plutôt que de parcourir toute la liste, la recherche se fera dans le groupe d'éléments concordant avec le hashCode

© Boris SAUVAGE - 2023

Méthode hashCode() 3/3

- L'IDE fourni des fonctionnalités pour générer **hashCode** et **equals** à partir des attributs

```java
public class Task extends Object {

    private Long id;
    private String description;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Task task = (Task) o;
        return id.equals(task.id) && description.equals(task.description);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, description);
    }
```



Méthode toString 1/2

- Permet d'obtenir une représentation d'un objet sous forme de chaîne de caractères
  - Utilisée durant le débogage et la production de log

```
public static void main(String[] args) {

    Task t1 = new Task(1L, "Tondre");
    Task t2 = new Task(2L, "Ramasser les feuilles");

    System.out.println(t1.toString());
    System.out.println(t2.toString());
}
```

```
fr.training.Task@12edcd21
fr.training.Task@34c45dca
```

- L'implémentation par défaut retourne le type de l'objet, suivi de @, suivi du hashcode

```
public String toString() {
   return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Méthode toString 2/2

- La redéfinition de **toString()** permet de définir la manière d'afficher notre objet

```
public class Task extends Object {

   private Long id;
   private String description;

   @Override
   public String toString() {
      return "Task{ id=" + id + ", description='" + description + "}";
   }
```

```
   Task t1 = new Task(1L, "Tondre");
   Task t2 = new Task(2L, "Ramasser ..");

   System.out.println(t1.toString());
   System.out.println(t2);
```

```
Task {id=1, description='Tondre'}
Task {id=2, description='Ramasser les feuilles'}
```

Méthode finalize 1/2

- La redéfinition de **finalize()** permet d'exécuter du code à la destruction de l'objet
  - Le ramasse-miettes exécute **finalize()** avant de libérer l'espace mémoire de l'objet
- **finalize() est déprécié**
- La JVM ne donne aucune garantie sur le moment ou **finalize()** sera appelée
  - Elle peut ne jamais être appelée si l'application se termine avant le passage du Garbage Collector
- L'implémentation par défaut de **finalize()** est vide

```
protected void finalize() throws Throwable { }
```

© Boris SAUVAGE - 2023

Méthode finalize 2/2

- Lorsqu'une variable est affectée à *null*, le Garbage Collector nettoie l'espace mémoire
  - Pour l'exemple, nous allons forcer le passage du ramasse miette avec **System.gc()**

```java
public static void main(String[] args) {
    Task t1 = new Task(1L, "Tondre");
    t1=null;

    System.gc();

    for (int i = 0; i < 1000; ++i) {
        System.out.print('.');
    }

}
```

```java
public class Task extends Object {

    private Long id;
    private String description;

    @Override
    protected void finalize() throws Throwable {
        System.out.print("finalize() of Task id=" + this.id);
    }

}
```

```
.............finalize() of Task id=1.....................................................................................
```

Méthode clone 1/2

- **clone()** est utilisée pour cloner une instance : obtenir une copie d'un objet
  - Est déclarée protected car toutes les classes ne désirent pas permettre de cloner une instance

- L'implémentation par défaut propage une exception **CloneNotSupportedException**

```
protected native Object clone() throws CloneNotSupportedException;
```

- Pour redéfinir **clone()**, l'interface **Cloneable** doit obligatoirement être implémentée
  - Sinon, une exception de type CloneNotSupportedException sera levée

© Boris SAUVAGE - 2023

Méthode clone 2/2

- Exemple d'implémentation de la méthode **clone()**

```java
public class Task extends Object implements Cloneable {

    @Override
    protected Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException cnse) {
            cnse.printStackTrace(System.err);
        }
        return o;
    }
}
```

```java
public static void main(String[] args) {
    Task t1 = new Task(1L, "Tondre");
    System.out.println(t1);

    Task t2 = (Task) t1.clone();
    System.out.println(t2);
}
```

```
Task {id=1, description='Tondre'}
Task {id=1, description='Tondre'}
```

© Boris SAUVAGE - 2023

Exercice

- Réaliser le projet Java Town

Développement JAVA

TYPES GÉNÉRIQUES

Types génériques 1/3

- Il est possible de créer des méthodes et des classes qui utilisent des types génériques
  - Les types sont des paramètres résolus au moment de l'invocation ou instanciation
  - Ces méthodes et ces classes deviennent donc génériques

- L' API Java Standard fournit un ensemble de classes génériques ou non
  - Il est également possible de créer ses propres classes génériques

- **Java.util.ArrayList** permet de gérer une liste d'objets avec des fonctionnalités pratiques

```
ArrayList<String> list = new ArrayList<String>();
list.add("Hey !");      // OK
list.add(100);          // Erreur de compilation
```

© Boris SAUVAGE - 2023

Types génériques 2/3

- Pour raccourcir la syntaxe, il est possible d'utiliser la notation en diamant :

```
ArrayList<String> list = new ArrayList<>();

list.add("Hey !");      // OK
list.add(100);          // Erreur de compilation
```

La notation **<>** est
la notation en diamant

- Par défaut, sans paramétrer la classe ArrayList, le type Objet est utilisé

```
ArrayList list = new ArrayList();

list.add("Hey !");      // OK
list.add(100);          // OK
```

```
Objet o = list.get(0);          // OK
```

Types génériques 2/3

- Le compilateur Java permet d'utiliser plusieurs types génériques :
  - **T** : type générique en général
  - **E** : type générique qui représente un élément
  - **K** : type générique utilisé comme une clé *(Key)*
  - **V** : type générique utilisé comme une valeur *(Value)*
  - **U,V,W** : identifie une suite de types génériques

- Ces types sont utiles pour créer une classe qui sert de conteneur à une autre classe

- **Définition de la classe Optional**
  - http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/2aae624bb833/src/share/classes/java/util/Optional.java

© Boris SAUVAGE - 2023

Classe générique

- Une classe peut être générique et supporter un ou plusieurs types paramétrés

```
public class Pair<U, V> {

    private U valeurGauche;
    private V valeurDroite;

    public Pair(U gauche, V droite) {
        this.valeurGauche = gauche;
        this.valeurDroite = droite;
    }
```

```
Gant gantGauche = new Gant();
Gant gantDroite = new Gant();
Pair gants = new Pair<>(gantGauche, gantDroite);
```

```
Chaussette chaussetteG = new Chaussette();
Chaussette chaussetteD = new Chaussette();
Pair chaussettes = new Pair<>(chaussetteG, chaussetteD);
```

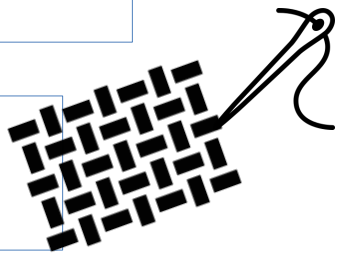
La capture d'un type générique 1/2

- Une capture permet de déterminer les classes acceptables pour un type générique

```
public class Pair<U extends Bonneterie, V extends Bonneterie> {

    private U valeurGauche;
    private V valeurDroite;
}
```

Est une

```
public class Bonneterie {
}
```



```
Chaussette chaussetteGauche = new Chaussette();
Chaussette chaussetteDroite = new Chaussette();
Pair chaussettes = new Pair<>(chaussetteGauche, chaussetteDroite);
```

© Boris SAUVAGE - 2023

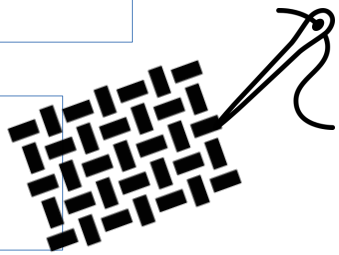
La capture d'un type générique 2/2

- La capture limite les types paramétrés U,V au type **Bonneterie** ou un type qui en hérite

```
public class Pair<U extends Bonneterie, V extends Bonneterie> {

    private U valeurGauche;
    private V valeurDroite;
}
```

Est PAS une

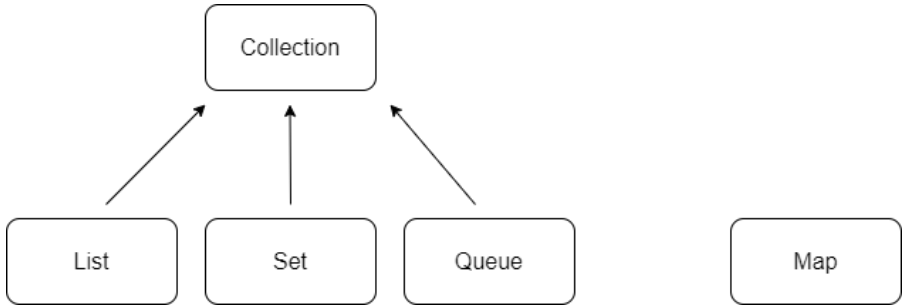
```
public class Bonneterie {
}
```



```
Gant gantGauche = new Gant();
Gant gantDroite = new Gant();
Pair gants = new Pair<>(gantGauche, gantDroite);
```

Développement JAVA

LES COLLECTIONS



Limites des tableaux

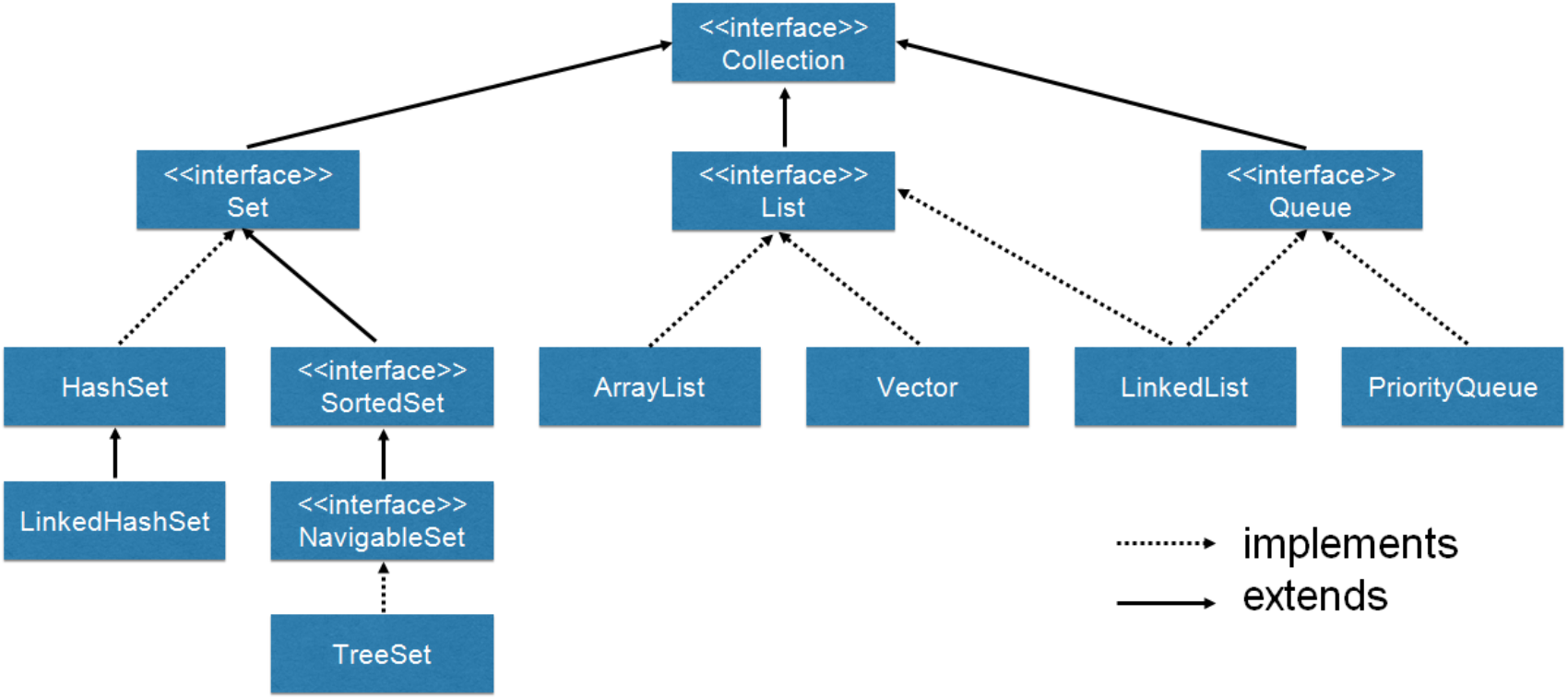
- Les tableaux peuvent être utilisés pour des opérations simples
  - La taille est définie lors de sa création
  - Le décalage des éléments lors d'une suppression n'est pas automatique
  - Les éléments sont obligatoirement indexés par des entiers

- Les collections sont l'évolution logique des tableaux
  - Disponible nativement dans Java
  - Facilitent la gestion d'ensembles d'éléments
  - Propose plusieurs formes d'implémentations de collection

© Boris SAUVAGE - 2023

Les collections : Introduction 1/4

- Les collections sont optimisées en fonction de besoins précis
  - Utilisent les types génériques



Les collections : Introduction 2/4

Les interfaces de base des collections sont : **Collection**, **List**, **Set** et **Map**

- **Collection** est l'interface racine de toutes les collections
  - (représente un groupe d'objets)

- **List** est une collection ordonnée qui peut contenir des doublons
- **Set** est une collection qui ne permet pas les doublons
- **Queue** est une collection qui représente une file d’attente ou pile
- **Map** est une structure de données qui associe des clés à des valeurs

© Boris SAUVAGE - 2023

Les collections : Introduction 3/4

Les classes d'implémentation des collections sont :

- **ArrayList** et **LinkedList** pour l'interface **List**
- **HashSet** et **TreeSet** pour l'interface **Set**
- **LinkedList, ArrayDeque** et **PriorityQueue** pour l'interface **Queue**
- **HashMap** et **TreeMap** pour l'interface **Map**

© Boris SAUVAGE - 2023

Les collections : Introduction 4/4

List (ordonnée - avec doublons)

```
List<String> words = new ArrayList<>();
words.add("Hello");
words.add("World");
words.add("!");
System.out.println(words);
```

[Hello, World, !]

Set (non ordonné- sans doublons)

```
Set<String> characteres = new HashSet<>();
characteres.add("H");
characteres.add("e");
characteres.add("l");
characteres.add("l");
characteres.add("o");
System.out.println(characteres);
```

[H, e, l, o]

Map (dictionnaire clé/valeur – clé unique)

```
String lorem = "Hello World";
Map<Character, Integer> myMap = new HashMap<>();
for (char c : lorem.toCharArray()) {
   myMap.merge(c, 1, Integer::sum);
}
System.out.println(myMap);
```

{ =1, r=1, d=1, e=1, W=1, H=1, l=3, o=2}

© Boris SAUVAGE - 2023

L’interface List 1/3

- Ensemble de données ordonné (tableau)
- Peut contenir des doublons
- Permet l'ajout et la suppression d'éléments
- **ArrayList** est l’implémentation la plus utilisée
  - Performante pour la lecture aléatoire des éléments et l’ajout/suppression à la fin
- **LinkedList** est l’implémentation d’une liste chaînée
  - Performante pour l’ajout et la suppression d’un élément à n’importe quelle position

© Boris SAUVAGE - 2023

L'interface List 2/3

- Ajouter un élément est possible avec la méthode **add()**

```
List<String> data = new ArrayList<>();
data.add("Ma");
data.add("liste");
data.add("dynamique");
```

- Supprimer un élément est possible avec la méthode **remove()**

```
List<User> list = List.of( new User("John", "DOE"), new User("Jane", "DAY") );
User u3 = new User("foo", "bar");
list.add(u3);

list.remove(u3);          // suppression de la référence vers u3
list.remove(0);           // suppression de l'élément en position
```

© Boris SAUVAGE - 2023

L'interface List 3/3

Il existe de nombreuses méthodes sur l'interface List :

- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html

- **size()**
- **isEmpty()**
- **sort()**
- **clear()**
- **contains()**
- **…**

```
List<User> list = List.of(
            new User("John", "DOE"),
            new User("Jane", "DAY");

System.out.println(list.size());
System.out.println(list.isEmpty());
list.clear();
System.out.println(list.isEmpty());
```

```
2
false
true
```

Parcourir les données 1/3

- Il est possible de parcourir les éléments d'une liste à l'aide d'une boucle for :

```
List<String> countries = Arrays.asList("France", "Belgique", "Suisse");

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

- Il est possible d'utiliser la boucle foreach :
  - Préférable vis-à-vis de sa syntaxe

```
for (String country : countries) {
   System.out.println(country);
}
```

Parcourir les données 2/3

- Depuis Java 8 , l'API Stream proposent la fonction forEach() :

```
countries.stream().forEach((c) -> System.out.println(c));

countries.forEach(System.out::println);
```

- Offre une syntaxe plus concise

- Requiert la version Java 8

© Boris SAUVAGE - 2023

Parcourir les données 3/3

- Pour parcourir une liste et la modifier, Il existe également un itérateur :
  - Permet de supprimer un élément durant la lecture de la liste

```
Iterator<String> countriesIterator = countries.iterator();
while(countriesIterator.hasNext()) {
   System.out.println(countriesIterator.next());
}

ListIterator<String> listIterator = countries.listIterator();
while(listIterator.hasNext()) {
   System.out.println(listIterator.next());
}
```

- **ListIterator** permet de parcourir les éléments dans les deux sens

© Boris SAUVAGE - 2023

L’interface Set 1/3

- Ensemble de données non ordonné (ordre non garanti)
- **Ne peut pas** contenir des doublons
- Permet l'ajout et la suppression d'éléments
- **HashSet** est l’implémentation la plus utilisée
  - Performante pour la lecture aléatoire des éléments et l’ajout/suppression à la fin
- **TreeSet** est l’implémentation d’une **HashSet** triée
  - Performante moins intéressantes que HashSet

© Boris SAUVAGE - 2023

L'interface Set 2/3

- Ajouter un élément est possible avec la méthode **add()**

```
Set<String> data = new HashSet<>();
data.add("XN-455-BW");
data.add("HQ-123-UU");
data.add("AB-987-YT");
data.add("XN-455-BW");
```

- Supprimer un élément est possible avec la méthode **remove()**

```
Set<User> set = Set.of( new User("John", "DOE"), new User("Jane", "DAY") );
User u3 = new User("foo", "bar");
set.add(u3);

set.remove(u3);          // suppression de la référence vers u3
set.remove(0);           // suppression de l'élément en position
```

© Boris SAUVAGE - 2023

L'interface Set 3/3

Il existe de nombreuses méthodes sur l'interface Set :

- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashSet.html

- **size()**
- **isEmpty()**
- **sort()**
- **clear()**
- **contains()**
- **…**

```
Set<User> set = Set.of(
            new User("John", "DOE"),
            new User("Jane", "DAY");

System.out.println(set.size());
System.out.println(set.isEmpty());
set.clear();
System.out.println(set.isEmpty());
```

```
2
false
true
```

L’interface Queue/Deque 1/3

- Représente une file d’attente ou bien une pile
- Principe du FIFO (First In, First Out) ou bien du LIFO (Last In, First Out)
- **ArrayDeque** est l’implémentation la plus utilisée
  - Performante pour l’ajout et la suppression aux deux extrémités
- **LinkedList** est l’implémentation d’une liste chaînée
  - Performante pour l’ajout et la suppression d’un élément à n’importe quelle position
- **PriorityQueue** est l’implémentation pour maintenir un ordre basé sur une priorité
  - Ordre basé sur la priorité des éléments, influencé grâce à un comparateur

© Boris SAUVAGE - 2023

L'interface Queue/Deque 2/3

- Création d'une file d'attente (FIFO) :

```
Queue<String> folders = new ArrayDeque<>();
folders.offer("Dossier A");
folders.offer("Dossier B");
folders.offer("Dossier C");

while (! folders.isEmpty() ) {
   System.out.println( folders.poll() );
}
```

Dossier A
Dossier B
Dossier C

- La méthode **offer()** ajoute un élément à la file

- La méthode **poll()** retire et renvoie le **premier** élément de la file

L'interface Queue/Deque 3/3

- Création d'une pile (LIFO) :

```
DeQueue<String> folders = new ArrayDeque<>();
folders.offer("Dossier A");
folders.offer("Dossier B");
folders.offer("Dossier C");

while (! folders.isEmpty() ) {
   System.out.println( folders.pollLast() );
}
```

Dossier C
Dossier B
Dossier A

- La méthode **offer()** ajoute un élément à la file

- La méthode **pollLast()** retire et renvoie le **dernier** élément de la file

L’interface Map 1/3

- Ne fait pas parti des Collections Java
- Peut contenir des doublons
- Dictionnaire de données non ordonné (ordre des clés non garanti)

- **HashMap** est l’implémentation la plus utilisée
- **TreeMap** est l’implémentation d’une **HashMap** triée
  - Ordre d’itération garanti selon l’ordre naturel des clés ou selon un comparateur précisé
- **LinkedHashMap** est l’implémentation pour conserver l’ordre d’insertion

© Boris SAUVAGE - 2023

L’interface Map 2/3

- Ajouter un élément est possible avec la méthode **put()**

```
Map<String, Car> data = new HashMap<>();
data.put("XN-455-BW", new Car("BMW", Color.RED));
data.put("HQ-123-UU", new Car("Tesla", Color.WHITE));
data.put("AB-987-YT", new Car(" Lamborghini ", Color. YELLOW));
data.put("XN-455-BW", new Car("BMW", Color.RED));
```

- Récupérer un élément avec la méthode **get()**

```
Car car = data.get("XN-455-BW");
```

- Supprimer un élément est possible avec la méthode **remove()**

```
data.remove("XN-455-BW");    // suppression de l’élèment par sa clé
```

© Boris SAUVAGE - 2023

L'interface Map 3/3

Il existe de nombreuses méthodes sur l'interface Map :

- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Map.html

- **size()**
- **get()**
- **isEmpty()**
- **clear()**
- **contains()**
- **entryset()**
- **keyset()**

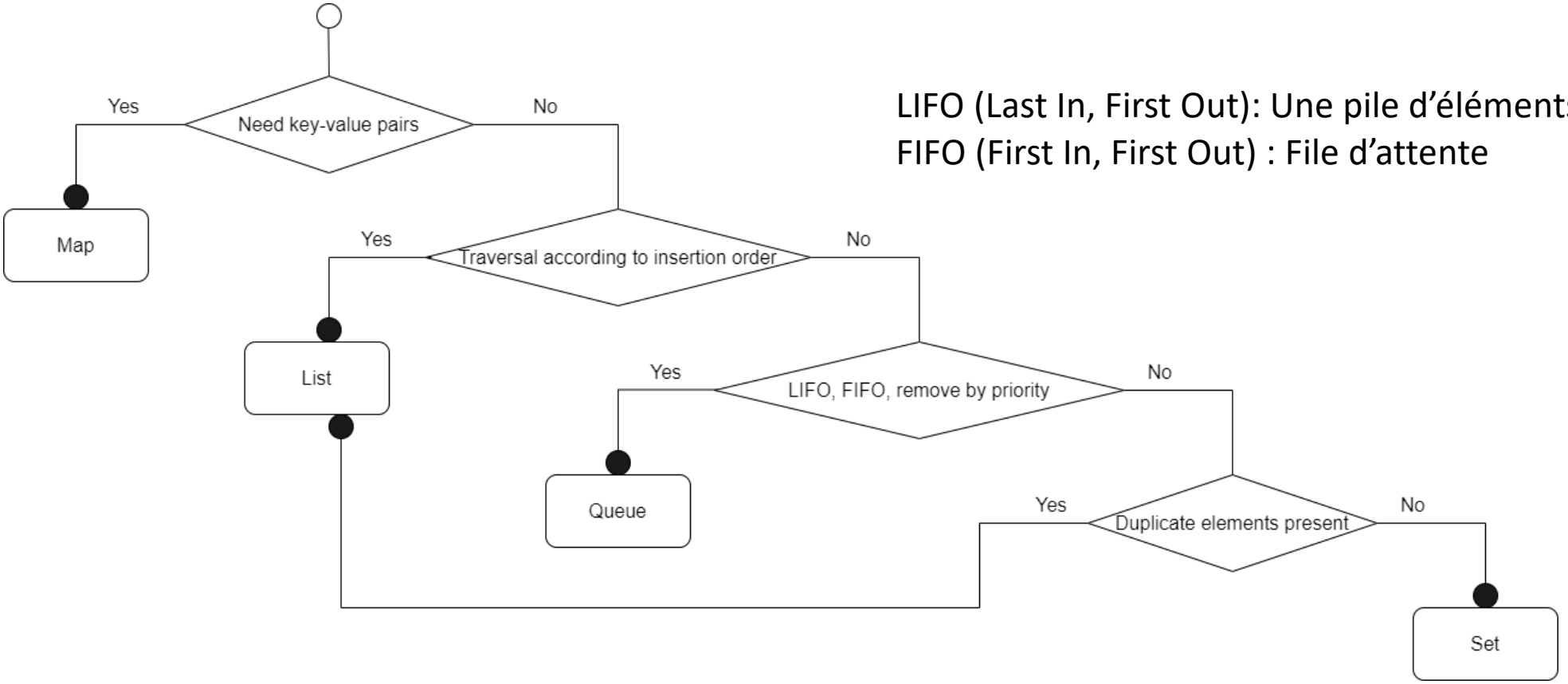
```
for (Map.Entry<Character, Car> entry : myMap.entrySet()) {
   System.out.println(entry.getKey() + ":" + entry.getValue());
}
```

```
for (String key : myMap.keySet()) {
   System.out.println(key + ":" + myMap.get(key));
}
```

```
for (Integer value : myMap.values()) {
   System.out.println(value);
}
```

Choisir une collections

Afin de choisir la bonne collection, vous pouvez utiliser le diagramme ci-dessous :



LIFO (Last In, First Out): Une pile d'éléments
FIFO (First In, First Out) : File d'attente

Développement JAVA

LES ENUMÉRATIONS

Les énumérations 1/3

- Permet de représenter les listes finies d'éléments
- Les éléments d'une énumération sont des constantes
- Défini un type avec un nombre fini de valeurs
- Il existe déjà des énumérations dans Java

```
public enum DayOfWeek {

    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY

}
```

© Boris SAUVAGE - 2023

Les énumérations 2/3

- Une énumération est une classe qui étends la classe **Enum**
  - La classe **Enum** étend de la classe Objet

- Les méthodes d'Objet sont disponibles : equals(), toString(), etc

```
public static void main(String[] args) {
    System.out.println("Jour : " + Days.FRIDAY);
    System.out.println(Days.MONDAY == Days.MONDAY);
}
```

© Boris SAUVAGE - 2023

Les énumérations 3/3

- **valueOf()** retourne la valeur énumérée à partir de sa chaîne de caractères

```
public static void main(String[] args) {
   System.out.println(DayOfWeek.valueOf("MONDAY"));
}
```

- Méthode **values()**

- Méthode **ordinal()**

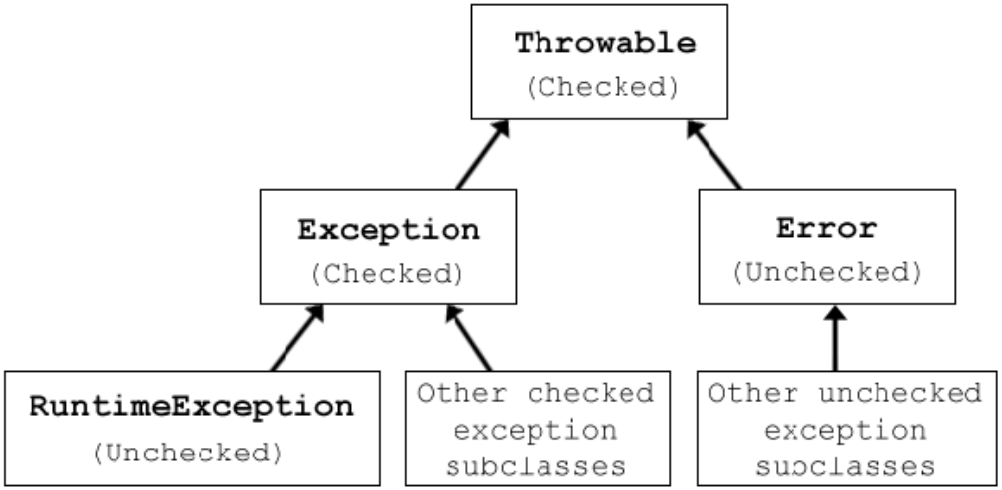
© Boris SAUVAGE - 2023

Exercice

- Réaliser les exercices List, Set et Map

Développement JAVA

LES EXCEPTIONS



Une Exception 1/2

- Événement anormal qui se produit lors de l'exécution d'un programme Java
- Interrompt le flux normal d'exécution
- Peut être causée par plusieurs raisons différentes :
  - erreurs de programmation
  - erreurs d'exécution
  - erreurs liées à l'environnement d'exécution

© Boris SAUVAGE - 2023

Une Exception 2/2

- Exemples d'exception lors de l'accès en dehors d'une liste :

```
List<String> words = List.of("Hello", "World", "!");
words.get(5);
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 5 out of bounds for length 3…
```

- Exception lors d'une tentative incorrecte pour supprimer sans itérateur :

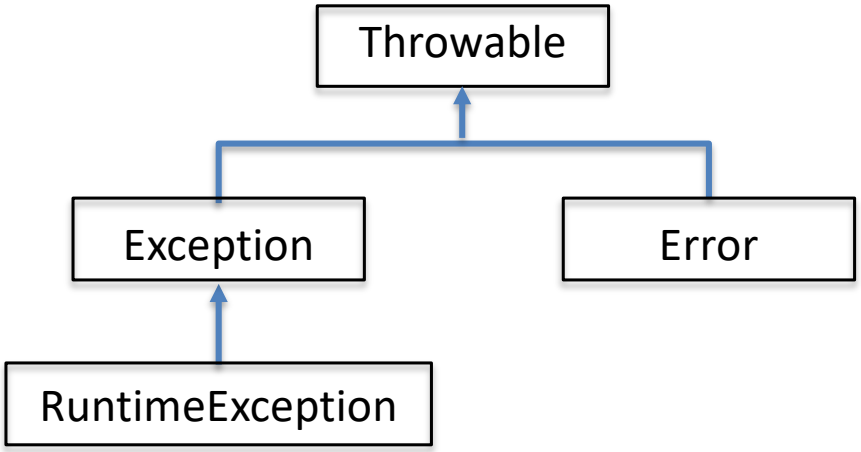
```
List<String> words = new ArrayList<>(List.of("Hello", "World", "!"));
for (String w : words) {
   if (words.size() > 1) words.remove(w);
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

Hiérachie des Exceptions

Les exceptions sont organisées sous forme d'une hiérarchie de classes

- **Exception** est utilisée pour les erreurs d'exécution gérables
- **Error** est utilisée pour les erreurs graves et irrécupérables
  - Généralement causées par des problèmes système



© Boris SAUVAGE - 2023

Capturer une exception 1/3

- Le bloc **try-catch** est utilisé pour exceptions en java

- Encadre le code qui peut générer une exception

```
try {
   Class.forName("fr.sauvageb.Class");   // le code qui peut générer une exception
} catch (ClassNotFoundException e) {
   e.printStackTrace();                  // le code qui traite l'exception
}
```

- Permet de continuer l'exécution du programme même en cas d'exception

© Boris SAUVAGE - 2023

Capturer une exception 2/3

- Le bloc **try-catch** peut gérer plusieurs Exceptions :

```
try {
   // Code pouvant générer une exception
   FileInputStream fis = new FileInputStream("fichier.txt");
   // ...
} catch (FileNotFoundException e) {
   System.out.println("Erreur : Le fichier n'a pas été trouvé.");
} catch (NullPointerException e) {
   System.out.println("Erreur : Pointeur null rencontré.");
} catch (ArithmeticException e) {
   System.out.println("Erreur : Division par zéro.");
} catch (Exception e) {
   System.out.println("Erreur générale : " + e.getMessage());
}
```

© Boris SAUVAGE - 2023

Capturer une exception 3/3

- Le bloc **try-catch** peut être combiné à une clause **finally** :
  - Utilisé pour exécuter du code dans tous les cas (exception levée ou non)

```
try {
    FileInputStream fis = new FileInputStream("fichier.txt");
   // ...
} catch (FileNotFoundException e) {
   System.out.println("Erreur : Le fichier n'a pas été trouvé.");
} catch (Exception e) {
   System.out.println("Erreur générale : " + e.getMessage());
} finally {
   System.out.println("Bloc finally : Ce code sera exécuté qu'il y ait une exception ou non.");
}
```

© Boris SAUVAGE - 2023

Mot clé Throws 1/2

- Indique qu'une méthode peut générer certaines exceptions
  - Utilisé dans les signatures de méthodes
- Signifie que la méthode ne gère pas elle-même les exceptions
  - Propage les exceptions à son appelant ou à ses appelants
- L'appelant d'une méthode contenant ***throws*** devra gérer celles-ci
  - Utilisation du bloc try/catch ou bien d'une clause throws
- Les exceptions vérifiées sont uniquement déclarées dans **throws**
  - Les exceptions non vérifiées n'ont pas besoin d'être déclarées

© Boris SAUVAGE - 2023

Mot clé Throws 2/2

- Exemple d'une méthode utilisant **throws** :

```
void myMethod() throws TypeException1, TypeException2 {
   // Instructions
}
```

- Exemple du traitement de cette méthode appelée :

```
try {
   myMethod();   // previous method is called
} catch (TypeException1 e1) {
   System.out.println("Exception de TypeException1 : " + e1.getMessage());
} catch (TypeException2 e2) {
   System.out.println("Exception de TypeException2 : " + e2.getMessage());
}
```

© Boris SAUVAGE - 2023

Les types d’exceptions 1/4

Il existe deux types d'exceptions :

- les exceptions vérifiées **(checked exceptions)**
  - doivent être gérées explicitement par le code
- les exceptions non vérifiées **(unchecked exceptions)**
  - peuvent être gérées ou non par le code (optionnel)

© Boris SAUVAGE - 2023

Les types d’exceptions 2/4

Les exceptions vérifiées **(checked exceptions)**

- Sont vérifiées lors de la compilation
- Exemples d'exceptions vérifiées : IOException, SQLException, ClassNotFoundException
- Doivent être entourées par un bloc try-catch ou déclarées dans la clause throws

```
try {
    Class.forName("fr.sauvageb.MyClass");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

```
private static void loadLibrary() throws ClassNotFoundException {
    Class.forName("fr.sauvageb.MyClass");
}
```

© Boris SAUVAGE - 2023

Les types d’exceptions 3/4

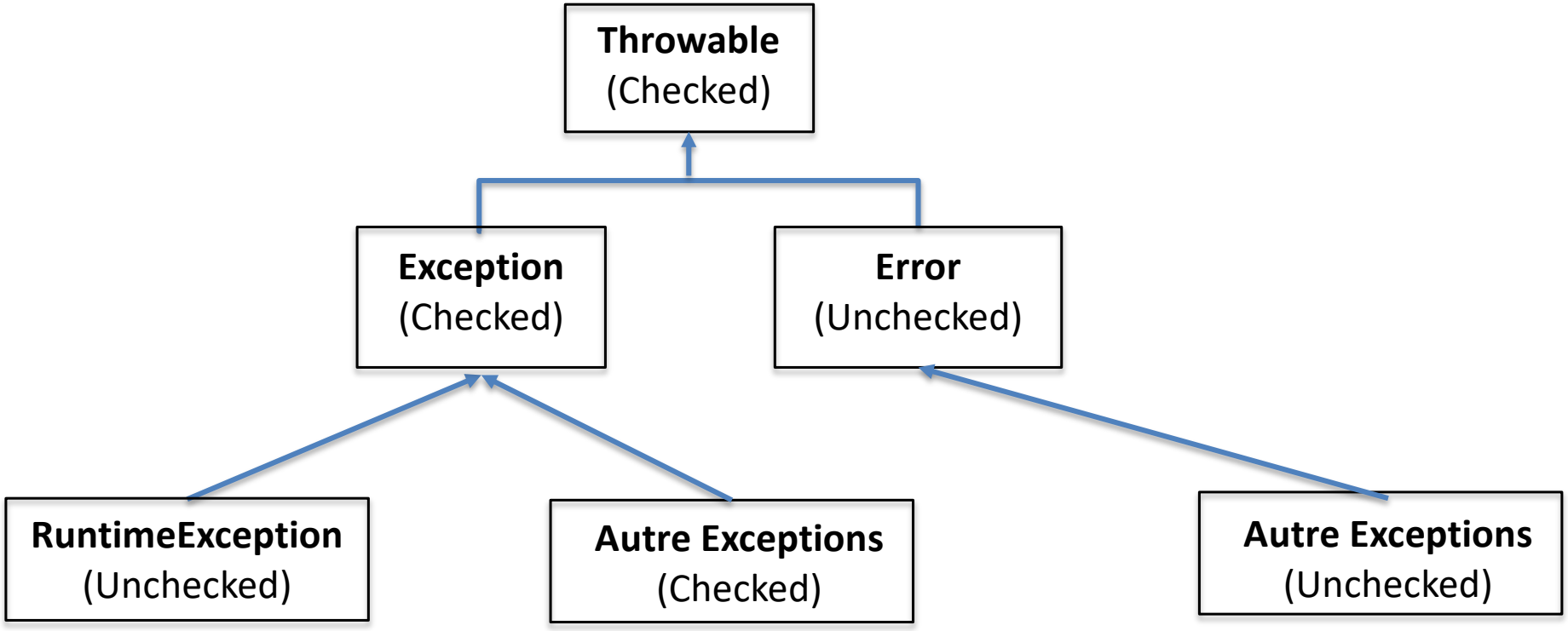
Les exceptions non vérifiées **(unchecked exceptions)**

- Peuvent être gérées ou non par le code (optionnel)
- Ne sont pas vérifiées lors de la compilation
- Ne nécessitent pas de block try-catch ou de clause throws
- Exemples d'exceptions non vérifiées : NullPointerException, ArrayIndexOutOfBoundsException, etc.

© Boris SAUVAGE - 2023

Les types d’exceptions 4/4

Il existe 2 types d'exceptions :



© Boris SAUVAGE - 2023

Exception personnalisée

- Il est possible de créer ses propres exceptions

- Il faut créer une classe héritant de **java.lang.Exception** ou de ses sous-classes
  - Cette classe doit avoir un constructeur avec un message d'erreur en paramètre

```
public class MyException extends Exception {

   public MyException(String message) {
      super(message);
   }
}
```

- Pour une exception non vérifiée, il est possible d'hériter de la classe RuntimeException

Lever une exception

- Il est possible de lancer une exception manuellement depuis un bloc de code
  - Permet de signaler des erreurs/situations anormales dans le flux d'exécution

```
public int divide(int a, int b) throws ArithmeticException {
   if (b == 0) {
      throw new ArithmeticException("Error : Division by zero.");
   }
   return a / b;
}
```

- Cette exception hérite de RuntimeException mais il est possible de la gérer

```
try {
   int result = divide(10, 0);
} catch (ArithmeticException e) {
   System.out.println("Error : " + e.getMessage());
}
```

Exercice

- Réaliser les exercices sur les Exceptions

Développement JAVA

TRY-WITH-RESOURCES

Try-with-resources

- Permet de simplifier la gestion des ressources depuis Java 7
- Garantit que les ressources sont fermées de manière automatique
- Syntaxe simplifiée basée sur le bloc try-catch classique :

Avant Java 7

```
Resource resource = new Resource();
// …
resource.close():
```

Depuis Java 7

```
try (Resource resource = new Resource()) {
    // …
} // Fermeture automatique de la ressource
```

- La ressource doit implémenter l'interface **AutoCloseable** ou **Closeable**
  - Sera automatiquement fermée à la fin du bloc try, qu'il y ait une exception ou non

© Boris SAUVAGE - 2023

Développement JAVA

POLYMORPHISME DÉTAILLÉ

Polymorphisme

- Le polymorphisme se traduit par ***Qui peut rendre plusieurs formes***

- Il existe 3 types :

Ad hoc	➡	Surcharge
Héritage	➡	Spécialisation/Généralisation
Paramétrique	➡	Généricité

© Boris SAUVAGE - 2023

Polymorphisme Ad hoc 1/2

- Connu sous le nom de **polymorphisme par surcharge**
- Une méthode peut être définie plusieurs fois avec des signatures différentes
  - Avec des types et des nombres d'arguments différents
- Utilise des objets de manière générique en utilisant une référence de type de classe de base
- Appeler une méthode surchargée en fonction des arguments passés en paramètre en cours d'exécution

© Boris SAUVAGE - 2023

Polymorphisme Ad hoc 2/2

- **Exemple d'une surcharge**

```
public class Animal {

public void makeSound() {
        System.out.println("Some animal sound");
    }


public void makeSound(String sound){
        System.out.println(sound);
    }
}
```

```
public static void main(String[] args) {
    Animal animal1 = new Animal();
    animal1.makeSound();

    Animal animal2 = new Animal();
    animal2.makeSound("Woof");
}
```

```
Some animal sound
Woof
```

Polymorphisme Héritage 1/2

- Connu sous le nom de **polymorphisme statique**
- Une méthode est définie pour une classe de base et réécrite par des sous-classes
- Utilise des objets de manière générique en utilisant une référence de type de classe de base
- Souvent utilisé en conjonction avec le polymorphisme Ad Hoc pour offrir une flexibilité maximale dans la conception d'une application

© Boris SAUVAGE - 2023

Polymorphisme Héritage 2/2

- **Exemple d'un héritage**

```java
public class Animal {

public void makeSound() {
        System.out.println("Some animal sound");
    }
}
```

```java
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

```java
public static void main(String[] args) {
    Animal animal1 = new Cat();
    animal1.makeSound();
    Animal animal2 = new Dog();
    animal2.makeSound();
}
```

```java
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof");
    }
}
```

Meow
Woof

Polymorphisme Paramétrique 1/2

- Connu sous le nom de généricité
- Permet de définir plusieurs fonctions de même noms avec des paramètres différents
- Permet le choix automatique de la bonne méthode
  - En fonction du type de donnée passée en paramètre
- Souvent utilisé dans le cadre d'une implémentation offrant des comportements partagés et redondants dont seul les paramètres pourraient changer
  - Exemple du design pattern DAO

© Boris SAUVAGE - 2023

Polymorphisme Paramétrique 2/2

- **Exemple d'une classe générique**

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

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

}
```

```java
public class Main {
    public static void main(String[] args) {

        Box<String> box1 = new Box<>("Bonjour");
        Box<Integer> box2 = new Box<>(42);
        Box<Cat> box3 = new Box<>(new Cat("Réglisse"));

        System.out.println(box1.getContent());
        System.out.println(box2.getContent());
        box3.getContent().makeSound();
    }
}
```

```
Hey
42
Meow
```

Développement JAVA

LES OPTIONALS

La classe Optional

- Apparue avec Java 8, l'API Optional est une alternative à l'utilisation du mot-clé **null**

Aujourd'hui

```
Optional<Car> findByMatriculation(Car[] cars, String matriculation) {
    for (Car car : cars) {
        if (car.getMatriculation().equals(matriculation)) {
            return Optional.of(car);
        }
    }
    return Optional.empty();
}
```

Java 7

```
Car findByMatriculation(Car[] cars, String matriculation) {
    for (Car car : cars) {
        if (car.getMatriculation().equals(matriculation)) {
            return car;
        }
    }
    return null;
}
```

La classe Optional

- **Optional** permet de signaler que notre méthode peut ne pas renvoyer de valeur

.isPresent() : permet de vérifier si une valeur est présente

.get() : permet de récupérer la valeur, après vérification

```
Optional<Car> foundCar = findByMatriculation(cars, "EW-123-NX");

if(foundCar.isPresent() ){
   System.out.println( foundCar.get() );
}
```

- Si une méthode renvoie une liste de résultat, on évitera d'utiliser l'API Optional

© Boris SAUVAGE - 2023

- FIN -