FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Module : Programmation Java

Licence Professionnelle : Développement Informatique

Pr: Y.Ouassit

Objectifs du module

- Décrire les éléments-clé de la plate-forme Java
- Compiler et exécuter une application Java
- Comprendre le paradigme OO et utiliser Java pour le mettre en œuvre
- Comprendre et utiliser les exceptions
- Développer une interface graphique avec AWT et Swing et JavaFX
- Comprendre la gestion des événements
- Lire et écrire sur des fichiers
- Comprendre les bases du multithreading

Descriptif horaire

- 10 heures de cours ;
- 20 heures de TD et TP;
- Évaluation : Examen final

Chapitres

- Introduction au langage Java
- Programmation orientée objet (POO)
- Collections et tableaux
- Gestion des exceptions
- Gestion des fichiers et IO en Java
- Thread et programmation concurrente
- Interfaces graphiques avec Java (Swing , JavaFX,…)
- **Manipulation de bases de données (JDBC)**
- **Initiation au patron de conception en Java**

Selon l'avancement

Sujets non couverts

- Développement des applets
- Servlets
- Java Server Pages (JSP)
- Enterprise Java Beans (EJB)
- Le framework JSF (JavaServer Faces):
- …

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Introduction à Java

Licence : Physique Chimie
Filières : Physique

Pr: Y.Ouassit

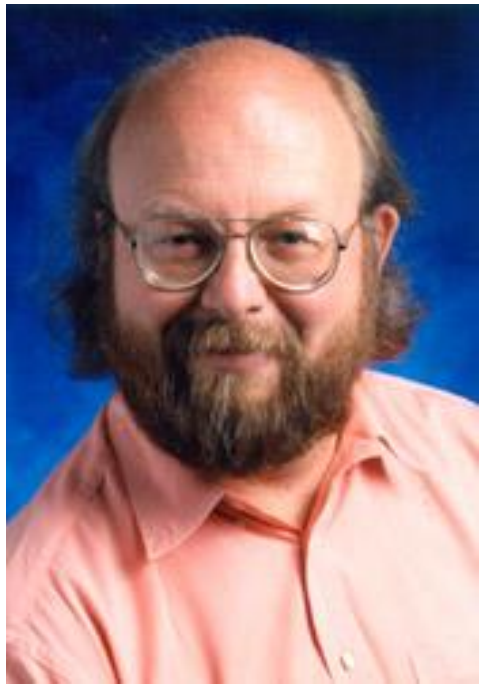
Plan Chapitre 1: Introduction à Java

1. **Introduction au langage Java**
   a. Quelques notions historiques
   b. Qu'est-ce que Java ?
   c. Java comme langage de programmation
   d. La plateforme Java
   e. La Java Virtual Machine
   f. Les interfaces de programmation d'application (API)
   g. Déploiement d'un programme
2. **Syntaxe du langage**
   a. Variables ,constantes et types de données
   b. Opérateurs et expressions
   c. Structures de contrôles
   d. Fonctions et méthodes

Introduction

Histoire de Java (1/7)

1991 - The Birth of Java Project ("Green Project"):



Java a été développé à partir de décembre 1990 par une équipe de **Sun Microsystems** dirigée par James Gosling

Au départ, il s'agissait de développer un langage de programmation pour permettre le dialogue entre de futurs ustensiles domestiques (Green Project)

Or, les langages existants tels que C++ ne sont pas à la hauteur : recompilation dès qu'une nouvelle puce arrive, complexité de programmation pour l'écriture de logiciels fiables...

Slogan : **WRITE ONCE RUN ANYWHERE**

Histoire de Java (2/7)

1992 - First Working Demo of Oak:

L'équipe a développé le premier produit de démonstration utilisant **Oak** pour un petit appareil portable appelé **Star7**.





Histoire de Java (3/7)

1994 - Pivot to the Web:

Avec la montée en puissance du World Wide Web, l'équipe réalise que l'indépendance de plateforme d'Oak pourrait être utile pour le développement web. Oak est adapté pour fonctionner sur les navigateurs web sous forme d'applets, le rendant adapté à du contenu dynamique et interactif sur le web.

1995 - Oak devient Java:

Oak est renommé Java car le nom était déjà déposé. Le nouveau nom est inspiré par le café Java, très apprécié par les développeurs.

Java 1.0 est officiellement lancé lors de la conférence SunWorld. Sun Microsystems rend Java disponible gratuitement, ce qui favorise son adoption rapide.

Netscape Navigator, le navigateur web le plus populaire à l'époque, annonce son support de Java, ce qui lui donne une grande visibilité.

Histoire de Java (4/7)

1997 - Processus de la communauté Java (JCP)

Sun Microsystems lance le Java Community Process (JCP) pour impliquer d'autres entreprises et organisations dans l'évolution de la plateforme Java en proposant des JSR (Java Specification Requests).

JSR 31 : Servlets

JSR 200: JPA

1998 - Java 2 et introduction des éditions (J2SE, J2EE, J2ME)

La sortie de Java 2 (J2SE 1.2) marque une mise à jour importante avec l'introduction de Swing, des Java Foundation Classes (JFC) et des JavaBeans.

Histoire de Java (5/7)

2004 - Java 5 (J2SE 5.0 / Java SE 5.0):

Java 5 introduit des améliorations majeures, comme :

- **Génériques** pour plus de sécurité de type.
- **Annotations** pour simplifier la configuration.
- **Boucle for-each**, **autoboxing/unboxing**, **varargs** et l'**API de concurrence**.

La dénomination est simplifiée, abandonnant le "2", et **J2SE** devient **Java SE** (Edition Standard).

2006 - Java devient open source:

Sun Microsystems annonce que Java sera rendu open source sous la licence GNU General Public License (GPL).

Le projet OpenJDK est lancé, qui deviendra la référence pour l'implémentation de Java.

Histoire de Java (6/7)

2010 - Oracle rachète Sun Microsystems:

Oracle finalise le rachat de **Sun Microsystems**, prenant le contrôle de Java et de son développement.

2011 - Java SE 7 (Première version majeure d'Oracle):

Parmi les nouvelles fonctionnalités, on trouve **try-with-resources** pour la gestion automatique des ressources, l'**opérateur diamant**, et le **framework Fork/Join** pour le traitement parallèle.

2014 - Java 8 et l'introduction des expressions Lambda

Java 8 est lancé, introduisant les **expressions lambda** et l'**API des Streams**, révolutionnant la gestion des collections et la programmation fonctionnelle en Java.

Histoire de Java (7/7)

2017 - Java 9 et le système de modules (Project Jigsaw):

Java 9 introduit le **Java Platform Module System (JPMS)**, également connu sous le nom de **Project Jigsaw**, permettant la modularisation de la JDK et des applications Java.

2018 - Adoption d'un nouveau cycle de publication:

Oracle met en place un **nouveau cycle de publication semestriel** pour Java, avec une **version de support à long terme (LTS)** tous les quelques années.

2023-2024 - Évolution continue de Java:

Les différente versions de Java

- De nombreuses versions de Java depuis 1995
  - Java 1.0 en 1995
  - Java 1.1 en 1996
  - Java 1.2 en 1999 (Java 2, version 1.2)
  - Java 1.3 en 2001 (Java 2, version 1.3)
  - Java 1.4 en 2002 (Java 2, version 1.4)
  - Java 5 en 2004
  - Java 6 en 2006
  - Java 7 en 2011
  - Java 8 en 2014
  - Java 9 en 2018
  - …
  - Java 20 en 2023
  - Java 25 (LTS) en 2025
  - Java 27 en 2026
- Évolution très rapide et succès du langage

Les différentes versions de Java

Résumé des versions LTS :

- **Java 8 (2014)** : LTS.
- **Java 11 (2018)** : LTS.
- **Java 17 (2021)** : LTS.
- **Java 21 (2023)** : LTS.
- **Java 25 (2025)** : LTS.
- **Java 27 (2027)** : LTS.

Java évolue rapidement, avec une cadence de sortie de nouvelles versions tous les six mois. Cependant, les versions LTS (Long-Term Support) comme Java 8, Java 11, Java 17 et Java 21 sont privilégiées par les entreprises pour leur stabilité et leur support à long terme.

Qu’est-ce que Java ?

- Java est un langage de programmation:
  - Un programme Java est compilé et interprété (pseudo-compilé)

- Java est une plateforme :
  - La plateforme Java, uniquement software, est exécutée sur la plateforme du système d’exploitation
  - La « Java Platform » est constituée de :
    - La « Java Virtual Machine » (JVM)
    - Kit de développement
    - Des interfaces de programmation d’application (Java API)

Compilé vs Pseudo-Compilé vs interprété

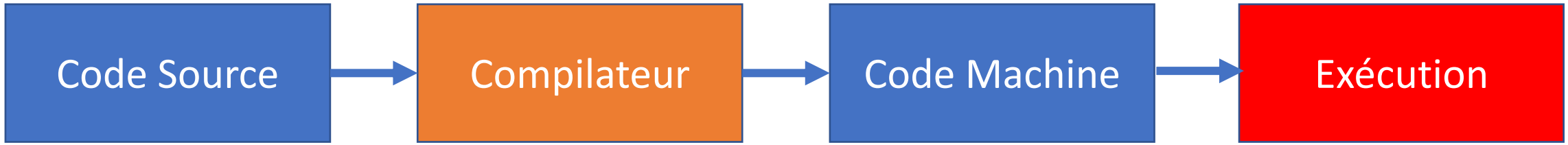
Un langage interprété:



- Le code source est interprété ligne par ligne à l'exécution.
- Facilité de développement et de debugging, mais exécution plus lente.
- Aucune compilation préalable en code machine.

Compilé vs Pseudo-Compilé vs interprété

Un langage compilé:



- Le code source est directement traduit en code machine natif spécifique à une plateforme.
- Exécution rapide car tout est compilé en amont.
- Nécessite une recompilation pour chaque type de machine.
- Exemple : C, C++

Compilé vs Pseudo-Compilé vs interprété

Un langage Pseudo-Compilé:



- Le code source est d'abord compilé en bytecode indépendant de la plateforme.
- Le bytecode est ensuite interprété ou compilé en code machine par la VM lors de l'exécution.
- Portabilité élevée car le bytecode peut être exécuté sur toute machine avec une VM.

Java comme langage de programmation

Simple et orienté objet:

- Java est un langage de programmation simple
  - Langage de programmation au même titre que C/C++/Perl/Smalltalk/Fortran mais plus simple
  - Les aspects fondamentaux du langage sont rapidement assimilés
- Java est orienté objet :
  - La technologie OO après un moment de gestation est maintenant complètement intégrée
  - En java, tout est un objet (à la différence du C++ par ex.)
- Simple aussi parce qu'il comporte un grand nombre d'objets prédéfinis pour l'utilisateur
- Java est familier pour les programmeurs C++

Java comme langage de programmation

Robuste et sécurisé :

- Conçu pour créer des logiciels hautement fiables
- Oblige le programmeur à garder à l'esprit les erreurs hardware et software
- Vérifications complètes à l'exécution et à la compilation
- Existence d'un « garbage collector » qui permet d'éviter les erreurs de gestion de la mémoire

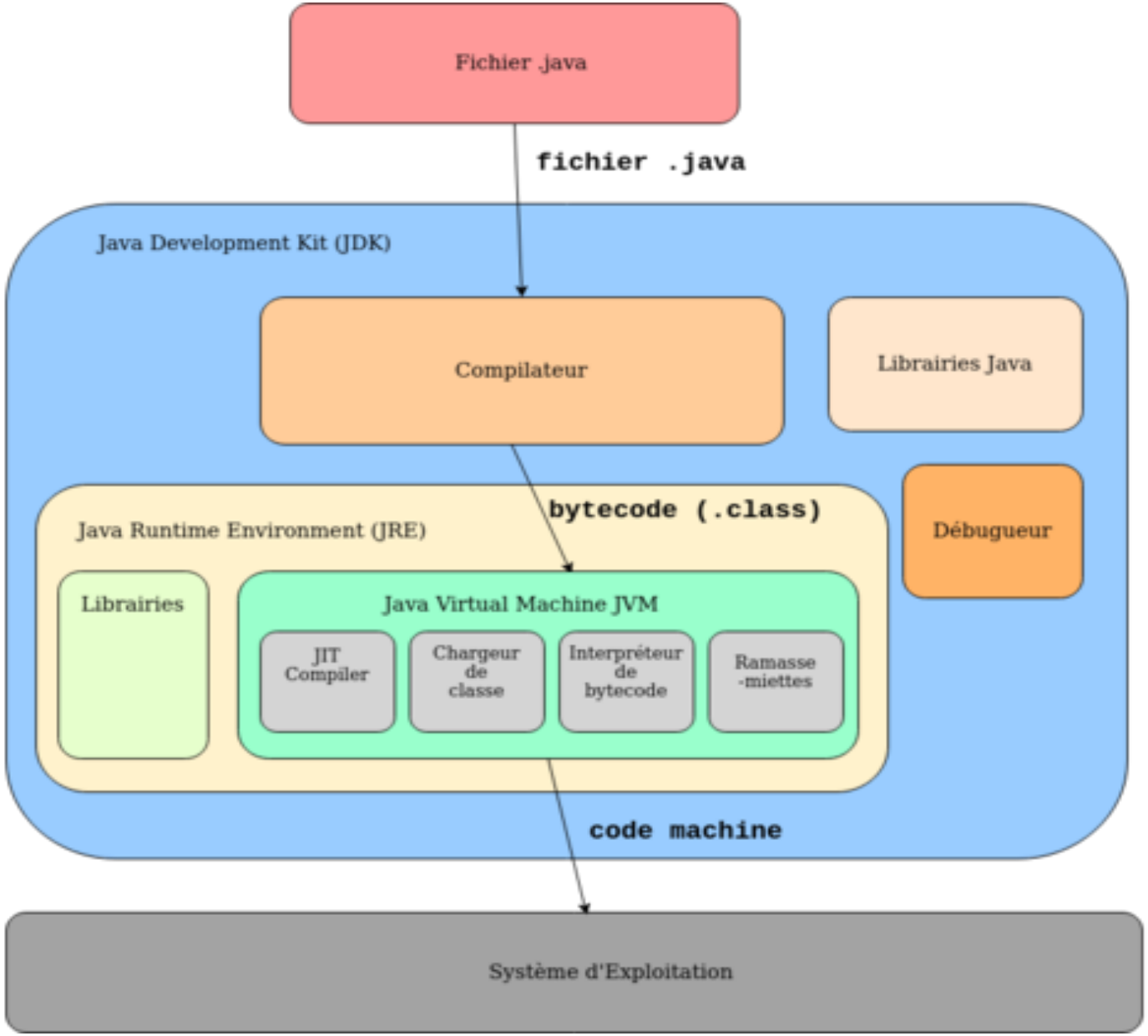
Java comme langage de programmation

Ouvert et distribué :

- Support intégré d'Internet
  - La Class URL
  - Communication réseaux TCP et UDP
  - RMI, CORBA, Servlets
- Connectivité aux bases de données
  - JDBC: Java DataBase Connectivity
  - Offre des facilités de connexions à la plupart des BD du marché
  - Offre un pont vers ODBC
- Support des caractères internationaux
  - Java utilise le jeu de caractères UNICODE
  - JVM équipée de tables de conversion pour la plupart des caractères
  - JVM adapte automatiquement les paramètres régionaux en fonction de ceux de la machine sur laquelle elle tourne

Java comme Plateforme

- Plateforme = environnement hardware ou software sur lequel le programme est exécuté.



Java comme Plateforme

Java Application Programming Interface (API) :

- L'API Java est structuré en libraires (packages). Les packages comprennent des ensembles fonctionnels de composants (classes)..
- Le noyau (core) de l'API Java (incluse dans toute implémentation complète de la plateforme Java) comprend notamment :
  - Essentials (data types, objects, string, array, vector, I/O,date,…)
  - Applet
  - Abstract Windowing Toolkit (AWT)
  - Basic Networking (URL, Socket –TCP or UDP-,IP)
  - Evolved Networking (Remote Method Invocation)
  - Internationalization
  - Security
  - …..

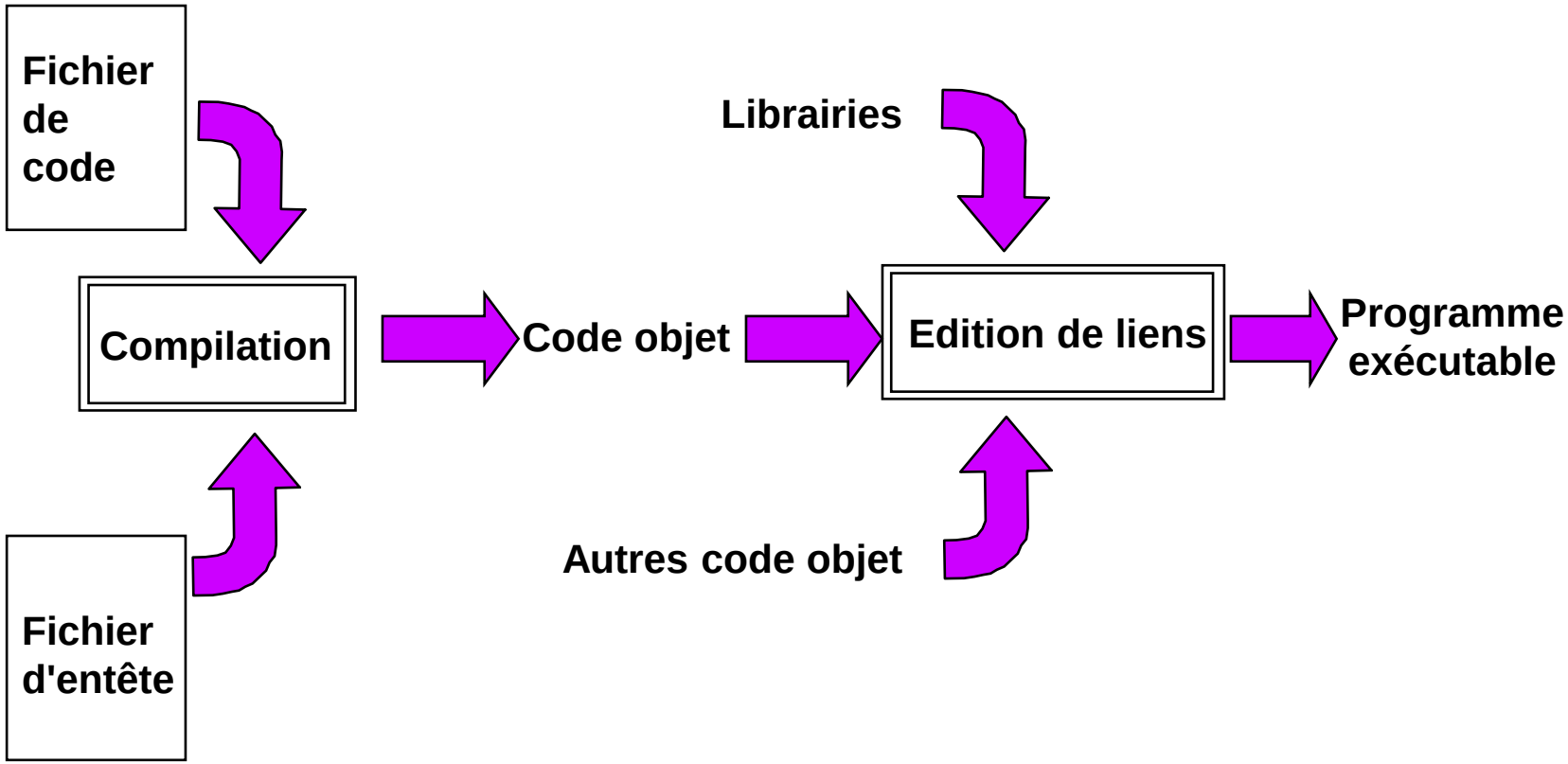
Java comme Plateforme

Java Virtual Machine :

- « An imaginery machine that is implemented by emulating it in software on a real machine. Code for the JVM is stored in .class files, each of which contains code for at most one public class ».
- Définit les spécifications hardware de la plateforme.
- Lit le bytecode compilé (indépendant de la plateforme).
- Implémentée en software ou hardware.
- Implémentée dans des environnements de développement ou dans les navigateurs Web.

Déploiement d'un programme (1/2)

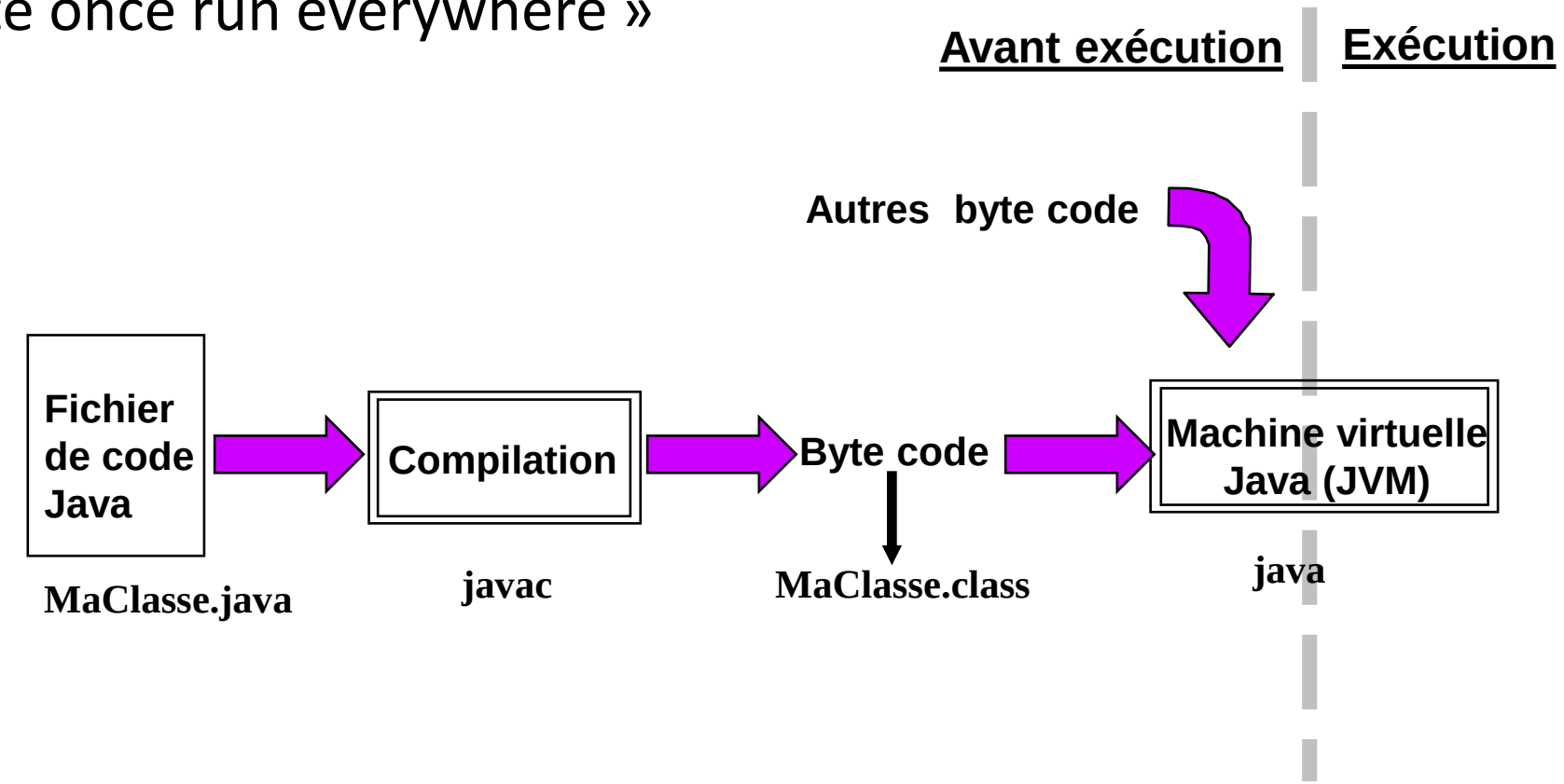
Paradigme classique de la compilation



Déploiement d'un programme (2/2)

Changement de la vision traditionnelle de la compilation

- Chaque programme est compilé et interprété
- « Write once run everywhere »



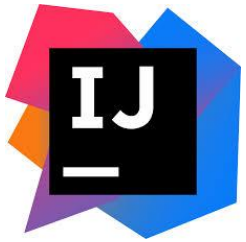
Préparation de l'environnement

1. Télécharger le Kit du développement :
   https://www.oracle.com/java/technologies/downloads
2. Installer un éditeur IDE











Les utilitaires Java

- javac
  - Compilateur, traduit fichier source .java en fichier bytecode .class
- java
  - Interpréteur java, lance des programmes
- javadoc
  - Générateur de documentation d'API
- jar
  - Utilitaire d'archivage et de compression

Les utilitaires Java

Javac

- Compile un fichier source .java ou un package entier
- Exemples:
  - `javac MyBankAccount.java`
    compile le fichier mentionné, qui doit se trouver dans le package par défaut
  - `javac com\fsac\*.java -d c:\classes`
    compile tout le package `com.fsac` et génère du code compilé dans `c:\classes`, qui doit exister

Les utilitaires Java

java

Lance un programme principal:

- `java MyProgram`
- `java com.fsac.MyProgram`
  Lance le programme spécifié dans la classe `MyProgram` qui se trouve dans le package `com.fsac`.

Les utilitaires Java

java

A partir de la version Java 11, la commande **java** permet d'exécuter directement le code sans passer par la première étape de compilation.

- `java MyProgram.java`

L'API de Java

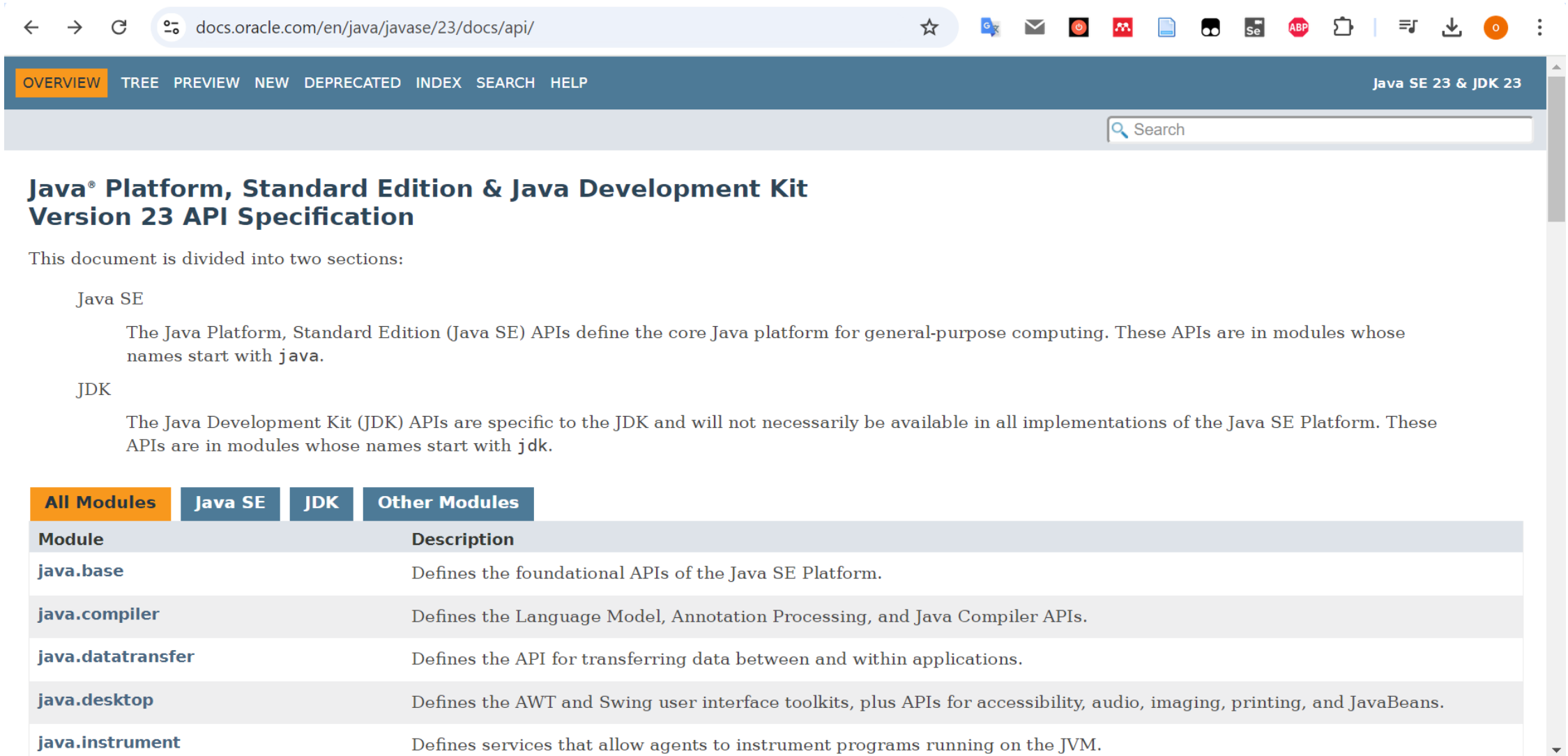
- Java fournit de nombreuses librairies de classes remplissant des fonctionnalités très diverses : c'est l'API Java
  - API (Application and Programming Interface /Interface pour la programmation d'applications) : Ensemble de bibliothèques permettant une programmation plus aisée car les fonctions deviennent indépendantes du matériel.
- Ces classes sont regroupées, par catégories, en paquetages (ou "packages").

L'API de Java

- Les principaux paquetages
  - java.util : structures de données classiques
  - java.io : entrées / sorties
  - java.lang : chaînes de caractères, interaction avec l'OS, threads
  - java.applet : les applets sur le web
  - java.awt : interfaces graphiques, images et dessins
  - javax.swing : package récent proposant des composants « légers » pour la création d'interfaces graphiques
  - java.net : sockets, URL
  - java.rmi : Remote Method Invocation (pas abordé dans ce cours)
  - java.sql : fournit le package JDBC (pas abordé dans ce cours)

L'API de Java

https://docs.oracle.com/en/java/javase/23/docs/api/



Java, un langage novateur ?

- Java n'est pas un langage novateur : il a puisé ses concepts dans d'autres langages existants et sa syntaxe s'inspire de celle du C++.

- Cette philosophie permet à Java
  - De ne pas dérouter ses utilisateurs en faisant "presque comme ... mais pas tout à fait"
  - D'utiliser des idées, concepts et techniques qui ont fait leurs preuves et que les programmeurs savent utiliser

- En fait, Java a su faire une synthèse efficace de bonnes idées issues de sources d'inspiration variées
  - Smalltalk, C++, Ada, etc.

NOTRE PREMIER PROGRAMME JAVA

Point d'entrée d'un programme Java

Pour pouvoir faire un programme exécutable il faut toujours une classe qui contienne une méthode particulière, la méthode « main »

- c'est le point d'entrée dans le programme : le microprocesseur sait qu'il va commencer à exécuter les instructions à partir de cet endroit

```
public static void main(String arg[ ])
{
      ……
}
```

Exemple

Fichier Hello.java

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

La classe est l'unité de base de nos programmes. Le mot clé en Java pour définir une classe est class

Exemple

Fichier Hello.java

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

Une méthode peut recevoir des paramètres. Ici la méthode main reçoit le paramètre args qui est un tableau de chaîne de caractères.

Compilation et exécution (1)

Fichier Bonjour.java

Le nom du fichier est nécessairement celui de la classe avec l'extension .java en plus. Java est sensible à la casse des lettres.

Compilation en bytecode java dans une console DOS:
javac Hello.java
Génère un fichier Hello.class

Exécution du programme (toujours depuis la console DOS) sur la JVM :
java Hello
Affichage de « bonjour » dans la console

A partir de la version java 11 :
java Hello.java
Permet de compiler et exécuter

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

Exercice 1:

- Ecrire une première classe avec une fonction main qui affiche "Hello World".
- Compiler et exécuter cette classe.
- Modifier cette classe pour qu'elle accepte un argument de la console.

Syntaxe de base

Syntaxe de base

- Conventions d’écriture
- Commentaires dans le code source
- Identificateurs
- Mots-clé
- Types primitifs et types de références
- Les tableaux (« Array »)
- La classe String
- Arithmétique et opérateurs
- Instructions de contrôle
  - if, then, else
  - for
  - while
  - do… While
  - break et continue
- Packages

Conventions d'écriture

Classes:

class Bank

class BankAccount

Interfaces:

interface Account

Méthodes:

deposit()

getName()

Packages:

package coursTechnofutur3.bank ;

Variables:

int accountNumber

Constantes:

MAXIMUM_SIZE

Les commentaires

Trois façons d'inclure des commentaires :

- Tout texte entre « // » et la fin de la ligne

```
// Commentaires sur une seule ligne
```

- Tout texte entre « /* » et « */ »

```
/* Commentaires
  sur un nombre important voire très important
  de lignes */
```

- Les textes entre « /** » et « */ » sont utilisés pour créer des commentaires que l'exécutable JAVADOC pourra traiter afin de produire une documentation (cf. documentation de l'API Java)

```
/** Commentaires destinés
à la documentation */
```

Les identificateurs

- Un identificateur (*identifier*) permet de désigner une classe, une méthode, une variable …
- On ne peut utiliser ce que l'on veut :
  - Interdiction d'utiliser les mots-clés
  - Contient des lettres et des chiffres
  - A condition de commencent par :
    - Une lettre
    - Un « $ »
    - Un « _ » (underscore)

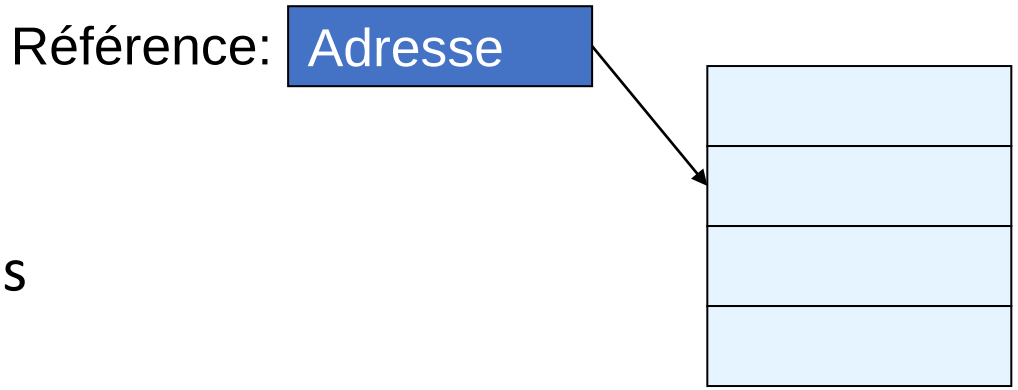
Syntaxe de base

Mots clé

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

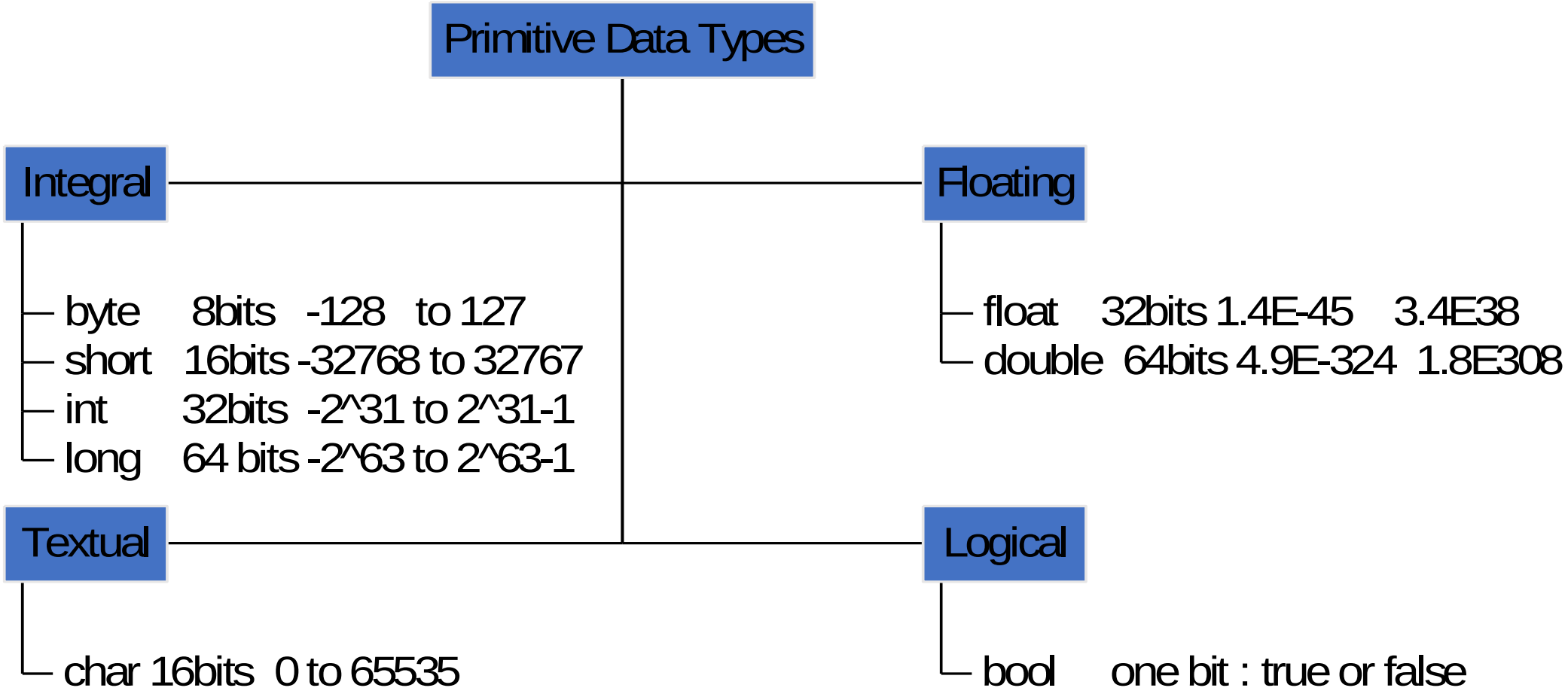
Types primitifs et types de référence

- Java est un langage fortement typé
- Le type de données précise
  - les valeurs que la variable peut contenir
  - les opérations que l'on peut réaliser dessus
- Deux types de données:
  - **Donnée primitive**: contient physiquement la valeur (caractère, nombre, booléen)
  - **Type de référence**: contient l'adresse mémoire où l'information relative à l'objet, l'interface, etc. est réellement stockée



Types primitifs et types de référence

Types de données primitifs :



Types primitifs et types de référence

Types de données primitifs :

```
byte age = 25;                  // Petit nombre entier
short annee = 2026;             // Nombre entier court
int quantite = 1500;            // Nombre entier
long population = 38000000L; // Grand nombre entier

float note = 15.5f;             // Nombre réel simple précision
double prix = 1250.75;          // Nombre réel double précision

char lettre = 'A';              // Un seul caractère
boolean disponible = true;   // Valeur vraie ou fausse
```

Types primitifs et types de référence

Types de référence

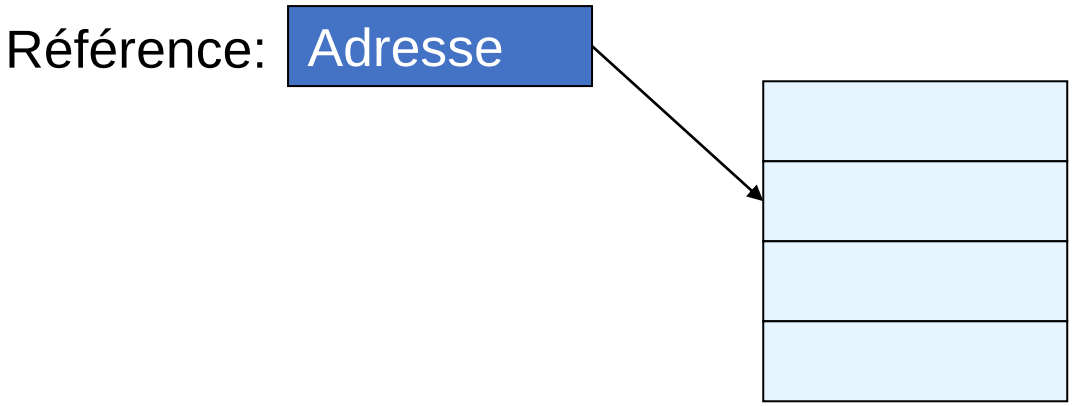
- Tous les types hormis les types primitifs
- « Pointeur implicite » sur un objet

Exemples :

Les chaînes de caractères

Les tableaux

…



Types primitifs et types de référence

Types de référence **Tableaux**

Déclaration et instanciation :

```
double[] notes = new double[5];

int[] ages = new int[100];
```

Déclaration et initialisation :

```
int[] notes = {12, 15, 10, 18, 14};

String[] noms = {"Ali", "Sara", "Ahmed"};

double[] prix = {10.5, 20.0, 15.75};
```

Types primitifs et types de référence

Types de référence **Tableaux**

Les méthodes des tableaux :

```
// Déclaration et initialisation d'un tableau
int[] notes = {12, 15, 10, 18, 14};

// Afficher le nombre d'éléments du tableau
System.out.println("Taille : " + notes.length);

// Afficher tous les éléments du tableau
System.out.println("Tableau : " + Arrays.toString(notes));
```

Types primitifs et types de référence

Types de référence **Tableaux**

Les méthodes des tableaux :

```
// Trier le tableau par ordre croissant
Arrays.sort(notes);
System.out.println("Tableau trié : " + Arrays.toString(notes));

// Remplir toutes les cases avec la valeur 10
Arrays.fill(valeurs, 10);
System.out.println("Tableau rempli : "  + Arrays.toString(valeurs));

// Copier le tableau notes
int[] copie = Arrays.copyOf(notes, notes.length);
System.out.println("Copie : " + Arrays.toString(copie));
```

Types primitifs et types de référence

Types de référence **Tableaux**

Parcourir un tableau :

```
// Parcourir le tableau avec une boucle for
for (int i = 0; i < notes.length; i++) {
    System.out.println(notes[i]);
}

// Parcourir le tableau avec for-each
for (int note : notes) {
    System.out.println(note);
}
```

Types primitifs et types de référence

Types de référence **Chaines de caractères**

Méthodes des chaînes de caractères :

```
String texte = "Bonjour Java";

texte.length();           // Nombre de caractères : 12
texte.toUpperCase();      // "BONJOUR JAVA"
texte.toLowerCase();      // "bonjour java"
texte.charAt(0);          // 'B'
texte.substring(8);       // "Java"
texte.contains("Java");   // true
```

Types primitifs et types de référence

Types de référence Chaines de caractères

Comparer les chaînes de caractères :

```
String nom1 = "Ahmed";
String nom2 = "Ahmed";

if (nom1.equals(nom2)) {
    System.out.println("Les chaînes sont identiques");
}
```

Déclaration des constantes :

En Java, on utilise le mot-clé final pour déclarer une constante. Une fois sa valeur affectée, elle ne peut plus être modifiée.

```
// Constante entière
final int AGE_MAX = 100;

// Constante réelle
final double PI = 3.14159;

/ Constante de type chaîne de caractères
final String PAYS = "Maroc";

// Constante booléenne
final boolean ACTIF = true;

// Déclaration d'un tableau constant
final int[] NOTES = {12, 15, 10, 18, 14};
```

Attention, il est possible de modifier les valeurs

Arithmétique et opérateurs

Arithmétique élémentaire

- Quelle est la valeur de : 5+3*4+(12/4+1)
- Règles de précédences sur les opérateurs:

Niveau	Symbole	Signification
1	()	Parenthèse
2	*	Produit
	/	Division
	%	Modulo
3	+	Addition ou concaténation
	-	Soustraction

Arithmétique et opérateurs

Opérateurs de comparaison

- Pour comparer deux valeurs:

Opérateur	Exemple	Renvoie TRUE si
>	v1 > v2	v1 plus grand que v2
>=	v1 >= v2	Plus grand ou égal
<	v1 < v2	Plus petit que
<=	v1 <= v2	Plus petit ou égal à
==	v1 == v2	égal
!=	v1 != v2	différent

- Opérateurs logiques:

Opérateur	Usage	Renvoie TRUE si
&&	expr1 && expr2	expr1 et expr2 sont vraies
&	expr1 & expr2	Idem mais évalue toujours les 2 expressions
||	expr1 || expr2	Expr1 ou expr2, ou les deux sont vraies
|	expr1 | expr2	idem mais évalue toujours les 2 expressions
!	! expr1	expr1 est fausse
!=	expr1 != expr2	si expr1 est différent de expr2

Arithmétique et opérateurs

Opérateurs d'assignation(d'affectation)

- L'opérateur de base est '='
- Il existe des opérateurs d'assignation qui réalisent à la fois
  - une opération arithmétique, logique, ou bit à bit
  - et l'assignation proprement dite

Opérateur	Exemple	Équivalent à
+=	expr1 += expr2	expr1 = expr1 + expr2
-=	expr1 -= expr2	expr1 = expr1 – expr2
*=	expr1 *= expr2	expr1 = expr1 * expr2
/=	expr1 /= expr2	expr1 = expr1 / expr2
%=	expr1 %= expr2	expr1 = expr1 % expr2

Instructions et structures de contrôle

Structures de contrôle

- Les structures de contrôles permettent d'arrêter l'exécution linéaire des instructions (de bas en haut et de gauche à droite)
- Elles permettent d'exécuter conditionnellement une instruction, ou de réaliser une boucle

Type d'instruction	Mots clés utilisés
Décision	if() else – switch() case
Boucle	for(; ;) – while () – do while()
Traitement d'exceptions	try catch finally – throw
Branchement	label : -- break – continue -- return

Instructions et structures de contrôle

Structures de contrôle - IF – ELSE

L'instruction if en Java est une structure de contrôle de flux qui permet d'exécuter un bloc de code en fonction d'une condition. Elle est couramment utilisée pour effectuer des actions uniquement si une certaine condition est vraie.

```
if (expression)
{
   //instructions
}
else
{
   //instructions dans les autres cas
}
```

```
if (expression)
{
    //instructions
}
```

Instructions et structures de contrôle

Structures de contrôle - IF – ELSE

Les instructions if multiples en Java permettent de vérifier plusieurs conditions les unes après les autres, afin d'exécuter différents blocs de code en fonction de chaque condition. Cela se fait souvent en utilisant des combinaisons de if, else if, et else.

```
if (expression 1)
{
   //instructions 1
}
else if(expression 2){
{
   //instructions 2
}
else if(expression 3){
   //instructions 3
}
else {
   //instructions 4
}
```

Instructions et structures de contrôle

Structures de contrôle - SWITCH - CASE

En Java, l'instruction switch permet de contrôler le flux d'exécution en fonction de la valeur d'une expression, souvent une variable. Elle est souvent utilisée comme alternative aux multiples instructions if-else if, offrant une syntaxe plus concise et lisible pour les cas où une variable peut prendre un ensemble limité de valeurs spécifiques.

```
switch (number)
{
      case 1 : instructions;  break;
      case 2 : instructions; break;
      default : instructions;
}
```

Instructions et structures de contrôle

Structures de contrôle - FOR

La boucle **for** en Java est une structure de contrôle qui permet de répéter un bloc de code un certain nombre de fois, en fonction d'une condition. C'est une des boucles les plus couramment utilisées, particulièrement adaptée lorsque le nombre d'itérations est connu à l'avance.

```
for (initialisation; condition; mise à jour de valeurs){
     // instructions
}
```

Initialisation: à exécuter lorsque le programme rentre pour la première fois dans la boucle
Condition : à remplir pour recommencer le bloc d'instructions
Mise à jour: instruction exécutée chaque fois que la boucle est terminée

Instructions et structures de contrôle

La boucle - WHILE

Définition :

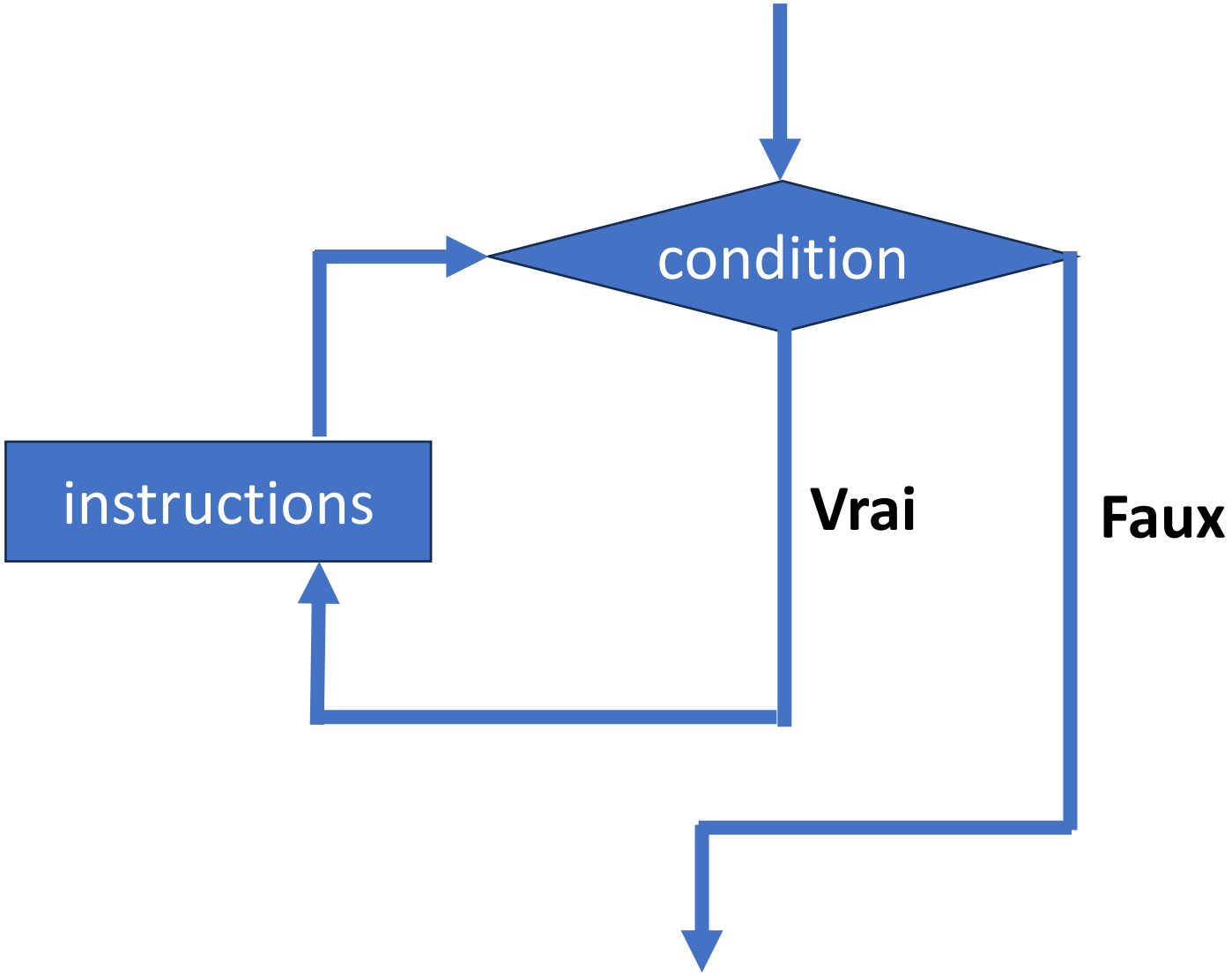
La boucle WHILE est une structure de contrôle en programmation qui permet d'exécuter un bloc de code répétitif tant qu'une condition donnée est vraie. Cela signifie que le bloc de code à l'intérieur de la boucle WHILE est exécuté de manière répétée tant que la condition spécifiée reste vraie. Une fois que la condition devient fausse, l'exécution de la boucle s'arrête et le programme passe à l'instruction suivante après la boucle.

Instructions et structures de contrôle

La boucle - WHILE

Syntaxe :

```
while (condition) {
    //instructions
}
```



Instructions et structures de contrôle

La boucle - DO-WHILE

Définition :

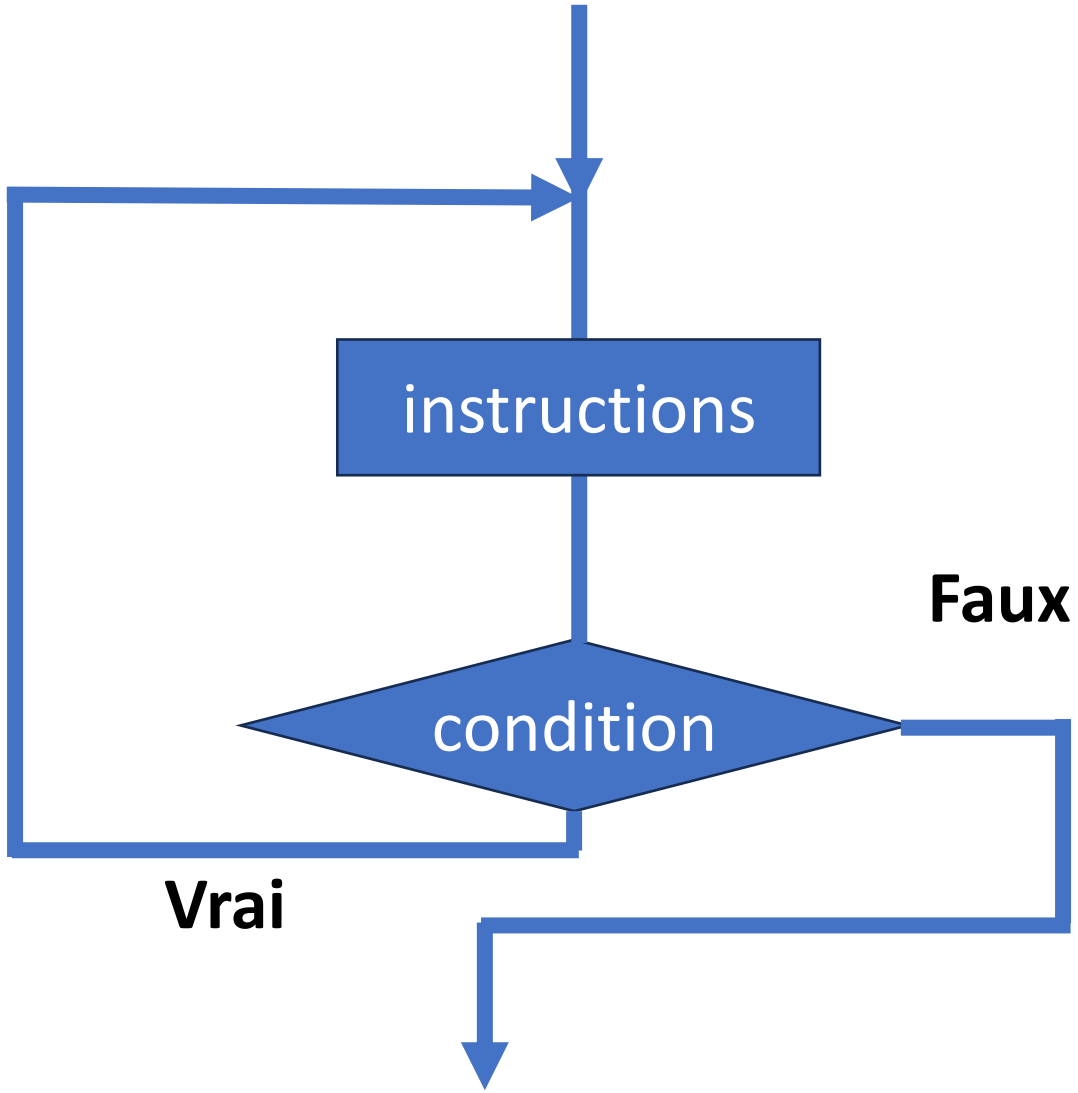
La boucle DO-WHILE est une structure de contrôle en programmation qui permet d'exécuter un bloc de code répétitif jusqu'à une condition donnée est vraie. Le nombre de répétition minimal de cette boucle est 1.

Instructions et structures de contrôle

La boucle - DO-WHILE

Syntaxe :

```
do {
  // instructions
} while (condition)
```



Instructions et structures de contrôle

Structures de contrôle - BREAK / CONTINUE

- BREAK: achève immédiatement la boucle
- CONTINUE: ignore le reste des instructions et recommence au début de la boucle

```
for (int i=0; i<10 ;i++){
    if (i==5) continue; // Si i=5, on recommence au début
    if (i==7) break;    /* Si i=7, on sort de la boucle et
                           les instructions suivantes sont
                           exécutées */
    System.out.println("The value of i is : " + i);
}
```

Instructions et structures de contrôle

Structures de contrôle - BREAK / CONTINUE

BREAK [LABEL]

CONTINUE [LABEL]

```
outer:
for (int i=0 ; i<10 ; i++)  {
  for(int j=20;j>4;j--){
    if (i==5) continue;       // if i=5, you jump to the beginning of the loop
    if (i==7)  break outer;  // if i=7, you jump outside the loop and continue
    System.out.println("The value of i,j is :" + i +  "," + j);
}
```

Instructions d'écriture

Pour afficher dans la sortie standard :

System.out.println() : Affiche un message et passe à la ligne suivante.

System.out.print() : Affiche un message sans passer à la ligne suivante.

Affichage formaté :

System.out.printf() : Affiche un message formaté.

Exemple : System.out.printf("Le factoriel de %d est %d", n, f);

Instructions de lecture

Pour lire de l'entré standard :

En Java, la classe Scanner permet de lire les données entrées par l'utilisateur à partir de la console.

Étapes pour utiliser Scanner :

1. Importer la classe Scanner : import java.util.Scanner;
2. Créer un objet Scanner : Scanner sc = new Scanner(System.in);
3. Utiliser les méthodes de Scanner pour lire des données.

Voici quelques méthodes utiles :

- nextInt() : Lit un entier.
- nextDouble() : Lit un nombre à virgule flottante.
- nextLine() : Lit une ligne complète de texte.
- next() : Lit un seul mot.

Transtypage (casting):

En Java, le transtypage (ou casting) est l'opération qui permet de convertir une variable d'un type à un autre, notamment lorsqu'on travaille avec des types de données primitifs ou des objets. Il en existe deux types principaux : le **transtypage implicite** et le **transtypage explicite**.

Transtypage (casting):

Transtypage Implicite (Upcasting):

Le transtypage implicite se fait automatiquement lorsque la conversion se fait vers un type plus grand ou plus large, c'est-à-dire lorsque la conversion n'entraîne pas de perte de données. C'est le cas, par exemple, du passage d'un int à un double.

```
int x = 42;
double y = x;  // Implicite : int est converti en double
```

Transtypage (casting):

Transtypage Explicite (Downcasting)

Le transtypage explicite est nécessaire lorsque la conversion se fait vers un type plus petit ou plus précis, ce qui pourrait entraîner une perte de données. Il s'applique aux types primitifs ainsi qu'aux types objets.

```
double x = 42.67;
int y = (int) x;  // double est converti en int, la partie décimale est perdue
```

Transtypage (casting):

Conversion des chaînes vers des types primitifs et vice versa :

```
String str = "42";
int a = Integer.parseInt(ch);  // Chaîne en entier

String str = "42.5";
int a = Double.parseDouble(ch);  // Chaîne en double

String str = "true";
int a = Boolean.parseBoulean(ch);  // Chaîne en booléen

int a = 20;
String str = String.valueOf(a);       // Entier en String
String str = Integer.toString(a);     // Entier en String
```

Exercices

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

La programmation orientée objet en Java

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Plan Chapitre 2: Programmations orientée objet (POO)

1. Introduction à POO
2. Concepts fondamentaux de la POO (classe, objet, encapsulation, héritage, abstraction, polymorphisme).
3. Déclaration de classes et création d'objets.
4. Constructeurs et mot-clé this.
5. Accès aux membres de la classe (attributs et méthodes).
6. Modificateurs d'accès (public, private, protected).

POO en Java

Premiers concepts orientés objets

Introduction à la POO

Définition:

Un **paradigme de programmation** est un modèle ou une approche permettant de résoudre des problèmes en suivant un ensemble de principes ou de concepts.
Chaque paradigme définit une façon particulière d'organiser et de structurer le code, influençant la manière dont les développeurs pensent et écrivent leurs programmes.

Principaux paradigmes de programmation :

- **Programmation impérative:**

  La programmation impérative est basée sur l'idée de donner des instructions séquentielles à l'ordinateur, comme une série d'ordres. Le code est construit autour de changements d'état à travers des instructions qui modifient directement les valeurs des variables. Ce paradigme inclut l'utilisation de boucles, de conditions, et de fonctions qui décrivent précisément chaque étape.

  Exemples de langages : C, Fortran, Python, Java.

Principaux paradigmes de programmation :

- **Programmation fonctionnelle:**

  La programmation fonctionnelle se concentre sur les **fonctions pures** et l'**absence d'effets de bord**. Elle évite les variables d'état et privilégie l'immuabilité, où les fonctions reçoivent des arguments et renvoient des résultats sans modifier d'autres données.

  Exemples de langages: Haskell, Lisp, Scala, Python, Java(versions récentes)

Principaux paradigmes de programmation :

- **Programmation logique:**

  Basée sur des règles logiques, ce paradigme permet de définir un ensemble de faits et de règles, et le programme déduit des résultats par la logique. Ce type de programmation est couramment utilisé dans les systèmes experts.

  Exemples de langages: Prolog.

Introduction à la POO

Principaux paradigmes de programmation :

- **Programmation déclarative:**

  La programmation déclarative consiste à décrire ce que le programme doit accomplir sans préciser comment. Elle englobe plusieurs paradigmes (comme la programmation fonctionnelle et la programmation logique) et est utilisée dans des contextes comme les bases de données ou les interfaces utilisateur.

  Exemples de langages : SQL, HTML, CSS.

Introduction à la POO

Principaux paradigmes de programmation :

- **<u>Programmation orientée objet (POO)</u>**

  La programmation orientée objet (POO) est un paradigme de programmation qui organise le code autour d'objets, qui sont des instances de classes. Ce paradigme vise à modéliser des concepts du monde réel ou des entités abstraites sous forme d'objets ayant des attributs (données) et des méthodes (comportements).

  Il est centré sur les concepts **d'encapsulation**, **d'héritage**, de **polymorphisme**, et **d'abstraction**.
  Exemples de langages : Java, C++, Python.

Introduction à la POO

Histoire de la POO:

- **Années 1960:** prémices de la POO avec le langage Simula-67
- **Années 1970-1980:** Smalltalk 72, puis Smalltalk 80 marquent le début effectif de la POO ainsi que la pose de ses concepts de base.
- **A partir de 1980:** adoption des langages orientés objet:
  - 1980: Objective C
  - 1983: C++, Common LISP object
  - 1984: Eiffel
  - 1996: java 1
- **Années 1990:** évolutions du concept qui s'étend aux aspects conception, analyse et bases de données.

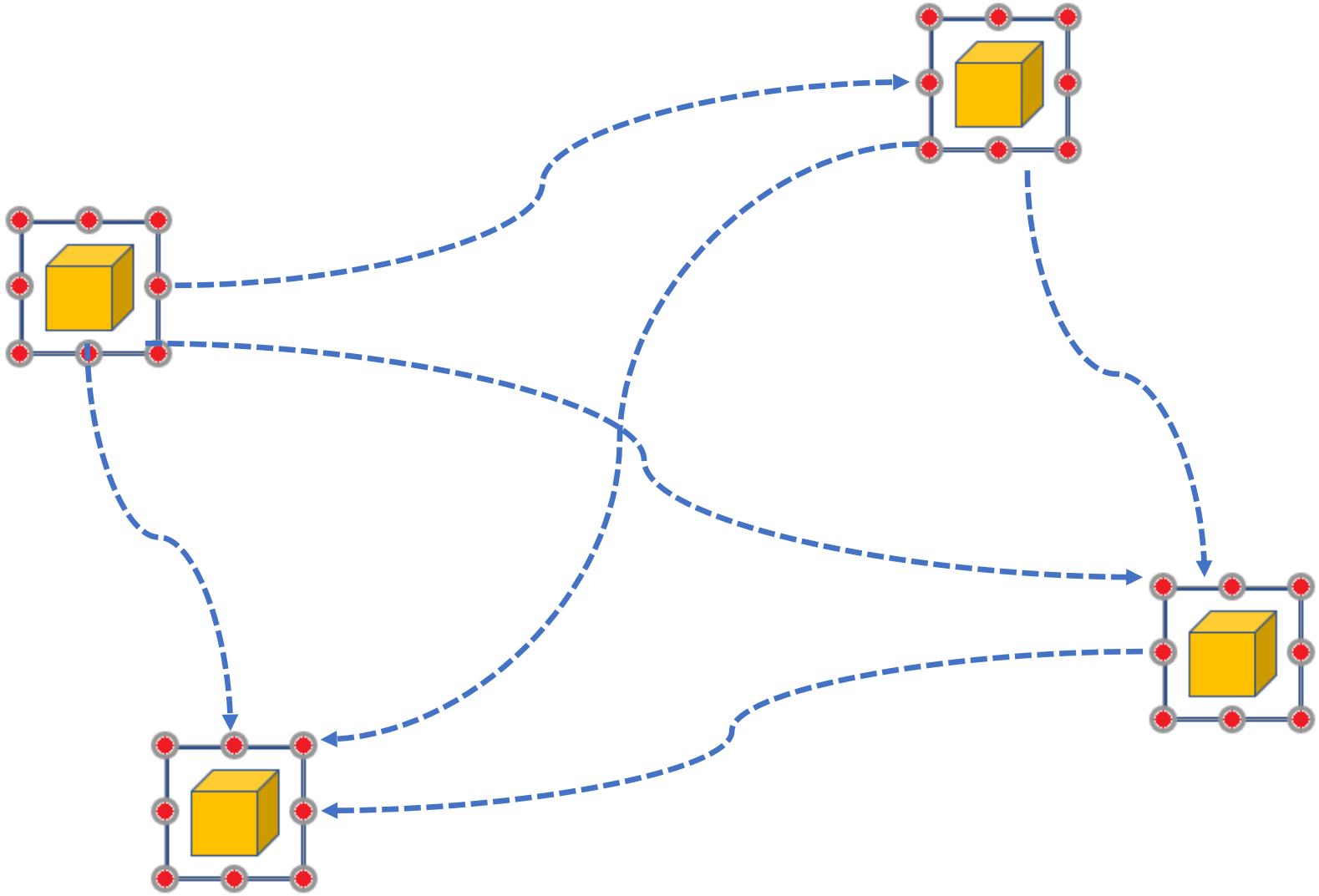
Introduction à la POO

Concepts de la POO:

- Objet
- Class
- Encapsulation
- Héritage
- Polymorphisme
- Abstraction

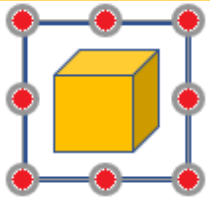
Concepts de la POO

Définition d'un objet



Concepts de la POO

Définition d'un objet



L'objet est une entité qui contient à la fois des données (sous forme **d'attributs**) et des comportements (sous forme de **méthodes**).

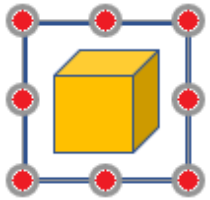
Caractéristiques d'un objet :

1. **Attributs** : Ce sont des variables qui stockent des informations propres à l'objet.
2. **Méthodes** : Ce sont des fonctions associées à un objet qui définissent son comportement.
3. **État** : L'ensemble des valeurs des attributs d'un objet à un moment donné.

Concepts de la POO

Définition d'un objet

Qu'est ce qu'un Objet ?





C1

race = Siamois name = Kati hungry = 2 energy = 9 mood = 7
moew() sleep() play() feed()

race = Persan name = Mika hungry = 8 energy = 2 mood = 2
moew() sleep() play() feed()

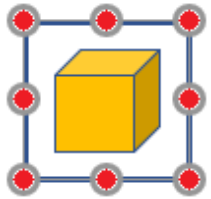


C2

Concepts de la POO

Définition d'un objet

Qu'est ce qu'un Objet ?





C1

race = Siamois
name = Kati
hungry = 2
energy = 9
mood = 7

moew()
sleep()
play()
feed()

hungry ++
energy ++

energy - -
mood ++
meow()

mood ++
energy ++
hungry --

Concepts de la POO

Définition d'un objet



V1

marque = Lucid
annee = 2023
carburant = Electrique
vitesse = 0
niveau_carburant = 53%
Transmission = P

arreter()
demarrer()
accelerer()
ralentir()
charger()

vitesse = 0
transmission = P

vitesse ++
transmission ++

vitesse --
transmission - -

niveau_carburant ++

Concepts de la POO

Définition d'un objet



V1

marque = Lucid
annee = 2023
carburant = Electrique
vitesse = 0
niveau_carburant = 13%
transmission = P

arreter()
demarrer()
accelerer()
ralentir()
charger()

nom = Mohamed
age = 50
ville = Casablanca
voiture = v1

manger()
dormir()
seDeplacer(v1)
changer_voiture()



P1

Concepts de la POO

Définition d'une classe

Une classe est **l'abstraction** d'un ensemble d'objet ayant une structure de **données commune** et disposant des **même méthodes**.

Une classe est un type permettant de regrouper dans la même structure :

- les informations (**attributs**) relatives à une entité ;
- les procédures et fonctions permettant de les manipuler (**méthodes**).

Concepts de la POO

Définition d'une classe

Classe vs Objet ?

Termes techniques :

- « Classe » est la structure (C'est le type) ;
- « Objet » est une instance de la classe (variable obtenue après instanciation) ;
- « Instanciation » correspond à la création d’un objet

Concepts de la POO

Définition d'une classe

Classe vs Objet ?

race name hungry energy mood
moew() sleep() play() feed()

Classe : Chat

instancier

instancier



C1

race = Siamois name = Kati hungry = 8 energy = 2 mood = 3
moew() sleep() play() feed()



C2

race = Persan name = Mika hungry = 2 energy = 9 mood = 8
moew() sleep() play() feed()

Concepts de la POO

Définition d'une classe

Exercice 1:

Donner la définition d'une classe **Point**, un point est caractérisé par ses coordonnées.

Donner la définition d'une classe **Cercle** qui modélise un cercle. Un cercle est caractérisé par son centre et son rayon.

Ajouter des méthodes utilitaires aux deux classes.

Concepts de la POO

Définition d'une classe

Exercice 1: Solution

Point
- x : Réel - y : Réel
+ deplacer(dx: Réel, dy: Réel)

Cercle
- centre : Point - rayon : double
+ perimetre() + surface() + deplacer(dx: Réel, dy: Réel)

Concepts de la POO

L'encapsulation

Principe :

Les méthodes agissent exclusivement sur les données de l'objet.

L'encapsulation est un concept qui concerne les données (attributs) d'un objet.

L'encapsulation (définition): est ce concept de la POO qui interdit l'accès (en lecture/écriture) direct aux données d'un objet, mais qui le permet via des méthodes réservées à cet effet.

Concepts de la POO

L'encapsulation

Notion de visibilité :

- La visibilité sert à garantir la protection des données en :
  - en permettant de masquer les données et certaines méthodes les gérant,
  - et en laissant visibles d'autres méthodes devant servir à la gestion publique de l'objet.

Concepts de la POO

L'encapsulation

Type de visibilités :

- Publique (+)
- Privée (-)
- Protégée (#)
- Visibilité Package (~)

Concepts de la POO

L'encapsulation

Visibilité publique:

- Les attributs et méthodes sont publiques : Cela signifie qu'ils sont accessibles depuis tous les descendants et dans tous les modules.

- Les attributs et méthodes publiques n'ont pas besoin de restriction particulière.

Concepts de la POO

L'encapsulation

Visibilité privée:

- La visibilité privé restreint la porté d'un attribut ou d'une méthode au module où il/elle est déclaré(e).

- Ainsi, si un objet est déclaré dans une unité avec une donnée privée, alors cette donnée ne pourra être accédé qu'à l'intérieur même de l'unité.

L'encapsulation

Visibilité protégée:

- La visibilité 'protégé' correspond à la visibilité 'privé' excepté que tout attribut ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où il se situe.

Concepts de la POO

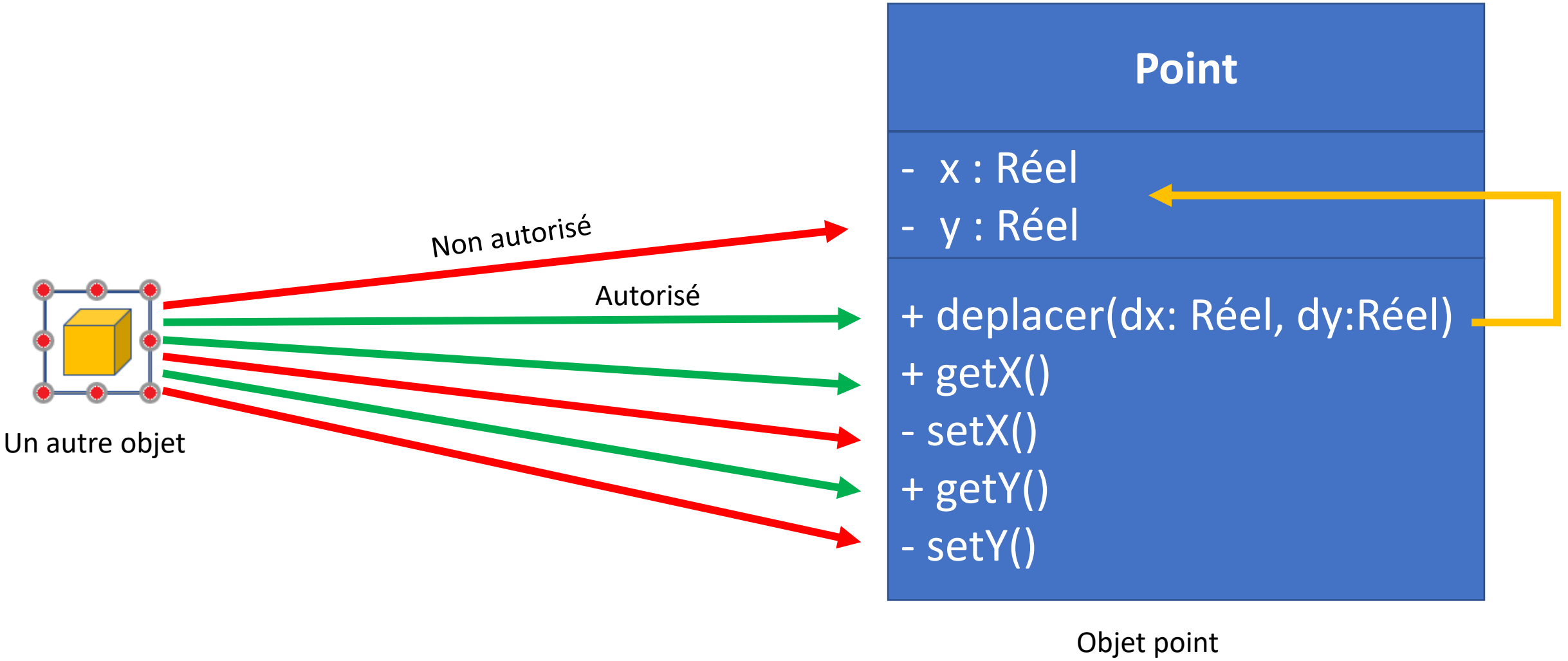
L'encapsulation

Visibilité package:

- Package (visibilité par défaut) (~) : Accessible uniquement aux classes du même package.

Concepts de la POO

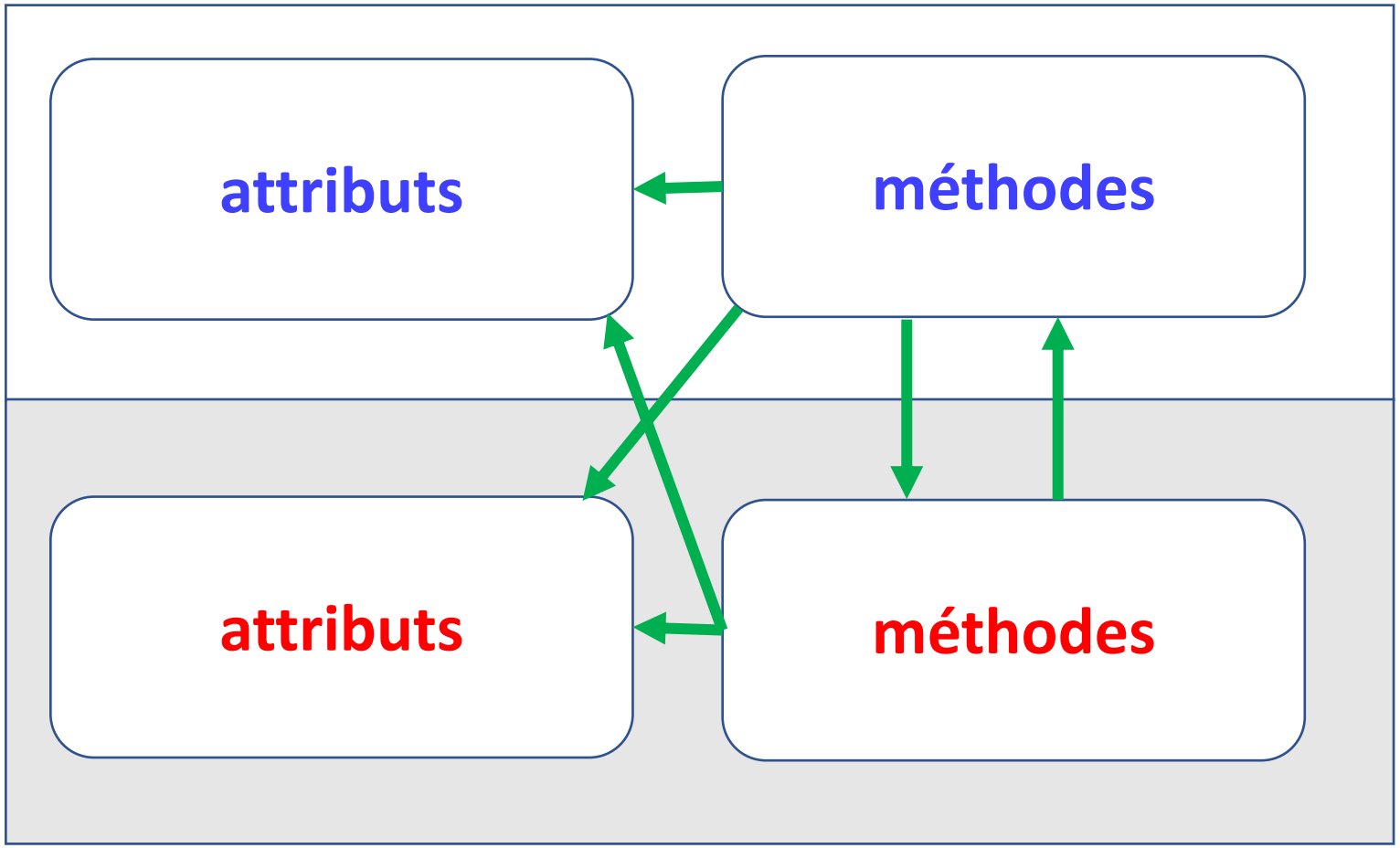
L'encapsulation



Concepts de la POO

L'encapsulation

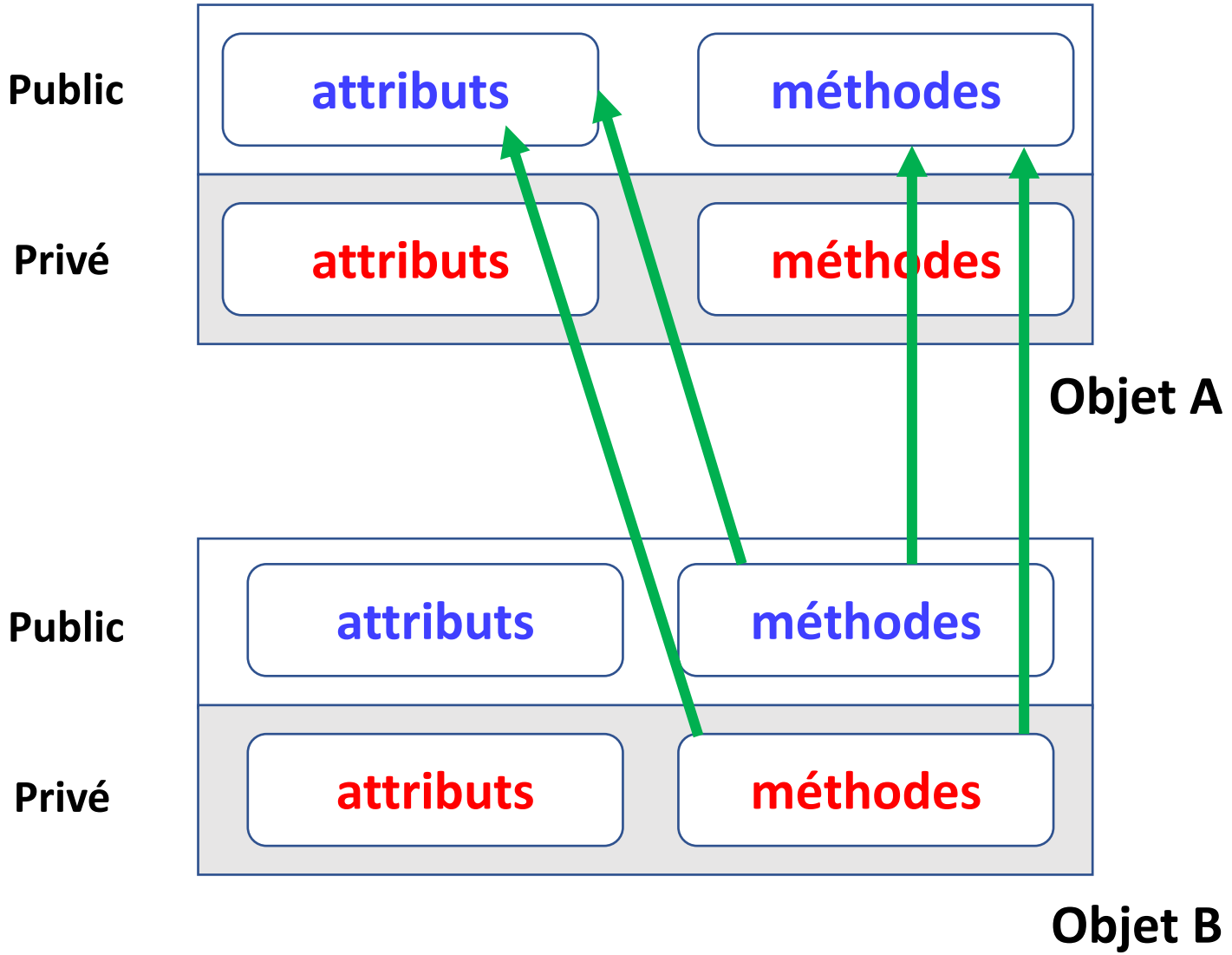
Au sein du même objet :



Concepts de la POO

L'encapsulation

Entre deux objets différents :



Concepts de la POO

L'encapsulation

Recommandation:

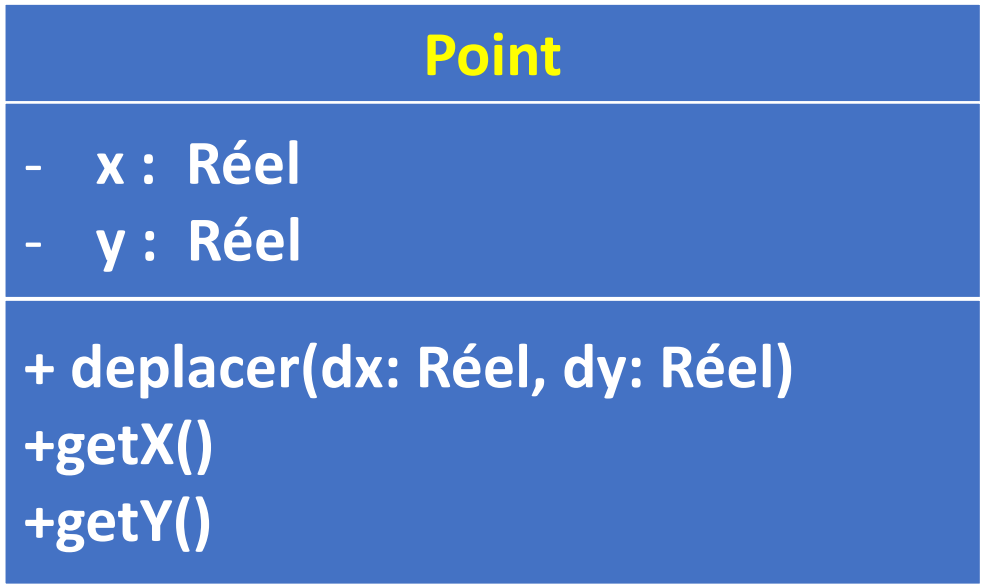
- Il est recommandé d'utiliser des attributs privés et les rendre accessibles en lecture ou modification via des méthodes publics ou protégés.

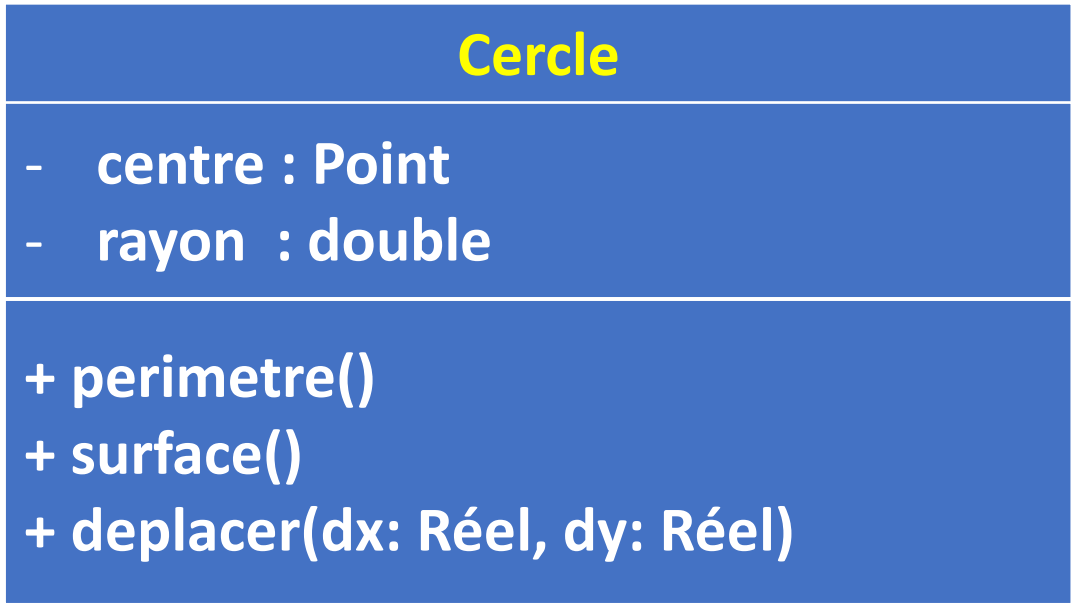
Exemple : attribut (age) qui doit être positif.

Concepts de la POO

Définition d'une classe

Exercice : Implémenter en Java les classes suivantes





Exercices

Concepts de la POO

L'héritage

Définition:

L'héritage est une manière de définir des abstractions de façon incrémentale:

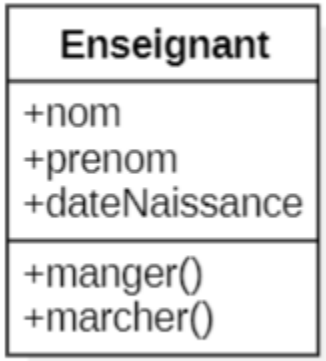
- une définition B peut "hériter" d'une autre définition A.
- la définition B prend A comme base, avec éventuellement des modifications et des extensions.

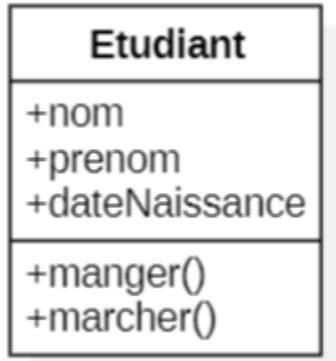
Héritage (définition): L'héritage en programmation orientée objet (POO) est un mécanisme qui permet à une classe (dite classe fille ou dérivée) d'hériter des propriétés et des méthodes d'une autre classe (dite classe mère ou base). Cela favorise la réutilisation du code et la création d'une hiérarchie entre les classes.

Concepts de la POO

L'héritage

Exemple:

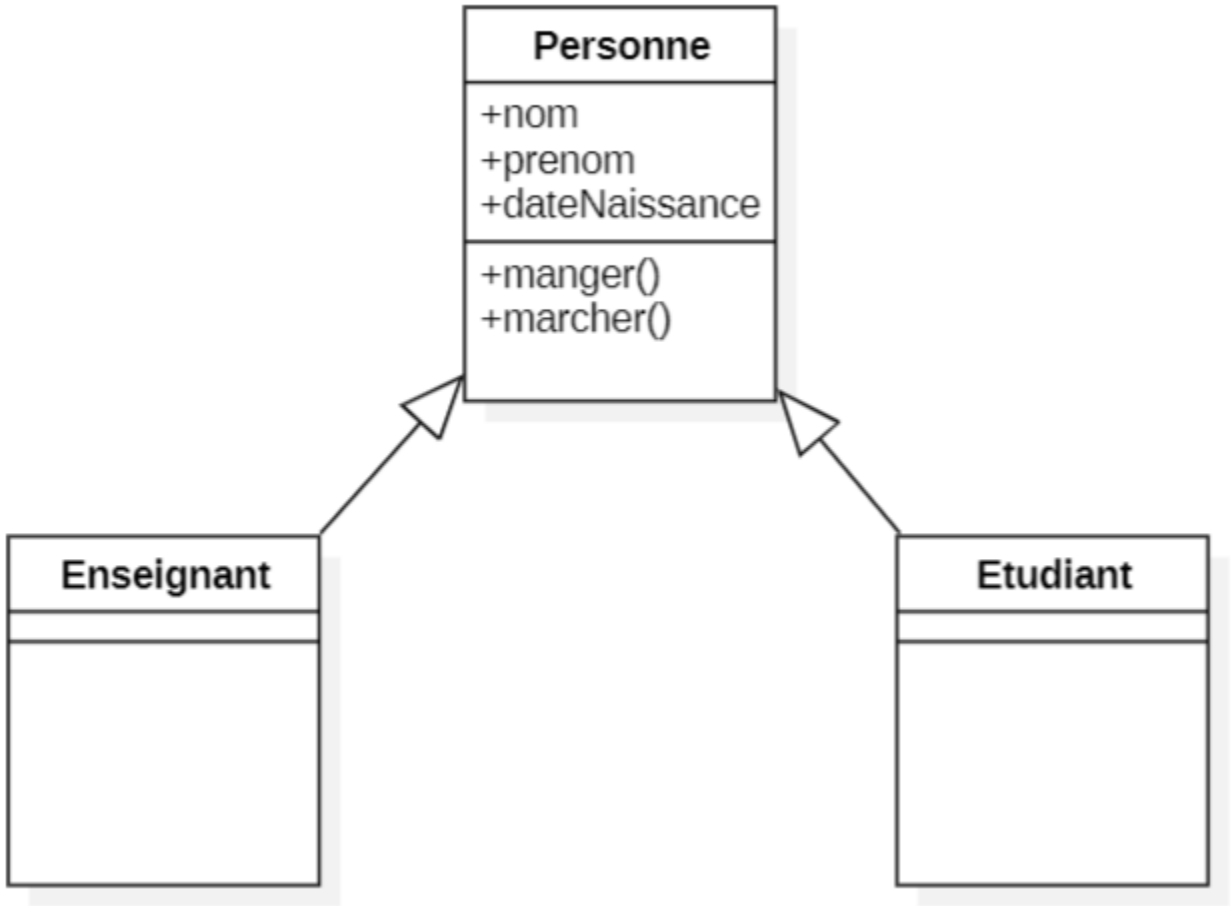




Concepts de la POO

L'héritage

Exemple:



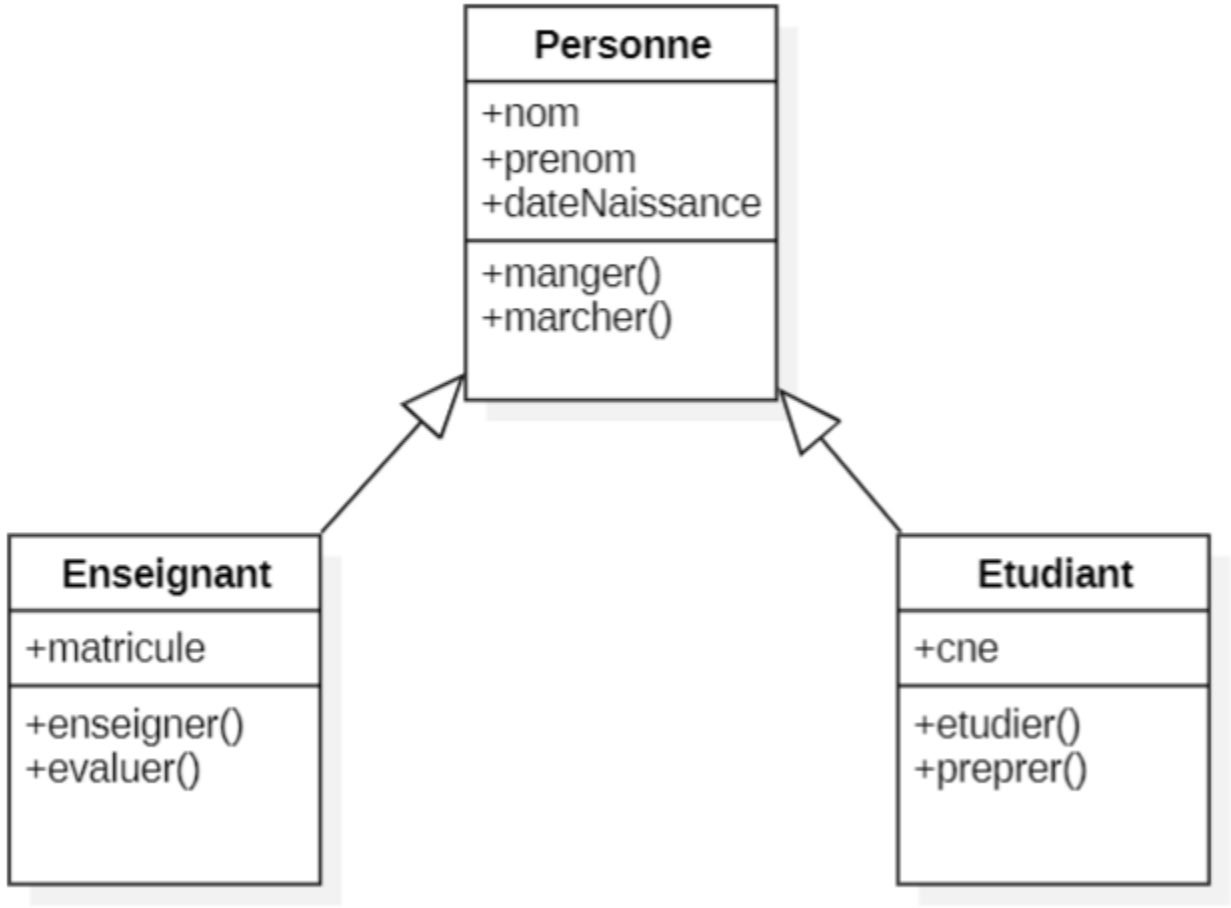
Classe Mère

Classe Fille

Concepts de la POO

L'héritage

Exemple:



Concepts de la POO

L'héritage

Exemples :

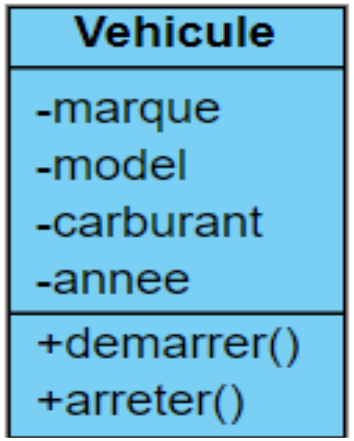
Pour chacun des abstractions suivantes trouver des exemples pour illustrer le concept d'héritage:

- Vehicule
- Animal

Concepts de la POO

L'héritage

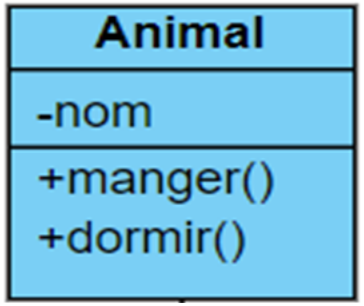
Exemples :



Concepts de la POO

L'héritage

Exemples :



Concepts de la POO

L'héritage

En Java :

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

    public Personne(String nom, String prenom, Date dateNaissance) {
        this.nom = nom;
        this.prenom = prenom;
        this.dateNaissance = dateNaissance;
    }
    public void manger(){
        System.out.println("La personne mange.");
    }
    public void marcher(){
        System.out.println("La personne marche.");
    }
    // getters and setters
}
```

Concepts de la POO

L'héritage

En Java :

```
public class Etudiant extends Personne {


    public Etudiant (String nom, String prenom, Date dateNaissance) {
       super(nom, prenom, dateNaissance);
    }


}
```

Concepts de la POO

L'héritage

En Java :

```
public class Etudiant extends Personne {

    private String cne;

    public Etudiant (String nom, String prenom, Date dateNaissance, String cne) {
        super(nom, prenom, dateNaissance);
        this.cne = cne;
    }

    public String getCne(){
        return cne;
    }

    public void setCne(String cne) {
        this.cne = cne;
    }

}
```

Concepts de la POO

L'héritage

En Java :

```
public class Etudiant extends Personne {

    private String cne;

    public Etudiant (String nom, String prenom, Date dateNaissance, String cne) {
        super(nom, prenom, dateNaissance);
        this.cne = cne;
    }

    public void sepresenter() {
        System.out.println("Mon nom est " + this.nom + "CNE : " + cne);
    }

    …
}
```



Concepts de la POO

L'héritage

En Java :

Type de visibilités :

- Publique (+)
- Privée (-)
- Protégée (#)

Concepts de la POO

L'héritage

En Java :

Un classe fille peut accéder directement aux attributs protégés :

```
public class Personne {
    protected String nom;
    …
}
```

```
public class Etudiant extends Personne {
     public void afficher (){
          System.out.println("Mon nom est " + this.nom + "CNE : " + this.cne);
     }
}
```

Concepts de la POO

Polymorphisme

Définition:

Polymorphisme: signifie « plusieurs formes ». En programmation orientée objet, il se manifeste par la capacité d'une seule interface (ou méthode) à s'adapter à différents types d'objets et à produire des comportements différents en fonction de leur type réel.

Plusieurs types :

- Polymorphisme par sous-typage (ou par héritage)
- Polymorphisme ad-hoc
- …

Concepts de la POO

Polymorphisme par héritage

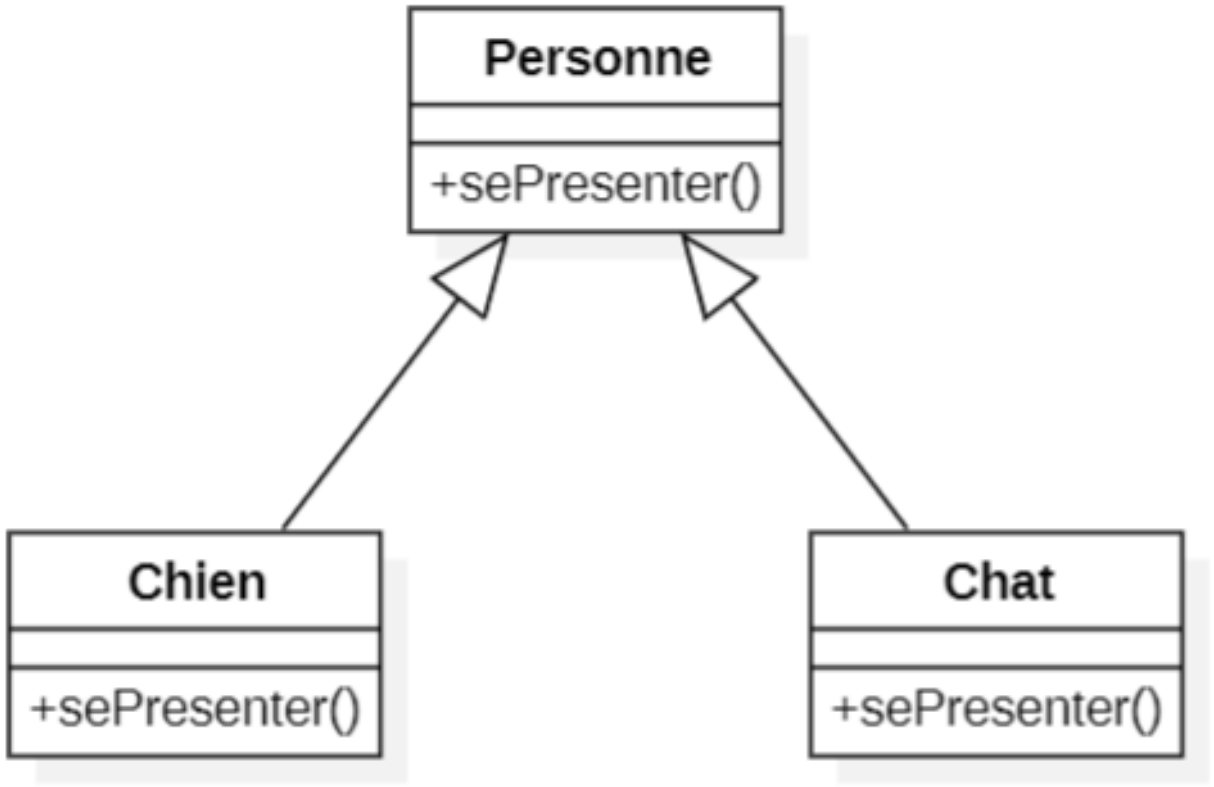
Définition :

Le polymorphisme par héritage est un concept fondamental de la programmation orientée objet (POO) où une classe dérivée (ou sous-classe) peut remplacer ou étendre les comportements définis dans une classe de base (ou super-classe). Ce type de polymorphisme permet d'utiliser des objets de sous-classes à travers une référence de super-classe, offrant une flexibilité et une extensibilité accrues.

Concepts de la POO

Polymorphisme par héritage

Exemple:



Concepts de la POO

Polymorphisme par héritage

Exemple:

Redéfinir une méthode :

```
public class Personne {


    public void sePresenter (){
            System.out.println("Je suis une personne");
     }


}
```

Concepts de la POO

Polymorphisme par héritage

Exemple:

Redéfinir une méthode :

```
public class Etudiant extends Personne {


    public void sePresenter (){
            System.out.println("Je suis un étudiant");
     }


}
```

Concepts de la POO

Polymorphisme par héritage

Exemple:

Redéfinir une méthode :

```
public class Enseignant extends Personne {


    public void sePresenter (){
            System.out.println("Je suis un enseignant");
     }


}
```

Concepts de la POO

Polymorphisme Ad-Hoc

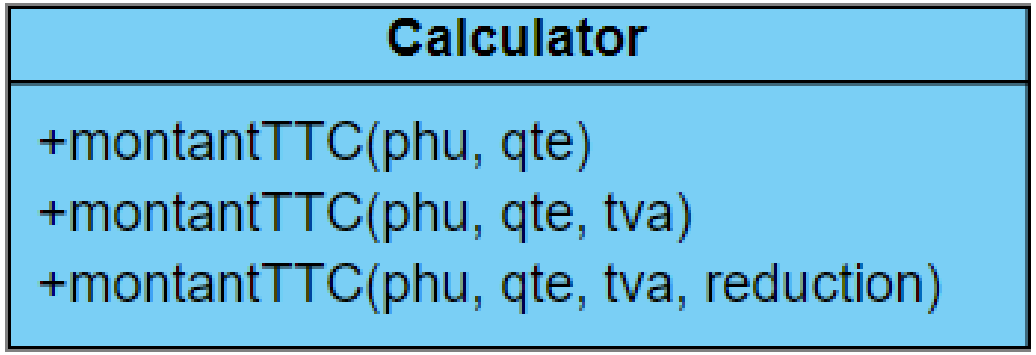
Définition :

Le polymorphisme ad-hoc est un type de polymorphisme qui permet d'utiliser une même fonction ou méthode avec différents types de données, souvent en fonction de la signature (nombre ou type d'arguments) des appels.

Concepts de la POO

Polymorphisme Ad-Hoc

Exemples :



Concepts de la POO

Héritage : Instanciation

Instanciation directe de la classe fille

```
Etudiant e = new Etudiant("Sido","Kamal", new Date(),"F2322442");
e.sePresenter();
```

Concepts de la POO

Héritage : Instanciation

Instanciation d'une classe fille via une référence de la classe mère :

```
Personne e = new Etudiant("Sido","Kamal", new Date(),"F2322442");
e.sePresenter(); // Appele la méthode redéfinie (override)
```

Concepts de la POO

Héritage : Instanciation

Un tableau d'objets :

```
Personne liste[] = new Personne[10];
liste[0] = new Personne("Doe","John", new Date(1990,1,1));
liste[1] = new Etudiant("Smith","Jane", new Date(1992,2,2), "CNE1256");
liste[2] = new Enseignant("Brown","Charlie",new Date(1988,3,3), "MAT7012");
...


for (int i = 0; i < liste.length; i++) {
            liste[i].sePresenter();
}
```

Concepts de la POO

Abstraction

Définition:

L'abstraction est l'un des principes fondamentaux de la programmation orientée objet (POO). Elle consiste à simplifier la complexité d'un système en se concentrant sur les aspects essentiels tout en cachant les détails de l'implémentation. L'objectif est de créer des classes, des objets, et des interfaces qui exposent uniquement les fonctionnalités nécessaires, tout en masquant la logique interne.

Concepts de la POO

Polymorphisme par héritage

Exemple: Déclarer une méthode abstraite dans une classe abstraite

```
public abstract class Personne {
    ...
    public abstract void sePresenter ();

}
```

Polymorphisme par héritage

Les classes abstraites :

- Une classe abstraite
  - Peut contenir ou hériter de méthodes abstraites (des méthodes sans corps)
  - Peut contenir des constantes globales
  - Peut avoir des méthodes normales, avec corps
- Une classe abstraite ne peut être instanciée
  - On peut seulement instancier une sous-classe concrète
  - La sous-classe concrète doit donner un corps à toute méthode abstraite
- La déclaration d'une classe abstraite contenant une méthode abstraite ressemble à ceci:

Concepts de la POO

Héritage : Instanciation

Un tableau d'objets :

```
Personne liste[] = new Personne[10];
liste[0] = new Personne("Doe","John", new Date(1990,1,1));
liste[1] = new Etudiant("Smith","Jane", new Date(1992,2,2), "CNE1256");
liste[2] = new Enseignant("Brown","Charlie",new Date(1988,3,3), "MAT7012");
...


for (int i = 0; i < liste.length; i++) {
            liste[i].sePresenter();
}
```

Les interfaces

Définition

- Une interface peut être considérée comme une classe 100% abstraite.
- Une interface décrit une fonctionnalité sous forme d'une liste de méthodes mais ne fournit aucune implémentation de ces méthodes.
- Une interface ne peut contenir de variable d'instance.
- Les seuls attributs autorisés sont les constantes déclarées static et final.
- Une interface ne peut être instanciée.Une classe peut implémenter une interface, ou bien des interfaces multiples

Concepts de la POO

Les interfaces

Java ne supporte pas l'héritage multiple :

```
public abstract class Nageur {
     public abstract void nager();
}
```

```
public abstract class Volant {
     public abstract void voler();
}
```

Concepts de la POO

Les interfaces

Java ne supporte pas l'héritage multiple :

```
public class Poisson extends Nageur {
      public void nager(){
            System.out.println("Je nage!");
      }
}
```

Concepts de la POO

Les interfaces

Java ne supporte pas l'héritage multiple :

```
public class Aigle extends Volant {
      public void voler(){
            System.out.println("Je vole!");
      }
}
```

Concepts de la POO

Les interfaces

Java ne supporte pas l'héritage multiple :

```
public class Canard extends  ?????  {


     ...


}
```

Concepts de la POO

Les interfaces

Déclarer une interface :

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

```
public interface Volant {
     public void voler();
}
```

Concepts de la POO

Les interfaces

Implémenter des interfaces :

```
public class Canard implements  Nageur, Volant  {

      public void nager(){
            System.out.println("Je nage!");
      }

      public void voler(){
            System.out.println("Je vole!");
      }
}
```

Concepts de la POO

Les interfaces

Instanciation :

```
Aigle v1= new Aigle();

Poisson p1= new Poisson();

Canard c1= new Canard();


public void afficher(Volant obj){
        ...
}
```

Introduction à Java

Programmation orientées objets en Java

Survol du chapitre

- La déclaration des classes
- La création d'objets: Constructeurs et mot-clé « new »
- Les variables: Déclaration et portée
- L'encapsulation: « public », « private » et « protected »
- Les méthodes: Déclaration, interface et surcharge
- Les membres d'instance et de classe: « static »
- Utilisation de l'héritage: « this » et « super »
- Conversion de types
- Polymorphisme
- Classes abstraites
- Interfaces

La déclaration des classes

En Java une classe est déclarée par le mot-clé **class** suivi du nom de la classe.

Il faut aussi spécifier la visibilité de la classe.

```
public class Point
{
      // Déclaration des attributs

      // Déclaration des méthodes
}
```

Les attributs

Déclaration des variables membres (1/6)

- Une variable est un endroit de la mémoire à laquelle on a donné un nom de sorte que l'on puisse y faire facilement référence dans le programme
- Une variable a une valeur, correspondant à un certain type.
- La valeur d'une variable peut changer au cours de l'exécution du programme

Les attributs

Déclaration des variables membres (2/6)

- Rappel: toute variable doit être déclarée et initialisée

- Syntaxe :

[modificateurs] type identificateur [= valeur];

Les attributs

Déclaration des variables membres (3/6)

Les modificateurs d'accès (visibilité) :

1. public : visibilité publique
2. private: visibilité privée
3. protected: visibilité protégée;

Les attributs

Déclaration des variables membres (4/6)

Autres modificateurs :

1. [static]: permet la déclaration d'une variable de classe
2. [final]: empêche la modification de la variable
3. [transient]: on ne tient pas compte de la variable en sérialisant l'objet
4. [volatile]: pour le multithreading

Les attributs

Déclaration des variables membres (5/6)

Exemples :

```
int a;
int a = 0;
public int a;
private int a;
public final float PI = 3.14;
public transient int a = 100;
```

Les attributs

Déclaration des variables membres (6/6)

```
public class Point{

    // les attributs de la classe
    private int x = 0;
    private int y = 0;

    // les méthodes de la classe

}
```

```
public class Point{

    // les attributs de la classe
    private int x = 0, x = 0;

    // les méthodes de la classe

}
```

La création d'objets (1/2)

Le constructeur

- A le même nom que la classe
- Quand un objet est créé, on invoque tout d'abord le constructeur de la classe
- Un constructeur utilise comme arguments des variables initialisant son état interne
- On peut surcharger les constructeurs, i.e définir de multiples constructeurs
- Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
- Signature d'un constructeur:
  - Modificateur d'accès (en général public)
  - Pas de type de retour
  - Le même nom que la classe
  - Les arguments sont utilisés pour initialiser les variables de la classe

```
public class Point {

    // les attributs de la classe
    private int x = 0, x = 0;

    // constructeur par défaut
    public Point( ){
        // instructions
    }

    // constructeurs avec deux paramètres
    public Point(int x, int y){
        // instructions
    }
}
```

La création d'objets (2/2)

L'appel au constructeur

- Se fait pour initialiser un objet
  - ➔Provoque la création réelle de l'objet en mémoire
  - ➔Par l'initialisation de ses variables internes propres
- Se fait par l'emploi du mot clé « new »

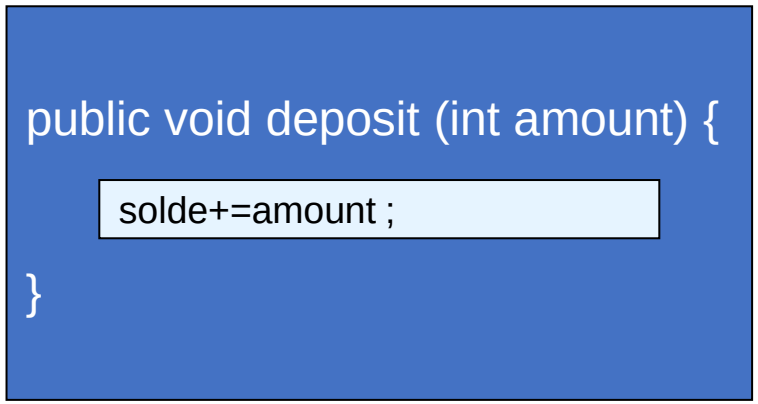
```
Point p1, p2;

p1 = new Point();

p2 = new Point(5, 4);
```

Les méthodes (1/2)

Déclaration d'une méthode



Une méthode est composée de:

- Signature d'une méthode:
  - Modificateurs d'accès : public, protected, private, aucun
  - [modificateurs optionnels] : static, native, synchronized, final, abstract
  - Type de retour : type de la valeur retournée
  - Nom de la méthode (identificateur)
  - Listes de paramètres entre parenthèses (peut être vide mais les parenthèses sont indispensables)
- Au minimum:
  - La méthode possède un identificateur et un type de retour
  - Si la méthode ne renvoie rien ➔ le type de retour est void

Les méthodes (2/2)

La surcharge de méthodes

- La surcharge est un mécanisme qui consiste a dupliquer une méthode en modifiant les arguments de sa signature
- Exemple:

```
Public Account {
        int solde ;
        public void deposit(int amount){
          solde+=amount;
        }
        public void deposit(double amount) {
                solde +=(int) amount;
        }
}
```

L'encapsulation (1/2)

Raisons d'être

Les modificateurs d'accès qui caractérisent l'encapsulation sont justifiées par différents éléments:

- Préservation de la sécurité des données
  - Les données privées sont simplement inaccessibles de l'extérieur
  - Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques
- Préservation de l'intégrité des données
  - La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable
- Cohérence des systèmes développés en équipes
  - Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

L'encapsulation (2/2)

Accès aux membres d'une classe

- En java, les modificateurs d'accès sont utilisés pour protéger l'accessibilité des variables et des méthodes.
- Les accès sont contrôlés en respectant le tableau suivant:

Mot-clé	classe	package	sous classe	world
`private`	Y			
`protected`	Y	Y	Y	
`public`	Y	Y	Y	Y
`[aucun]`	Y	Y		

Seul les membres publics sont visibles depuis le monde extérieur.

Une classe a toujours accès à ses membres.

Les classes d'un même package protègent uniquement leurs membres privés (à l'intérieur du package)

Une classe fille (ou dérivée) n'a accès qu'aux membres publics et protected de la classe mère.

Membres d'instance et membres de classe (1/2)

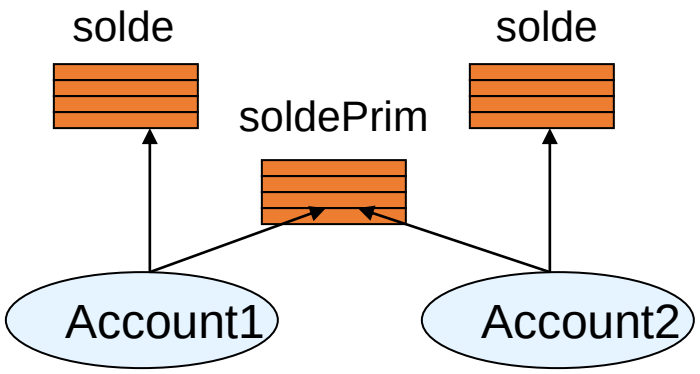
Le mot-clé « static »

- Chaque objet a sa propre "mémoire" de ses variables d'instance
- Le système alloue de la mémoire aux variables de classe dès qu'il rencontre la classe. Chaque instance possède la même valeur d'une variable de classe.

variable d'instance

variable de classe

```
class BankAccount {
        int solde;
        static int soldePrim;
        void deposit(int amount){
                solde+=amount;
                soldePrim+=amount;
        }
}
```

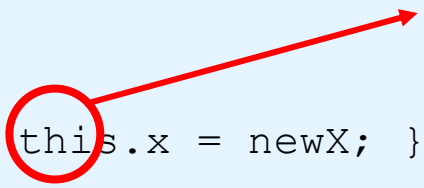


Membres d'instance et membres de classe (2/2)

Le mot-clé « static »

- Variables et méthodes statiques
  - Initialisées dès que la classe est chargée en mémoire
  - Pas besoin de créer un objet (instance de classe)
- Méthodes statiques
  - Fournissent une fonctionnalité à une classe entière
  - Cas des méthodes non destinées à accomplir une action sur un objet individuel de la classe
  - Exemples: `Math.random()`, `Integer.parseInt(String s)`, `main(String[] args)`
  - Les méthodes statiques ne peuvent pas accéder aux variables d'instances (elles sont "au-dessus" des variables d'instances)

```
class AnIntegerNamedX {
  int x;
  static public int x() { return x; }
  static public void setX(int newX) { this.x = newX; }
}
```

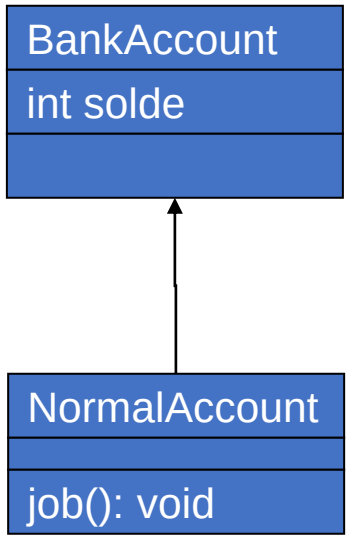


x est une variable d'instance, donc inaccessible pour la méthode static setX

Utilisation de l'héritage (1/5)

- Java n'offre pas la possibilité d'héritage multiple
- La « super super » classe, est la classe *Object* (parente de toute classe)
- Une sous-classe hérite des variables et des méthodes de ses classes parentes
- La clause *extends* apparaît dans la déclaration de la classe

```
class BankAccount {
  protected int solde;
  …
}
class NormalAccount extends BankAccount {
  public void job(){solde+=1000;}
}
```



Utilisation de l'héritage (2/5)

Cacher des données membres

- La variable aNumber du compte normal cache la variable aNumber de la classe générale compte en banque. Mais on peut accéder à la variable aNumber d'un compte en banque à partir d'un compte normal en utilisant le mot-clé super : `super.aNumber`

```
class BankAccount{
        Number aNumber;
}
class NormalAccount extends BankAccount{
        Float aNumber;
}
```

Utilisation de l'héritage (3/5)

Les mots-clé « this » et « super »

- Dans une méthode
  - « this » est une référence sur l'objet en cours lui-même
  - « super » permet d'accéder aux membres de la superclasse (peut être nécessaire en cas de redéfinition, par ex.)
- Dans le constructeur
  - Il existe toujours un constructeur. S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
  - « this » est toujours une référence sur l'objet en cours (de création) lui-même
  - « super » permet d'appeler le constructeur de la classe parent, ce qui est obligatoire si celui-ci attend des arguments

```
class MyClass{
  int x;
  MyClass(int x){
    this.x=x;// constructeur parent
  }
}
```

```
class Child extends MyClass {
  Child(){
    super(6); // appel du constructeur parent
  }
}
```

Utilisation de l'héritage (4/5)

Les mots-clé « this » et « super »

- En cas de surcharge du constructeur:

```
class Employee {
  String name,firstname;
  Address a;
  int age;
  Employee(String name,String firstname,Address a,int age){
    super();
    this. firstname= firstname;
    this.name=name;
    this.a=a;
    this.age=age;
  }
  Employee(String name,String firstname){
    this(name,firstname,null,-1);
  }
```

Utilisation de l'héritage (5/5)

Redéfinition de méthodes

- La redéfinition n'est pas obligatoire !! Mais elle permet d'adapter un comportement et de le spécifier pour la sous-classe.

```
class BankAccount {
        public void computeInterest(){
                solde+=300;                 //annual gift
}}
class NormalAccount extends BankAccount {
        public void computeInterest(){
                super.computeInterest();//call the overriden method
                solde*=1.07;                    //annual increase
}}
```

- Obligation de redéfinir les méthodes déclarées comme abstraites (*abstract*)
- Interdiction de redéfinir les méthode déclarées comme finales (*final*)

Polymorphisme (1/2)

Définition

- Concept basé sur la notion de redéfinition de méthodes
- Consiste à permettre à une classe de s'adresser à une autre en sollicitant un service générique qui s'appliquera différemment au niveau de chaque sous-classe du destinataire du message
- En d'autres termes, permet de changer le comportement d'une classe de base sans la modifier ➔ Deux objets peuvent réagir différemment au même appel de méthode
- Uniquement possible entre classes reliées par un lien d'héritage et suppose un cast « vers le haut » des objets des classes enfants

```
class Bank{
        BankAccount[] theAccounts = new BankAccount[10];
        public static void main(String[] args){
                theAccounts[0] = new NormalAccount("Joe",10000);
                theAccounts[0].computeInterest();
}}
```

Polymorphisme (2/2)

Utilisation du polymorphisme sur des collections hétérogènes

```
BankAccount[] ba=new BankAccount[5];

ba[0] = new NormalAccount("Joe",10000);
ba[1] = new NormalAccount("John",11000);
ba[2] = new SpecialAccount("Jef",12000);
ba[3] = new SpecialAccount("Jack",13000);
ba[4] = new SpecialAccount("Jim",14000);

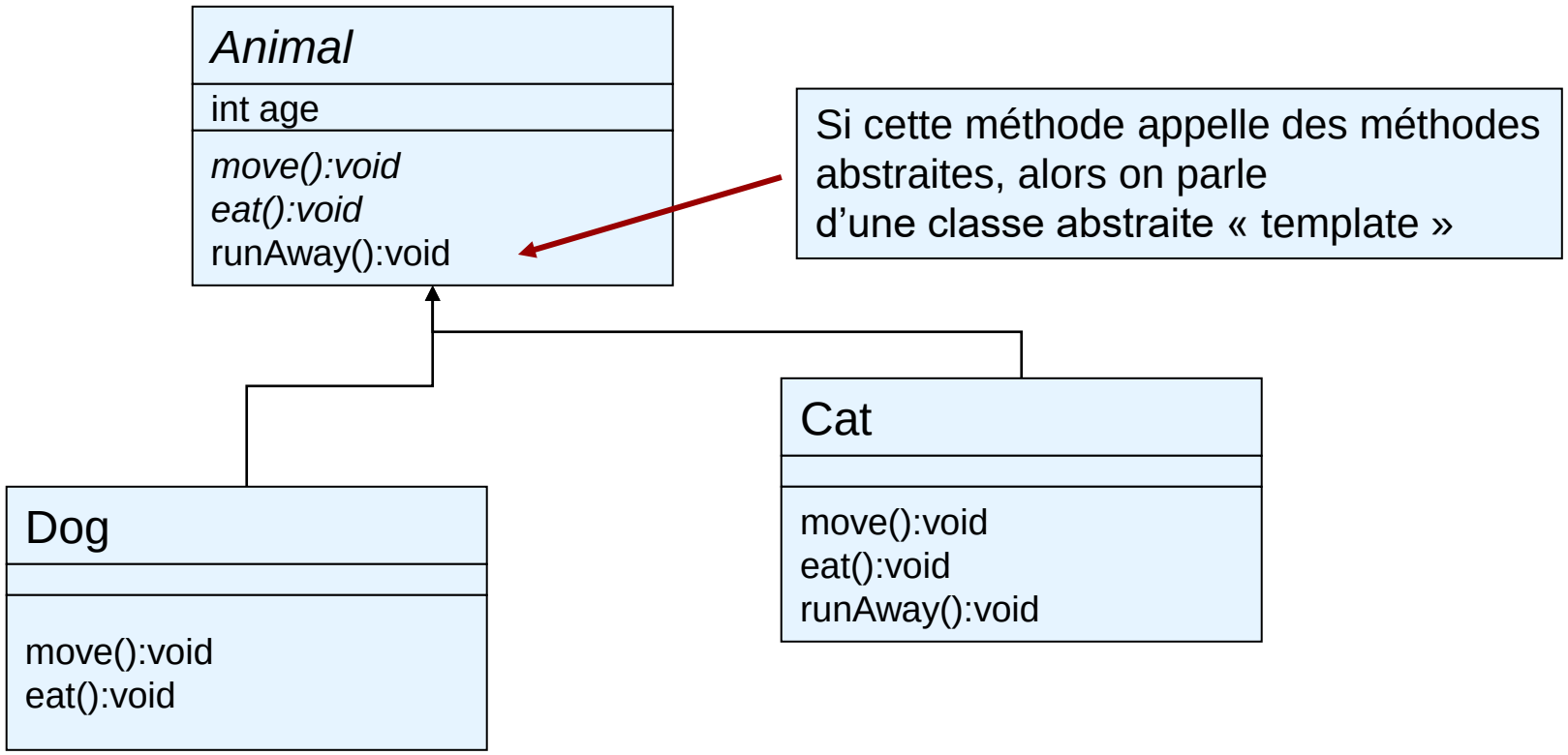
for(int i=0;i<ba.length();i++)
{
  ba[i].computeInterest();
}
```

Les classes abstraites (1/2)

- Une classe abstraite
  - Peut contenir ou hériter de méthodes abstraites (des méthodes sans corps)
  - Peut contenir des constantes globales
  - Peut avoir des méthodes normales, avec corps
- Une classe abstraite ne peut être instanciée
  - On peut seulement instancier une sous-classe concrète
  - La sous-classe concrète doit donner un corps à toute méthode abstraite
- La déclaration d'une classe abstraite contenant une méthode abstraite ressemble à ceci:

```
abstract class Animal {
   abstract void move();
}
```

Les classes abstraites (2/2)



Les interfaces (1/5)

Définition

- Une interface peut être considérée comme une classe 100% abstraite.
- Une interface décrit une fonctionnalité sous forme d'une liste de méthodes mais ne fournit aucune implémentation de ces méthodes.
- Une interface ne peut contenir de variable d'instance.
- Les seuls attributs autorisés sont les constantes déclarées static et final.
- Une interface ne peut être instanciée.Une classe peut implémenter une interface, ou bien des interfaces multiples

```
public interface Runnable {
  public void run();
}
```

```
public interface GraphicalObject {
  public void draw(Graphics g);
}
```

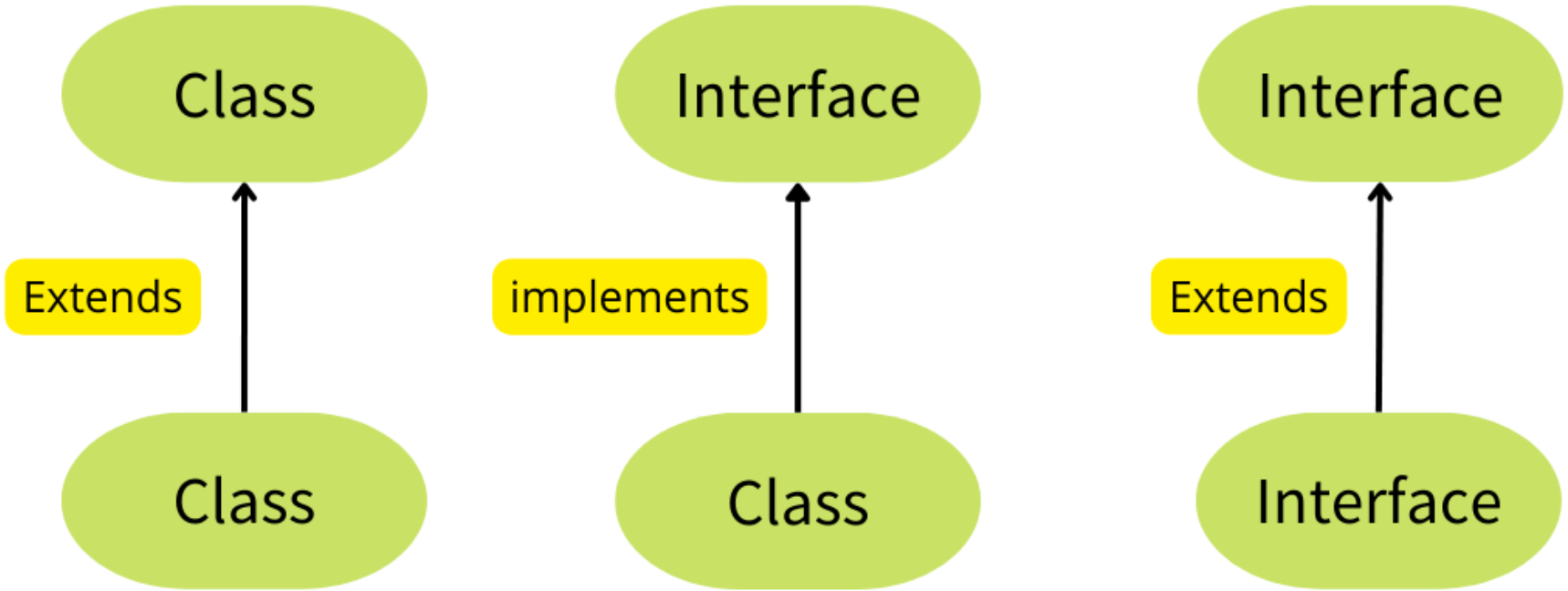
Les interfaces (2/5)

Définition

- Toutes les méthodes d'une interface sont implicitement publiques.
- Une interface peut étendre plusieurs interfaces.
- A partir de Java 8, on peut ajouter deux éléments supplémentaires dans une interface:
  - des méthodes statiques
  - des méthodes par défaut qui sont des méthodes concrètes.

Les interfaces (3/5)

Définition



Les interfaces (4/5)

Raisons d'être

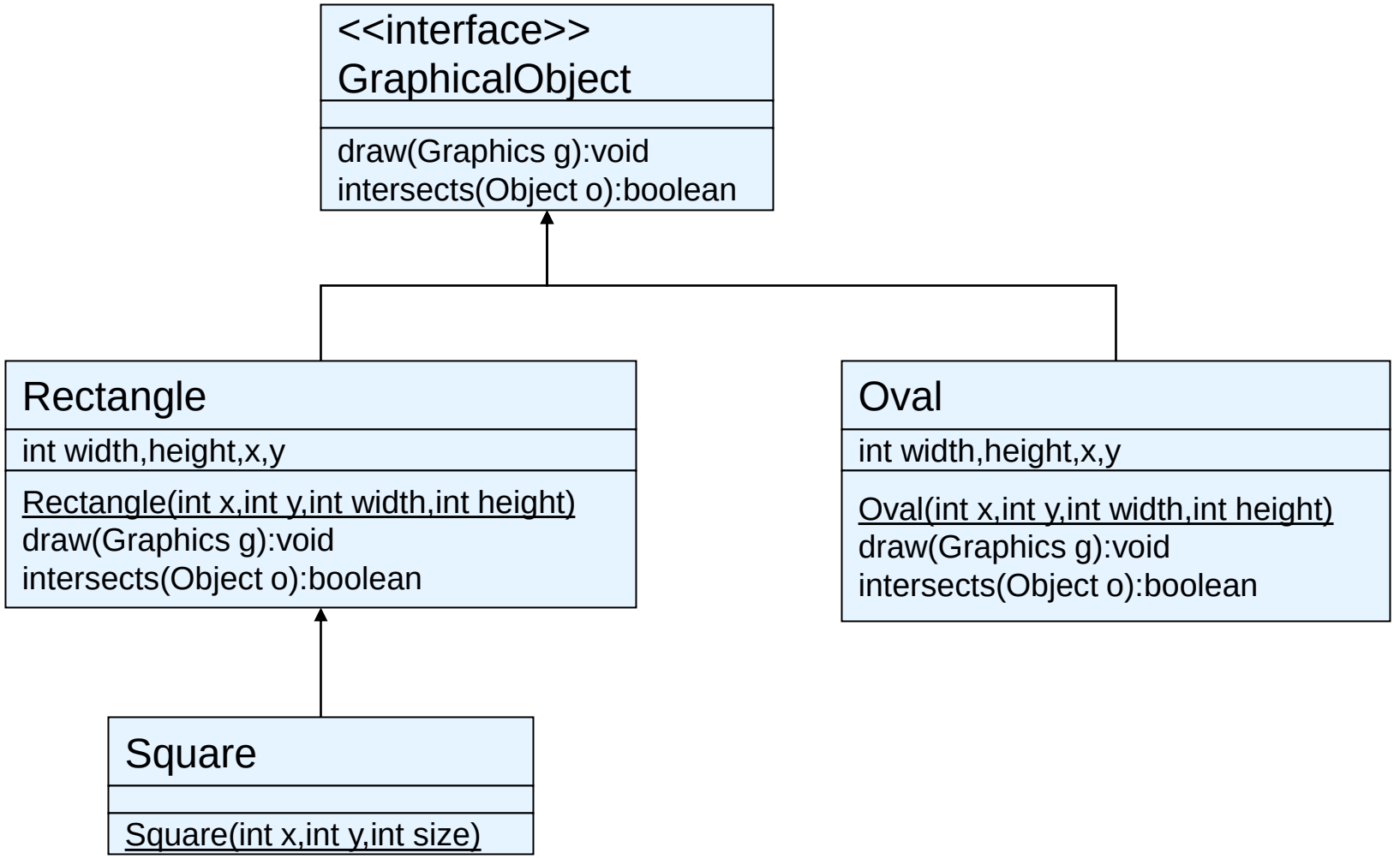
- Forcer la redéfinition / l'implémentation de ses méthodes
- Permettre une certaine forme de multi-héritage
- Faciliter et stabiliser la décomposition de l'application logicielle
- D'une classe qui dérive d'une interface, on dit qu'elle implémente cette interface
- Le mot clé associé est donc logiquement: `implements`

- Exemple:

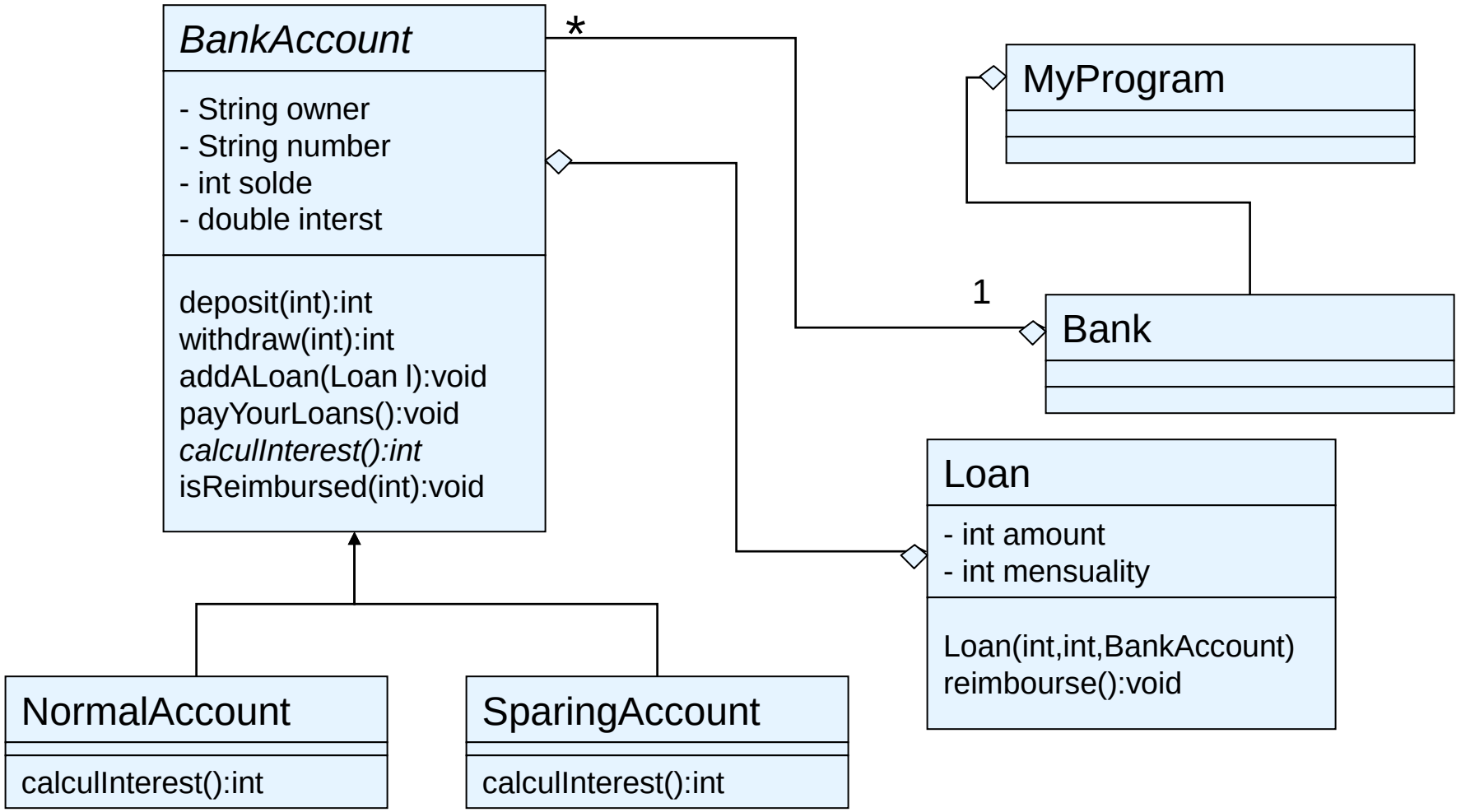
```
public class monApplet extends Applet implements Runnable, KeyListener
```

Les interfaces (5/5)

Exemple



Exercice


BankAccount
- String owner
- String number
- int solde
- double interst
deposit(int):int
withdraw(int):int
addALoan(Loan l):void
payYourLoans():void
calculInterest():int
isReimbursed(int):void
*
MyProgram
1
Bank
Loan
- int amount
- int mensuality
Loan(int,int,BankAccount)
reimbourse():void
NormalAccount
calculInterest():int
SparingAccount
calculInterest():int

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Interfaces graphiques et Gestion d'événements

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Plan Chapitre 1: Introduction à Java

1. **Les interfaces graphiques (GUI):**
   a. AWT vs SWING vs SWT vs JavaFX
   b. La structure de Swing
   c. Les « Components »
   d. Les « Containers »
   e. Les « LayoutManagers »
2. **Gestion d'événements**
   a. Mécanismes et structure
   b. Mise en œuvre

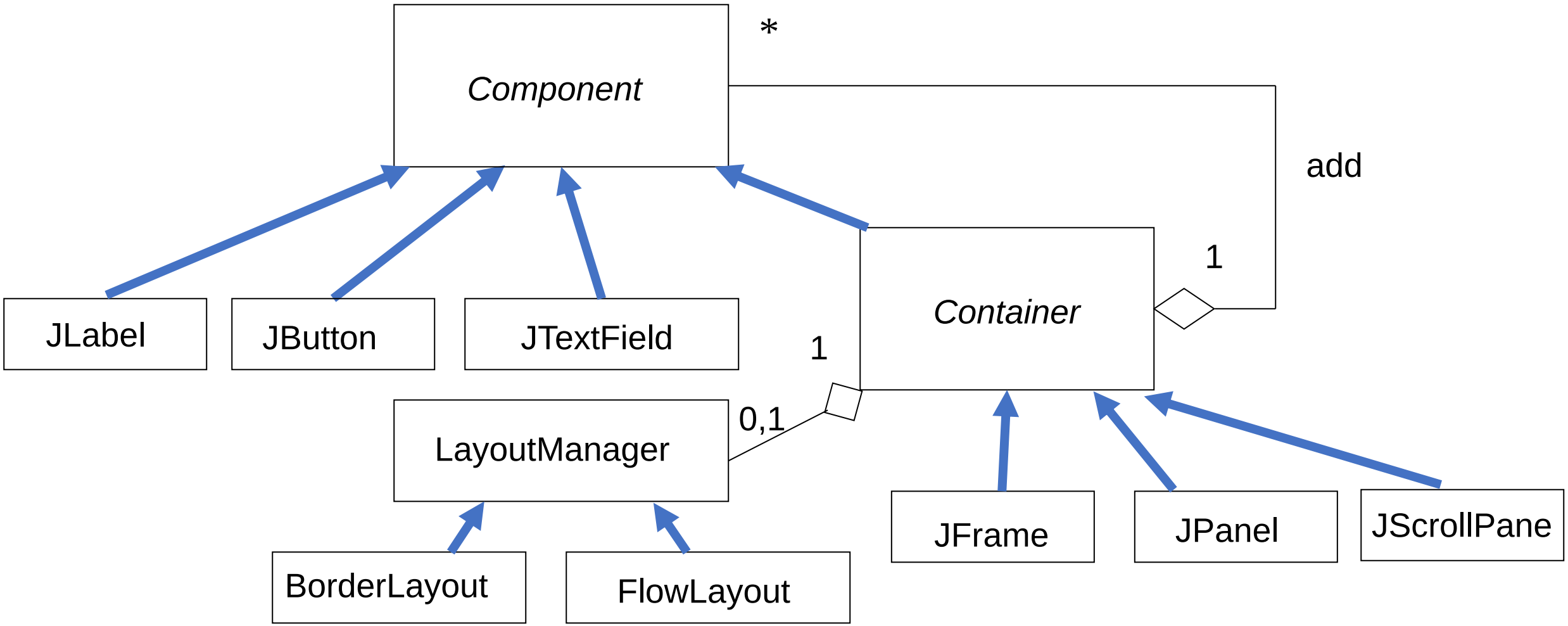
AWT vs SWING vs SWT vs JavaFX

- L'AWT (Abstract Window Toolkit)
  - AWT utilise les composants natifs du système d'exploitation, ce qui rend les interfaces dépendantes de la plateforme.
  - Il est plus limité que Swing ou JavaFX en termes de composants et de personnalisation.
- L'API Swing
  - Offre une large gamme de composants : boutons, boîtes de dialogue, listes déroulantes, tables, arbres, etc.
  - Supporte les interfaces graphiques complexes et personnalisées.
  - Les composants sont lourds (heavyweight), car ils sont dessinés par Java plutôt que par le système d'exploitation.
  - Il est possible de personnaliser les composants Swing avec des LookAndFeel pour changer l'apparence de l'interface.
- JavaFX : la dernière née des API graphiques proposées par le Java SE
  - Supporte les animations, les effets graphiques, et les interfaces utilisateur riches.
  - Utilise le langage FXML pour créer des interfaces graphiques de manière déclarative, semblable à HTML pour les applications web.
  - Supporte les graphiques vectoriels, les vidéos, l'audio, et le rendu 2D/3D.
  - Permet la création d'interfaces graphiques plus fluides et modernes.
- SWT : une autre librairie de mise en œuvre d'interfaces graphiques
  - Utilisé principalement par Eclipse IDE pour ses interfaces graphiques.
  - SWT utilise des composants natifs du système d'exploitation, ce qui donne un rendu plus natif mais peut rendre le déploiement plus complexe.

Structure de Swing (1/2)

- Le Swing offre trois types d'éléments graphiques
  - Les « Containers » (contenants)
  - Les « Components » (composants ou contenus)
  - Les « LayoutManagers » (disposition des objets d'un contenant)
- Les « Containers »
  - Sont destinés à accueillir des composants
  - Gèrent l'affichage des composants
  - *Ex*: JFrame, JPanel,…
- Les « Components »
  - Constituent différents éléments de l'affichage (boutons, barres de menus, etc.)
  - *Ex*: Button, Canvas, Label, Scrollbar, Checkbox
- Les « LayoutManagers »
  - Gèrent la disposition des composants au sein d'un conteneur

Structure de Swing (2/2)



Les « Components »

Les composantes de Java Swing peuvent être classés en deux classes :

- **Composants conteneurs** : Les conteneurs sont des composants qui contiennent d'autres composants Swing.
- **Composants de contenu** : Ces composants sont utilisés pour afficher du contenu ou interagir avec l'utilisateur.
  - **Composants de fenêtre :** Ce sont des composantes pour les conteneurs de type JFrame.

Les « Components »

Composants conteneurs :

- **JFrame :** C'est une fenêtre principale avec des décorations standard comme une barre de titre, des bordures et des boutons de contrôle (minimiser, agrandir, fermer).

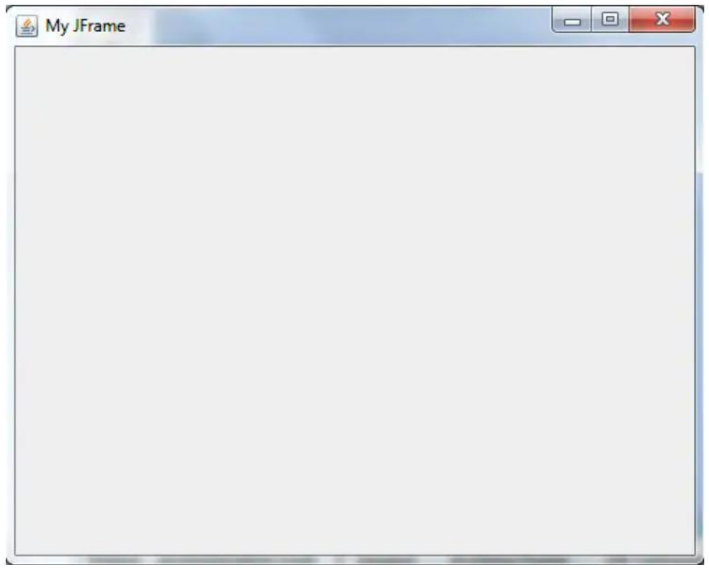
```
import javax.swing.JFrame;
public class MyFrame extends JFrame {

   public MyFrame() {
        this.setTitle("MyFrame");



        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        this.setSize(400, 400);
        this.setLocationRelativeTo(null);
        this.setVisible(true);
   }
}
```



Les « Components »

Composants conteneurs :

- **JDialog** : Utilisé pour créer des boîtes de dialogue. Un JDialog est une fenêtre modale ou non modale qui peut être utilisée pour obtenir des informations de l'utilisateur ou afficher des messages.

```
JDialog dialog = new JDialog(parent, "Mon JDialog", true);
dialog.setSize(250, 150);
dialog.setLocationRelativeTo(parent);
```

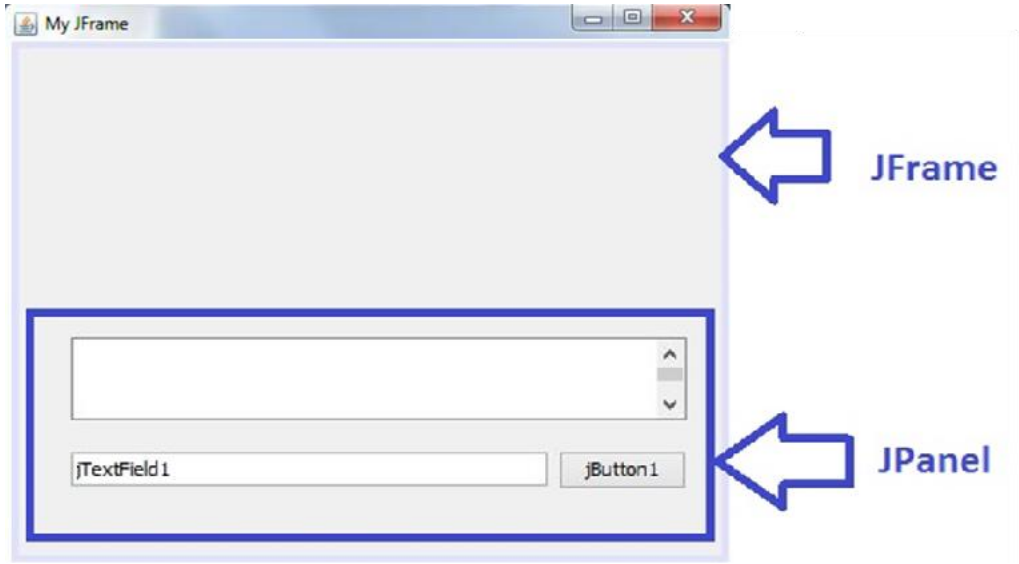


Les « Components »

Composants conteneurs :

- **JPanel** : Un conteneur général qui permet de regrouper des composants sous un même panneau. On peut l'utiliser pour diviser une interface en sections.

```
JPanel p = new JPanel();
p.setLyout(new BorderLayout());
```

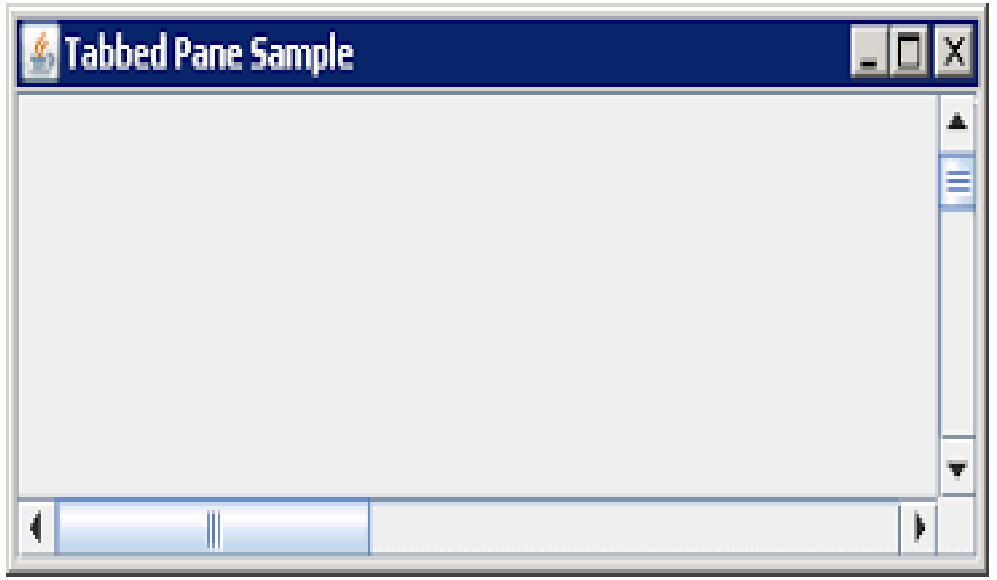


Les « Components »

Composants conteneurs :

- **JScrollPane** : Permet de rendre un autre composant, comme un JTextArea ou JList, défilable en cas de contenu trop grand.

```
JScrollPane p = new JScollPane();
p.setLyout(new BorderLayout());
```



Les « Components »

Composants de contenu:

- **JLabel :** Affiche du texte ou une image.

```
JLabel p = new JLabel("First Label");
```

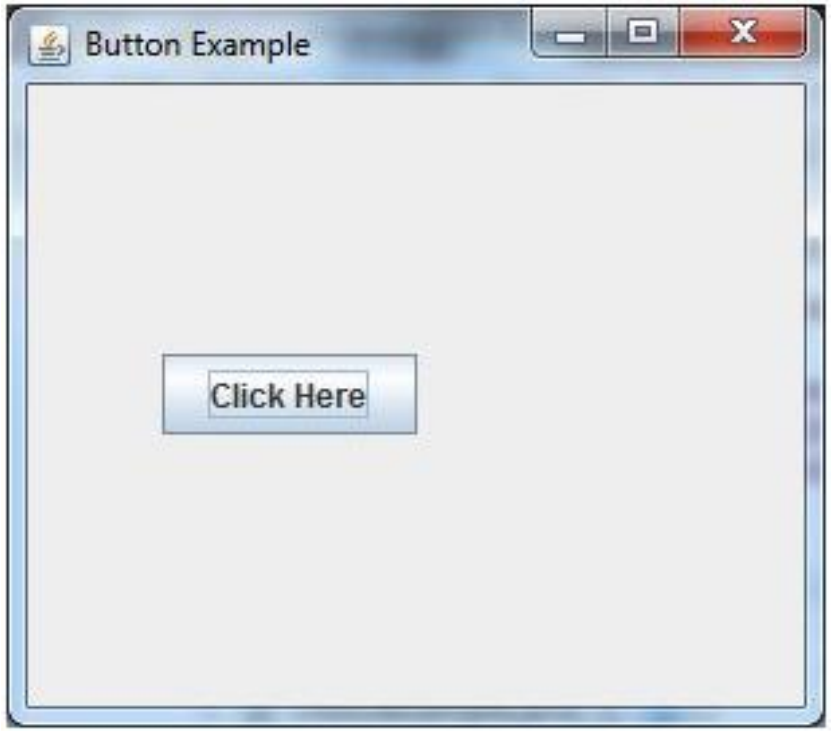


Les « Components »

Composants de contenu:

- **JButton :** Un bouton sur lequel l'utilisateur peut cliquer pour effectuer une action.

```
JButton p = new JButton("Click Here");
```

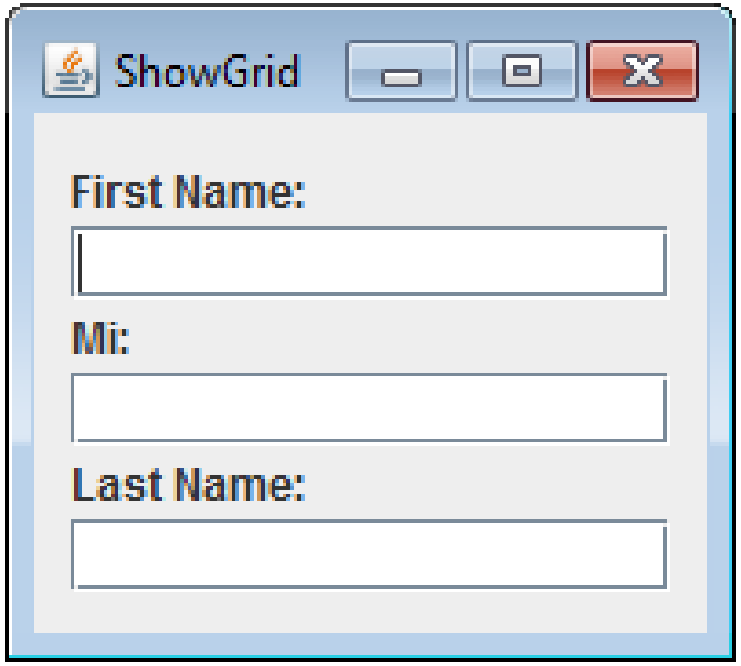


Les « Components »

Composants de contenu:

- **JTextField :** Un champ de texte permettant à l'utilisateur de saisir une ligne de texte.

```
JTextField nameTextField = new JTextField();
```



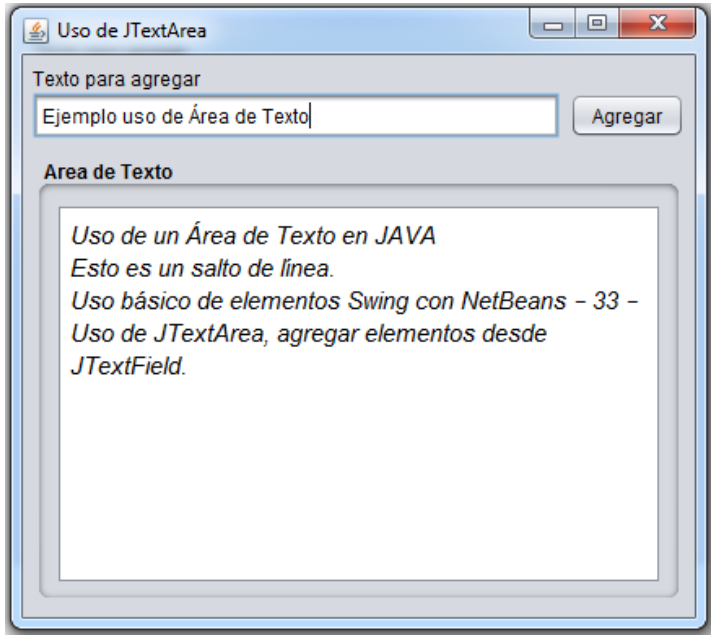
Les « Components »

Composants de contenu:

- **JTextArea :** Une zone de texte qui permet de saisir un long texte sur plusieurs lignes.

```
JTextArea nameTextField = new JTextArea();

// Spécifier le nombre de lignes et de colonnes
JTextArea nameTextField = new JTextArea(10, 30);
```



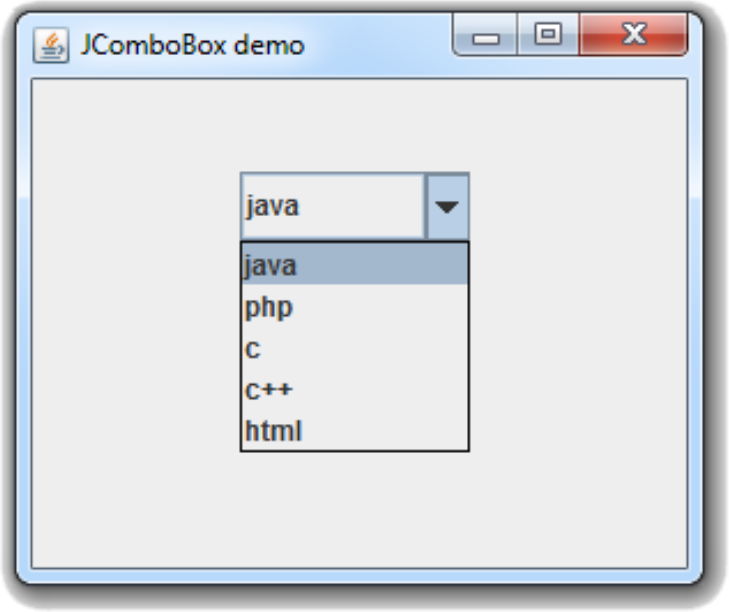
Les « Components »

Composants de contenu:

- **JComboBox :** Un composant de liste déroulante permettant à l'utilisateur de sélectionner une option parmi plusieurs.

```
String[] options = {"Java", "php", "c", "c++"};
JComboBox p = new JComboBox(options);

// Ajouter d'autres items
p.addItem("html");
```



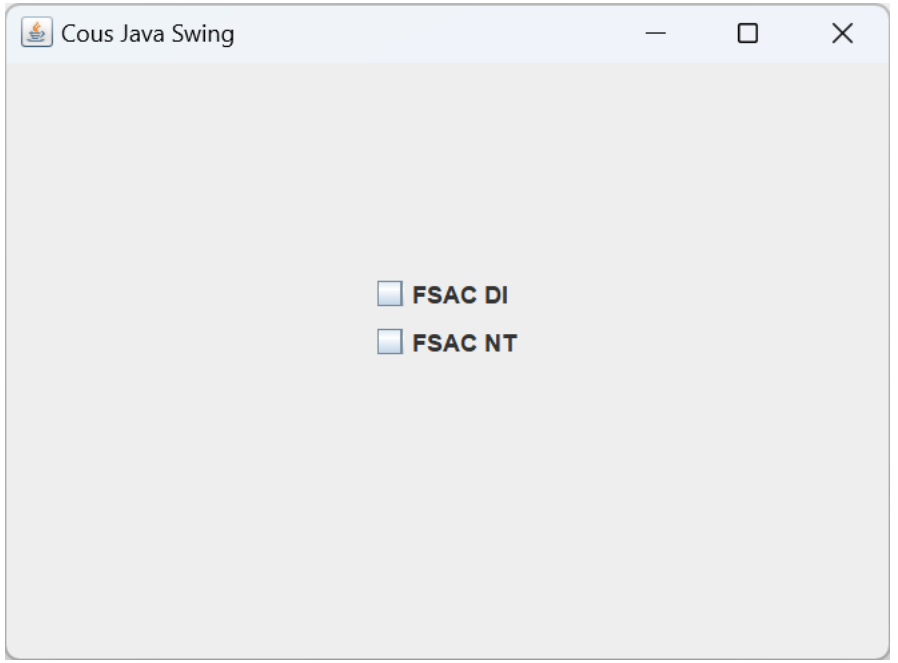
Les « Components »

Composants de contenu:

- **JCheckBox** : un élément qui peut être sélectionné ou désélectionné et qui affiche son état à l'utilisateur

```
JCheckBox p = new JCheckBox("FSAC DI");

// Pour vérifier si sélectionnée
p.isSelected();
```



Les « Components »

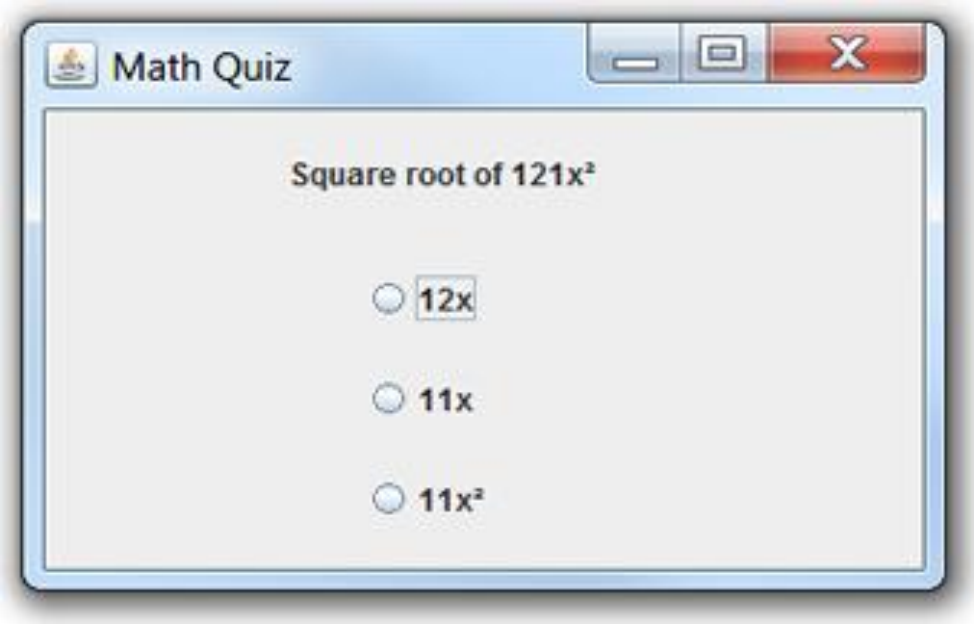
Composants de contenu:

- **JRadioButton** : Un bouton radio qui fait partie d'un groupe où une seule option peut être sélectionnée à la fois.

```
JRadioButton p = new JRadioButton("12x");

// Pour vérifier si sélectionnée
p.isSelected();

ButtonGroup group = new ButtonGroup();
group.add(option1);
group.add(option2);
group.add(option3);
```



Les « Components »

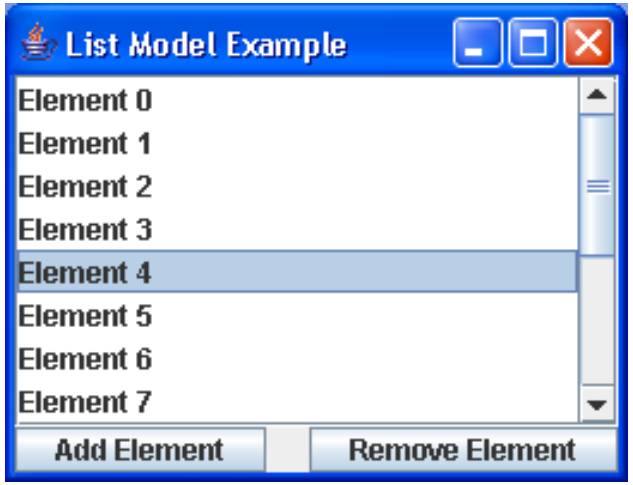
Composants de contenu:

- **JList** : Affiche une liste d'éléments parmi lesquels l'utilisateur peut sélectionner

```
DefaultListModel<String> listModel = new DefaultListModel<>();
for(int i=0; i<8; i++)
     listModel.addElement("Element "+i);
JList<String> list = new JList<>(listModel);

JScrollPane p = new JScrollPane(list);
contentPane.add(p);

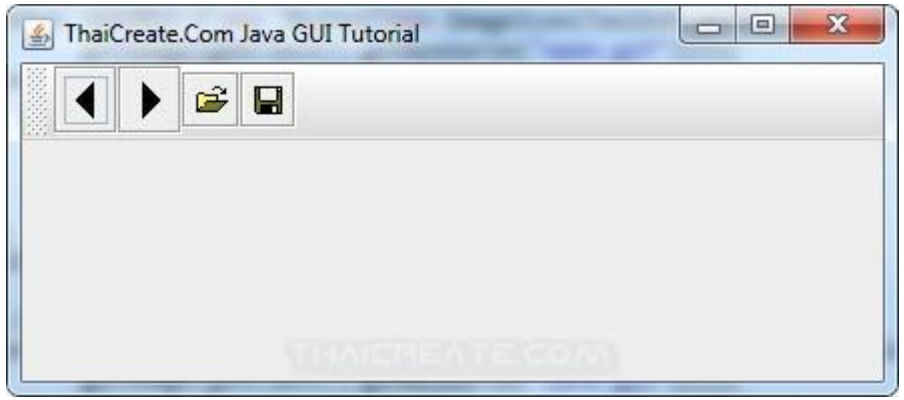
// Ajouter un élément
listModel.addElement("Element 8");
```

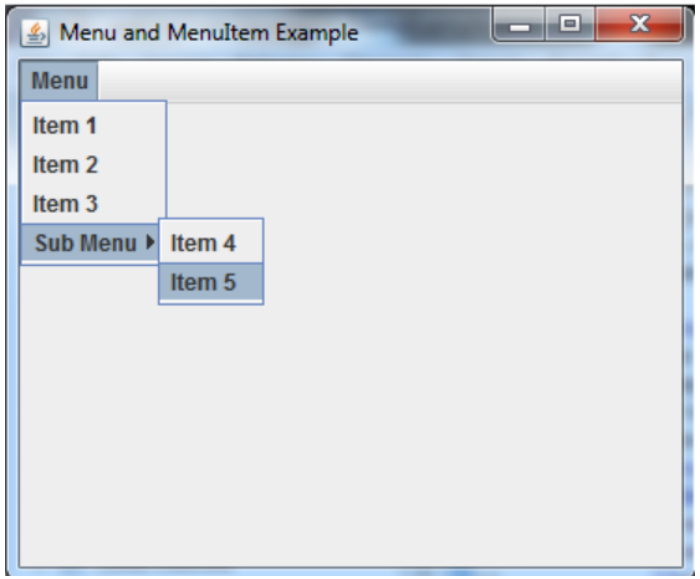


Les « Components »

Composants de fenêtre:

- **JToolBar** : Une barre d'outils qui contient des boutons pour des actions courantes dans l'application.
- **JMenuBar, JMenu, JMenuItem** : Composants pour créer des menus.





Les « LayoutManagers » (1/11)

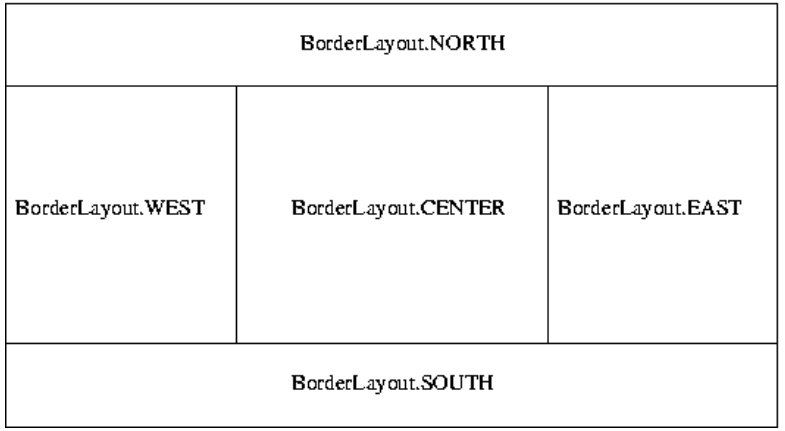
- Rôle :
  - Gérer la disposition des composants au sein d'un conteneur

- Types principaux:
  - BorderLayout: divise le conteneur en 5 zones
  - FlowLayout: rajoute les composants au fur et à mesure
  - GridLayout: applique une grille au conteneur pour aligner les composants
  - CardLayout: pour un conteneur qui contient plusieurs cartes
  - GridBagLayout: grille de cellules élémentaires

Les « LayoutManagers » (2/11)

BorderLayout

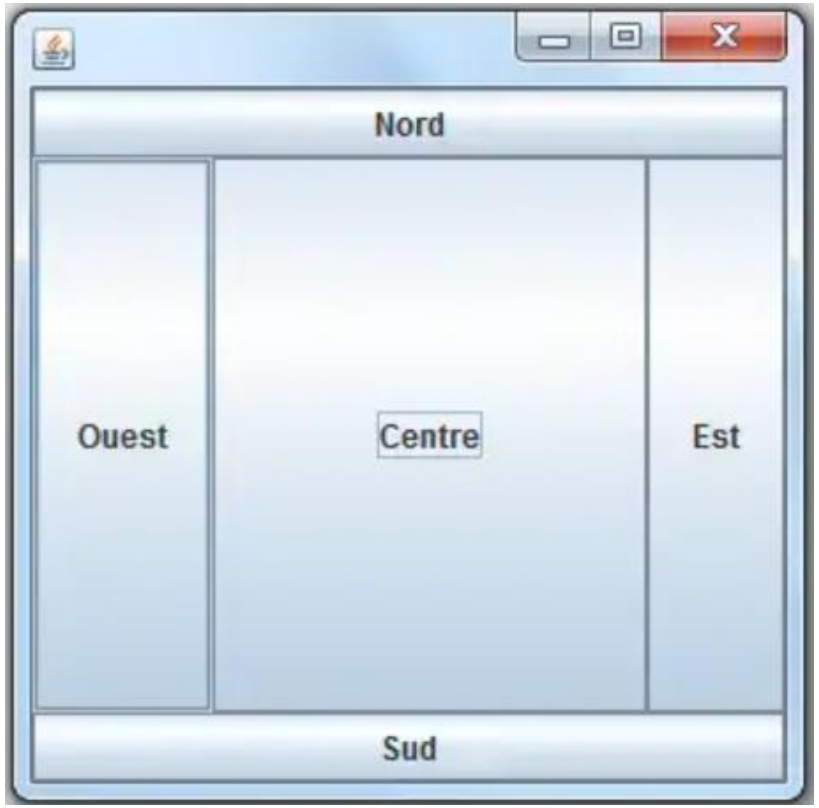
- Principe
  - Positionne les composants suivants les points cardinaux
  - Layout par défaut des « Frame »
- Répartition
  - La largeur l'emporte pour le Nord et le Sud
  - La hauteur l'emporte pour l'Est et l'Ouest
  - Le centre occupe tout ce qui reste
- Utilisation
  - add(unBouton, BorderLayout.NORTH)
  - new BorderLayout(int, int) : Le constructeurs accepte les intervalles horizontal et vertical entre les éléments



Les « LayoutManagers » (3/11)

BorderLayout :

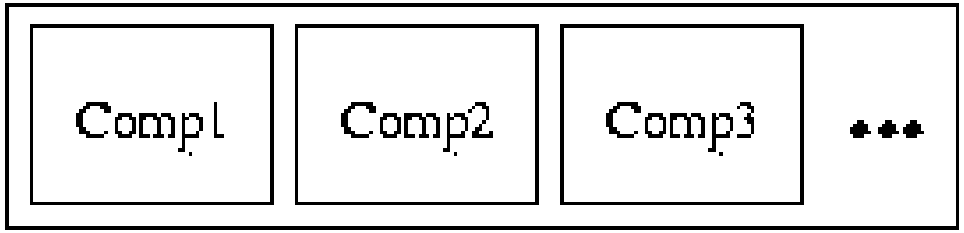
```
JButton btn1 = new JButton("Nord");
JButton btn2 = new JButton("Sud");
JButton btn3 = new JButton("Est");
JButton btn4 = new JButton("Ouest");
JButton btn5 = new JButton("Centre");
contentPane.add(btn1, BorderLayout.NORTH);
contentPane.add(btn2, BorderLayout.SOUTH);
contentPane.add(btn3, BorderLayout.EAST);
contentPane.add(btn4, BorderLayout.WEST);
contentPane.add(btn5, BorderLayout.CENTER);
```



Les « LayoutManagers » (4/11)

FlowLayout :

- Principe
  - Rajoute les composants au fur et à mesure
  - La taille des composants l'emporte

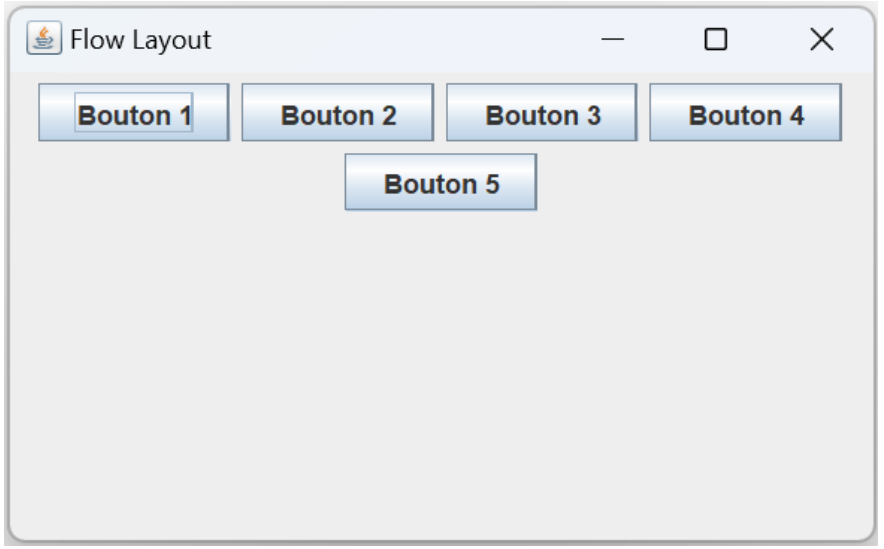


- Utilisation
  - new FlowLayout (int alignment)
    - alignment: LEFT, CENTER, RIGHT
  - new FlowLayout (int alignment, int hintervalle, int vintervalle)
    - alignment: LEFT, CENTER, RIGHT
    - hintervalle: L'intervalle horizontal entre les composants
    - vintervalle: L'intervalle vertical entre les composants

Les « LayoutManagers » (5/11)

FlowLayout :

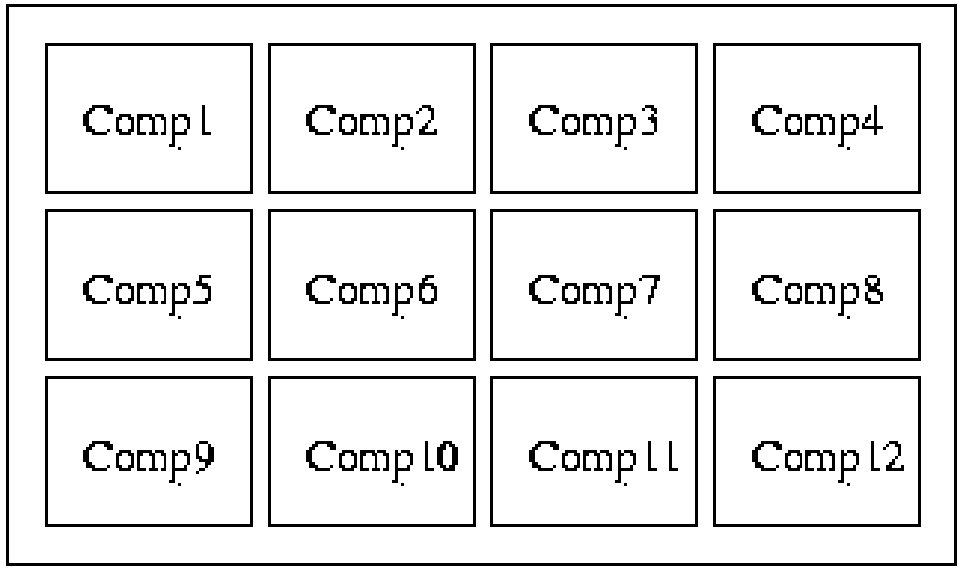
```
contentPane.setLayout(new FlowLayout());
this.add(new JButton("Bouton 1"));
this.add(new JButton("Bouton 2"));
this.add(new JButton("Bouton 3"));
this.add(new JButton("Bouton 4"));
this.add(new JButton("Bouton 5"));
```



Les « LayoutManagers » (6/11)

GridLayout :

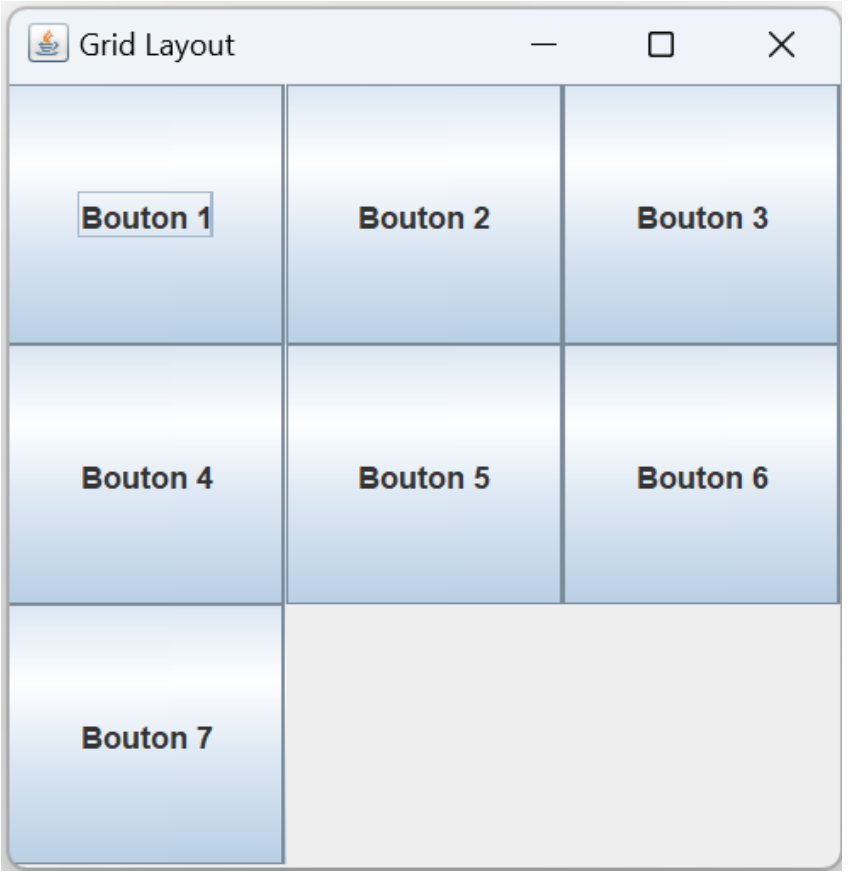
- Principe
  - Définit une grille a 2 dimensions
- Utilisation
  - new GridLayout (int, int)
    - Définit le nombre de lignes et de colonnes
  - new GridLayout (int, int, int, int)
    - Définit le nombre de lignes et de colonnes
    - Définit l'alignement horizontal et vertical



Les « LayoutManagers » (7/11)

GridLayout :

```
contentPane.setLayout(new GridLayout(3, 3));
this.add(new JButton("Bouton 1"));
this.add(new JButton("Bouton 2"));
this.add(new JButton("Bouton 3"));
this.add(new JButton("Bouton 4"));
this.add(new JButton("Bouton 5"));
this.add(new JButton("Bouton 6"));
this.add(new JButton("Bouton 7"));
```



Les « LayoutManagers » (8/11)

CardLayout :

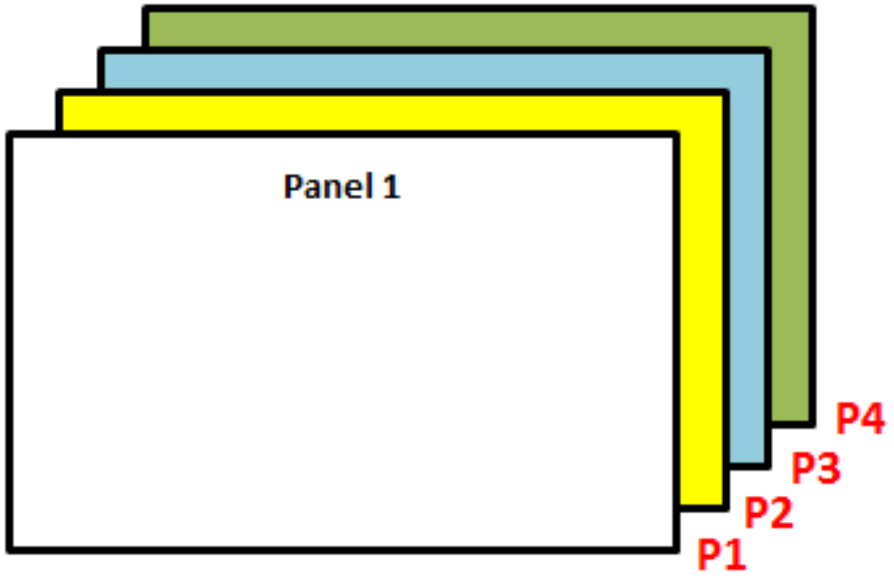
- Principe
  - Pour un container qui contient plusieurs cartes
  - Seule une carte est visible à chaque fois
  - L'utilisateur peut passer d'une carte à l'autre

Les « LayoutManagers » (9/11)

CardLayout :

```
CardLayout cardLayout = new CardLayout()
contentPane.setLayout(cardLayout);
JPanel card1 = new JPanel();
JPanel card2 = new JPanel();
JPanel card3 = new JPanel();
contentPane.add(card1);
contentPane.add(card2);
contentPane.add(card3);

cardLayout.next(contentPane);
```



Les « LayoutManagers » (10/11)

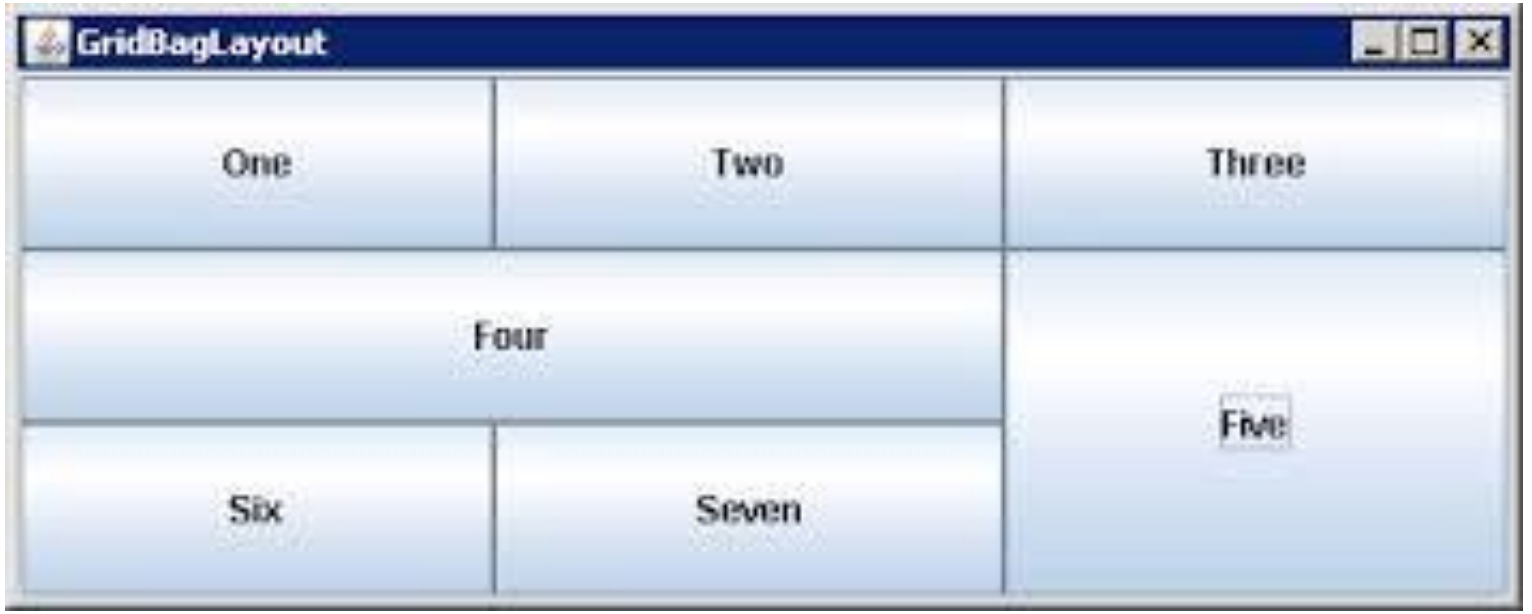
GridBagLayout :

- Principe
  - Le plus compliqué et le plus flexible
  - Une grille de cellules élémentaires
  - Les composants graphiques peuvent s'étaler indifféremment sur ces cellules
  - Pour positionner un composant, il faut utiliser un objet GridBagConstraints
  - On peut spécifier le comportement des composants quand on étire la fenêtre

- Utilisation
  - new GridBagLayout()
  - new GridBagConstraints();

Les « LayoutManagers » (11/11)

GridBagLayout :



Gestion d'événements : Mécanismes et structure (1/4)

- Une source d'événements
  - Génère des objets événements
  - Les fait écouter par un ensemble d'écouteurs d'événements
  - En général: un composant ou conteneur graphique
- Les objets événements
  - xxxEvent
  - Contiennent la description et les caractéristiques d'un événement
- Les objets écouteurs
  - xxxListener ou xxxAdapter
  - Concrétisent des méthodes définies dans les Interfaces
  - Indiquent leur réaction en réponse aux événements
  - Sont des interfaces implémentables dans les classes
  - Peuvent être implémentés par les sources d'événements elles-mêmes (Une source d'événements peut « s'écouter » elle-même)

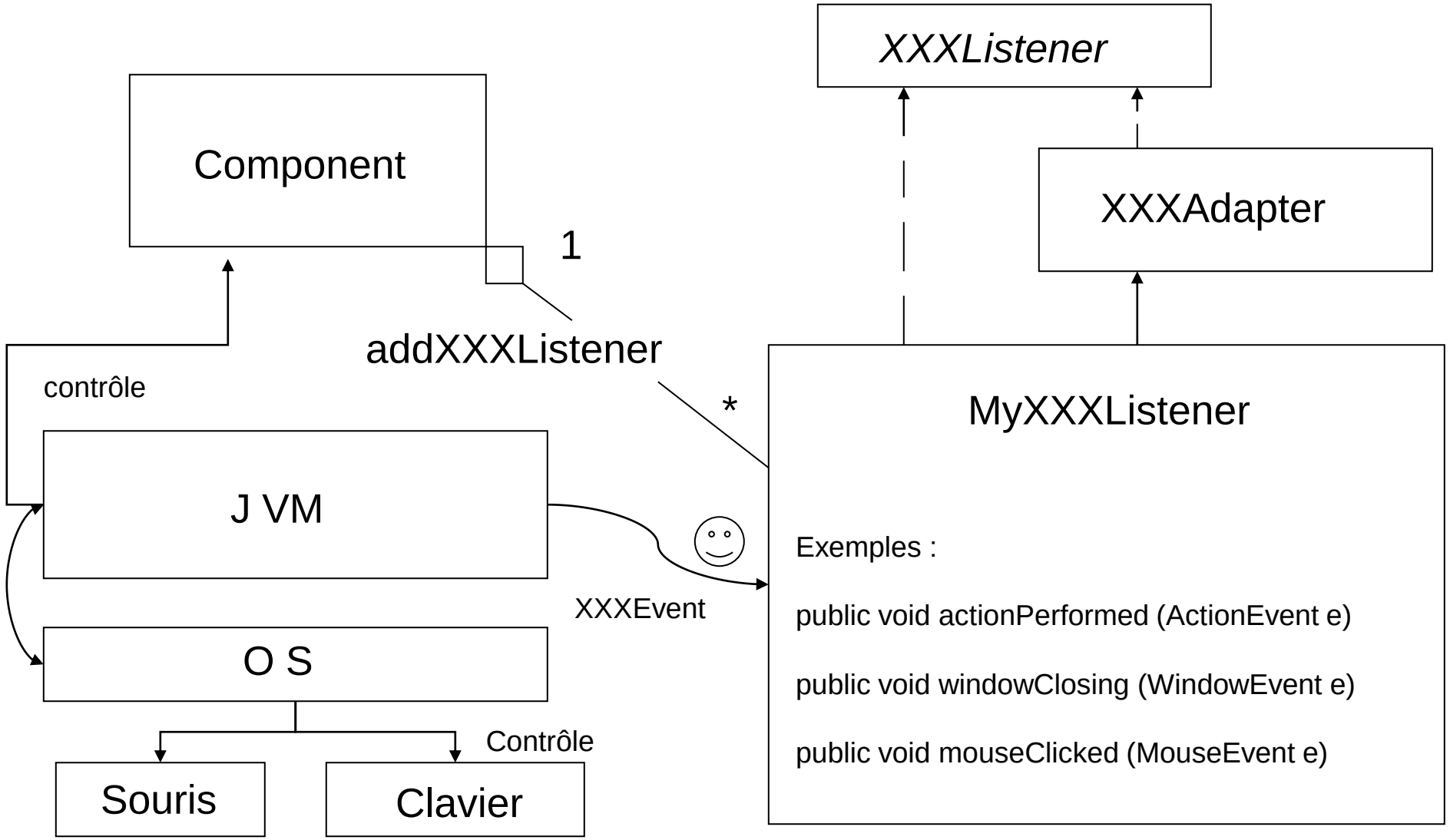
Gestion d'événements : Mécanismes et structure (2/4)

1. Un événement se produit
2. La source d'événement dans laquelle il se produit génère un objet de type événement
3. La source transmet l'événement à son (ses) écouteur(s)
4. L'écouteur appelle la méthode correspondant au type d'événement et lui passe en argument l'objet événement
5. La méthode en question spécifie les traitements à réaliser lorsqu'un événement du type correspondant se produit
6. Dans ses traitements, la méthode peut examiner les caractéristiques de l'événement (position du curseur de la souris, code de la touche pressée au clavier…) et adapter son comportement en fonction de ces caractéristiques

Gestion d'événements : Mécanismes et structure (3/4)

- Les Sources d'événements
  - Button, List, MenuItem, TextField, ScrollBar, CheckBox, Component, Container, Window
- Evénements
  - ActionEvent, KeyEvent , MouseEvent , WindowEvent , FocusEvent , AdjustmentEvent, ComponentEvent, ContainerEvent,, ItemEvent, TextEvent, …
- Les Interfaces Ecouteurs
  - ActionListener, KeyListener , MouseListener , WindowListener , FocusListener , AdjustmentListener, ComponentListener, ContainerListener,, ItemListener, MouseMotionListener,
- Les Adapteurs correspondants
  - ActionAdapter, WindowAdapter, KeyAdapter, MouseAdapter, ...

Gestion d'événements : Mécanismes et structure (1/4)



Gestion d'événements : Mise en oeuvre

- Par implémentation de l'interface
  - Usage
    - ➔ `public class MaClasse implements MouseListener`
  - Avantages et inconvénients
    - ☺ Meilleur sur le plan orienté objet
    - ☺ La classe peut hériter d'une autre classe
    - ☺ Consistance
- Par héritage de l'adapteur
  - Usage
    - A chaque interface correspond un adapteur qui l'implémente lui-même
    - ➔ `public class MaClasse extends MouseAdapter`
  - Avantages et inconvénients
    - ☺ Code simple (l'adapteur redéfinit déjà les méthodes, etc.)
    - ☹ La classe ne peut plus hériter d'une autre classe

Gestion d'événements : Mise en œuvre – Click de souris

```
class MonFrame extends Frame implements MouseListener
{
  public MonFrame()
  {
    addMouseListener(this);
  }
  public void mousePressed(MouseEvent e) {}
  public void mouseClicked(MouseEvent e)
  {
    if(e.getX()>50 && e.getY()<100){...}
  }
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited{MouseEvent e) {}
}
```

Gestion d'événements : Mise en œuvre – Déplacement de la souris

```
class MonFrame extends Frame implements MouseMotionListener
{
  public MonFrame()
  {
    addMouseMotionListener(this);
  }
  public void mouseDragged(MouseEvent e) {}
  public void mouseMoved(MouseEvent e)
  {
    if(e.getX()>50 && e.getY()<100){...}
  }
}
```

Gestion d'événements : Mise en œuvre – Clavier

```
class MonFrame extends Frame implements KeyListener
{
  public MonFrame()
  {
    addKeyListener(this);
  }
  public void keyPressed(KeyEvent e) {}
  public void keyTyped(KeyEvent e)
  {
    if(e.getKeyCode()==KeyEvent.VK_Q){System.exit(0)}
  }
  public void keyReleased(KeyEvent e) {}
}
```

Gestion d'événements : Mise en œuvre – Evènement de la fenêtre

```
class MonFrame extends Frame implements WindowListener
{
  public MonFrame()
  {
    addWindowListener(this);
  }

  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {System.exit(0);}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}
```

Gestion d'événements : Mise en œuvre – Actions

```
public class MonPanel extends Panel implements ActionListener
{                          // MonPanel devient un écouteur d'événements
  public void actionPerformed(Action evt)
  {     // Ce qui est fait en réponse à l'événement….
        // je peux extraire de l'information sur l'événement
        // transmis par evt
        // je suis obligé de re-définir cette méthode abstraite
        // dans ActionListener
  }
}


MonPanel panel = new MonPanel();
JButton button = new Jbutton("OK");// Crée un bouton qui est source
                                   // d'évènements de type ActionEvent
button.addActionListener(panel)    // Associer un écouteur
                                   // d'événements panel au bouton
```

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Gestion des Collections

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

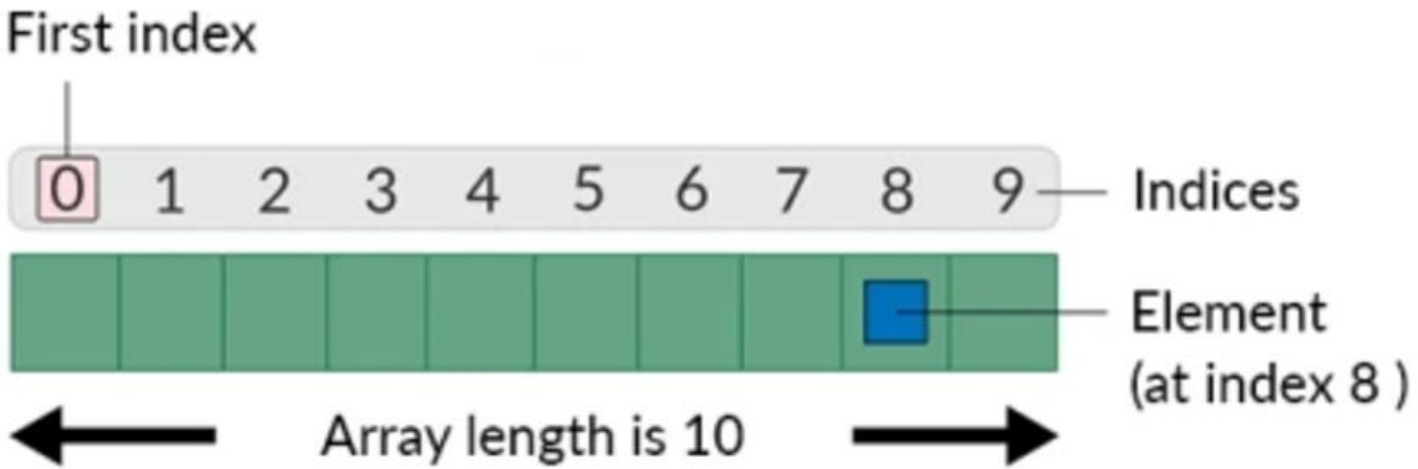
Plan Chapitre : Collections et tableaux

- Introduction
  - Les tableaux unidimensionnelles et multidimensionnels.
  - Qu'est-ce qu'une Collection?
  - Le Java Collections Framework.
- Interfaces
  - Collections ➔ « Set » et « List »
  - Maps ➔ « Map » et « SortedMap »
  - Comparateurs ➔ « Comparable » et « Comparator »
- Implémentations
  - HashSet et TreeSet
  - ArrayList et LinkedList
- Algorithmes
  - Tri
  - Autres: Recherche, Mélange, etc.

Introduction

Les tableaux : Définition

En Java, un tableau (tableau) est une **structure de données** contenant un groupe d'éléments tous du **même type**, avec des adresses consécutives sur la mémoire (memory). Le tableau **a le nombre fixé d'éléments et vous ne pouvez pas changer sa taille**.



Les tableaux : Déclaration

Syntaxe :

```
type   nomTableau[] = new type[taille];
type[]   nomTableau = new type[taille];
type   nomTableau[] = { liste des valeurs };
```

Exemples:

```
int A[ ] = new int[10];   // un  tableau d'entiers de taille 10
double B[ ] = new double[20];   // un  tableau de réels de taille 20
boolean C[ ] = new boolean[5];   // un  tableau de booléen de taille 5
int D[ ] ={7, 3, 9, 3};   // un  tableau d'entiers initialisé par 4 valeurs
```

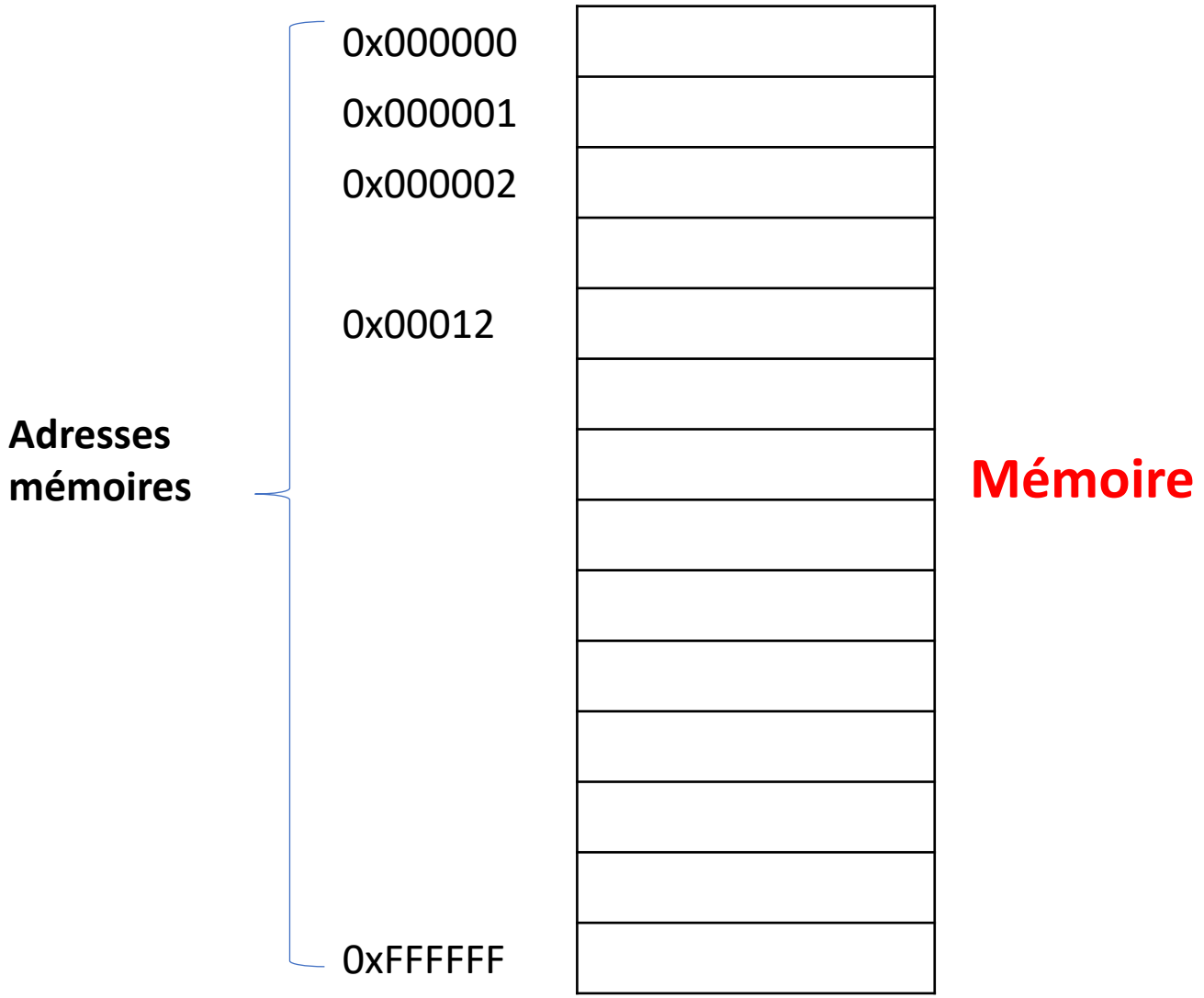
Introduction

Les tableaux : Définition

```
int a = 5;

int T[] = { 5, 6, 0}

int A[] = new int[4];
```



Les tableaux à deux dimensions: Déclaration

Syntaxe :

```
type   nomMatrice[][] = new type[lignes][colonnes];
Type[][]   nomMatrice = new type[lignes][colonnes];
type   nomMatrice[][] = { {liste des valeurs}, {liste valeurs} };
```

Exemples:

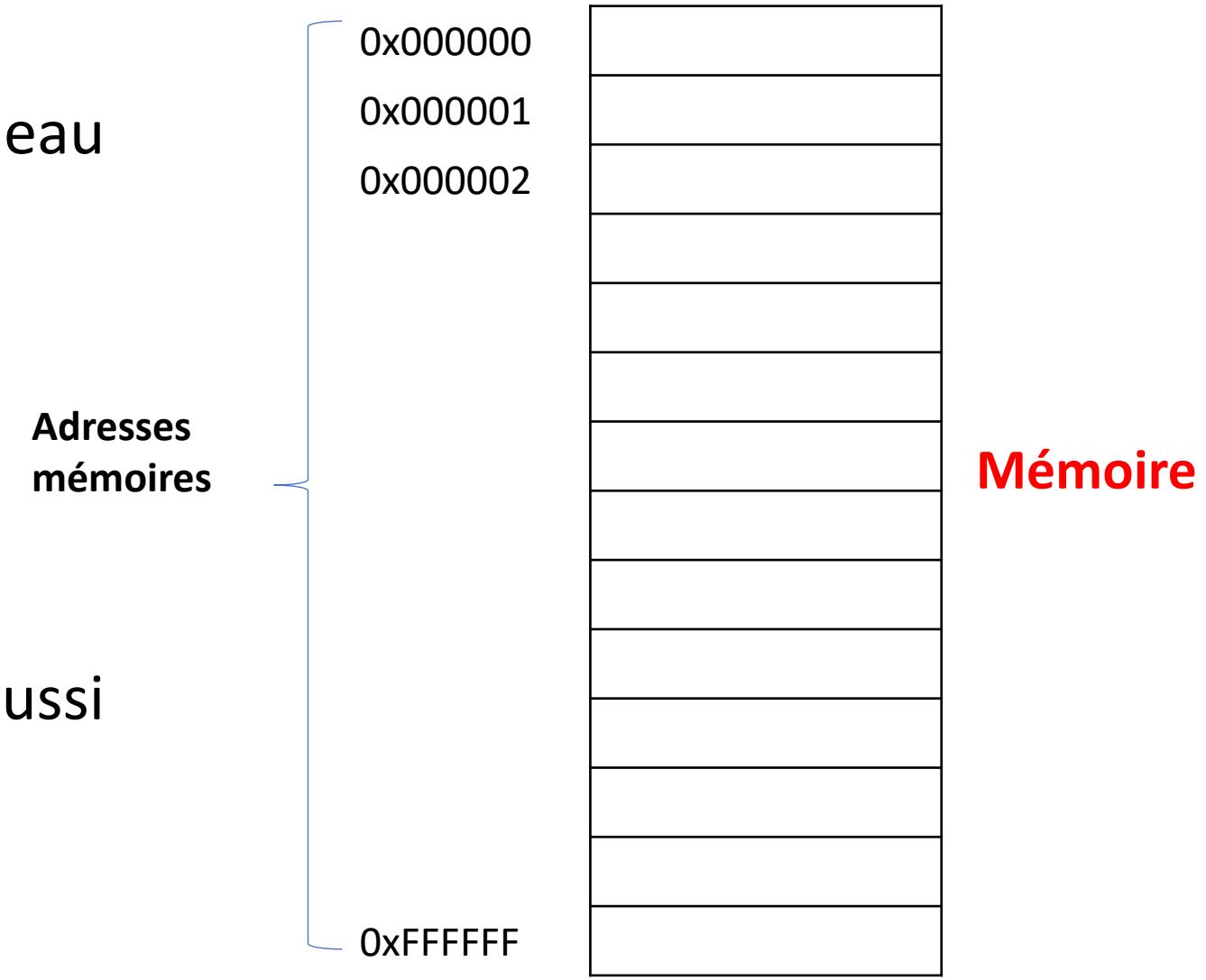
```
int A[][] = new int[5][4];   // une  matrice d'entiers de 5 lignes et 4 colonnes
double B[][] = new double[3][3];  // une  matrice de réels de 3 lignes et colonnes
int C[][]={{1, 0, 0}, {0, 1, 0} , {0, 0, 1}}; // une matrice initialisée dès la déclaration
```

Introduction

Les tableaux : Déclaration

Un tableau d'objets est un tableau d'adresses de ces objets :

String A[] = new String[4];
Personne P[] = new Personne[3];



NB : Les noms des tableaux sont aussi des objets.

Les tableaux : Accéder à un élément

Pour accéder à un élément du tableau :

nomTableau[indice]

nomMatrice[indice_ligne][indice_colonne]

Exemples :

```
int A[] = new int[5];
A[0] = 10;
A[1] = 5;
A[2] = 9;
A[3] = 20;
A[4] = 100;
```

```
int M[][] = new int[2][2];
M[0][0] = 5;
M[0][1] = 15;
M[1][0] = 10;
M[1][1] = 20;
```

Introduction

Les tableaux : Affectation

Exercice : qu'affichera le code suivant ?:

```
int A[] = {3, 1, 5};
int B[] = A;
A[0] = 10;
System.out.println("A[0] = " + A[0]);
System.out.println("B[0] = " + B[0]);
```

Solution :

```
A[0] = 10
B[0] = 10  // int B[] = A; a copié l'adresse et par les valeurs.
```

Les tableaux : Itérer un tableau

Itérer les éléments d'un tableau :

```
int T[] = new int[10];
int l = T.length;  # donne le nombre des éléments d'un tableau

// Itérer par indice
for (int i = 0; i < T.length; i++) {
    System.out.println(T[i]);
}
// Itérer par valeur (foreach)
for (int a : T) {
    System.out.println(a);
}
```

Les tableaux : Itérer une matrice

Itérer les éléments d'une matrice :

```
int M[][] = new int[4][3];

// Itérer une matrice ligne par ligne
for (int i = 0; i < M.length; i++) {
    for (int j = 0; j < M[0].length; j++) {
        System.out.println(M[i][j]);
    }
}
```

Les tableaux : Affectation

Remarque:

En Java, la taille d'un tableau est fixe après sa création. Cela signifie que si vous allouez un tableau de taille 10 et que vous avez besoin de plus d'espace, vous ne pouvez pas simplement ajouter de nouveaux éléments au tableau existant. Vous devrez créer un nouveau tableau de plus grande taille et copier les éléments existants dans ce nouveau tableau, ce qui peut être coûteux en termes de performance.

Introduction

Qu'est ce qu'une collection ?

- Collection
  - Un objet utilisé afin de représenter un groupe d’objets
- Utilisé pour:
  - Stocker, retrouver et manipuler des données
  - Transmettre des données d’une méthode à une autre
- Exemples:
  - Un dossier d’emails (collection de messages)
  - Un répertoire téléphonique (collection de correspondances NOM – N°)

Introduction

Java Collections Framework

- Définit toute la structure des collections en Java
- Constitue une architecture unifiée pour
  - la représentation des collections
  - la manipulation des collections
- Contient des:
  - Interfaces
    - Types de données abstraits représentant des collections
    - Permettent de manipuler les collections indépendamment de leur représentation
    - Organisées en une hiérarchie unique
  - Implémentations
    - Implémentations concrètes des interfaces ➔ Structures de données réutilisables
    - Fournies pour accélérer le développement
  - Algorithmes
    - Méthodes standards fournissant des opérations comme le tri, la recherche, etc.

Introduction

Java Collections Framework

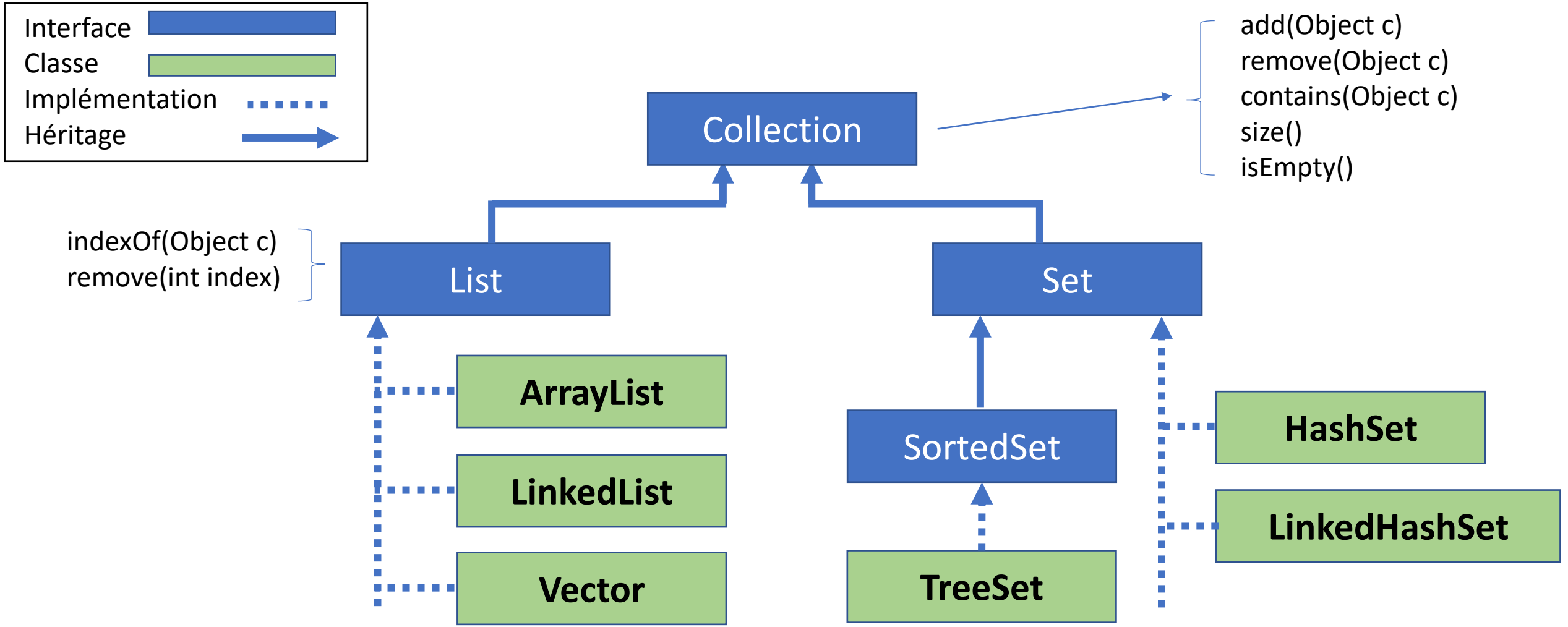
Il existe 3 groupes principales d'interfaces liées aux collections

- Collection
  - Racine de la structure des collections
  - Représente un groupe d'objets (dits « éléments »)
  - Peut autoriser ou non les duplicats
  - Peut contenir un ordre intrinsèque ou pas
- Map
  - Un objet qui établit des correspondances entre des clés et des valeurs
  - Ne peut en aucun cas contenir des duplicats

  ➔ Chaque clé ne peut correspondre qu'à une valeur au plus
- Interfaces de comparaison
  - Permettent le tri des objets du type implémentant l'interface
  - Deux versions: « Comparator » et « Comparable »

Java Collections Framework

Interface: Collection



Java Collections Framework

Interface: Collection

La classe ArrayList :

La classe ArrayList en Java est une des implémentations les plus utilisées de l'interface List de la bibliothèque Java Collections Framework. Elle fournit une structure de données dynamique qui peut contenir des éléments et dont la taille s'ajuste automatiquement lorsque vous ajoutez ou supprimez des éléments.

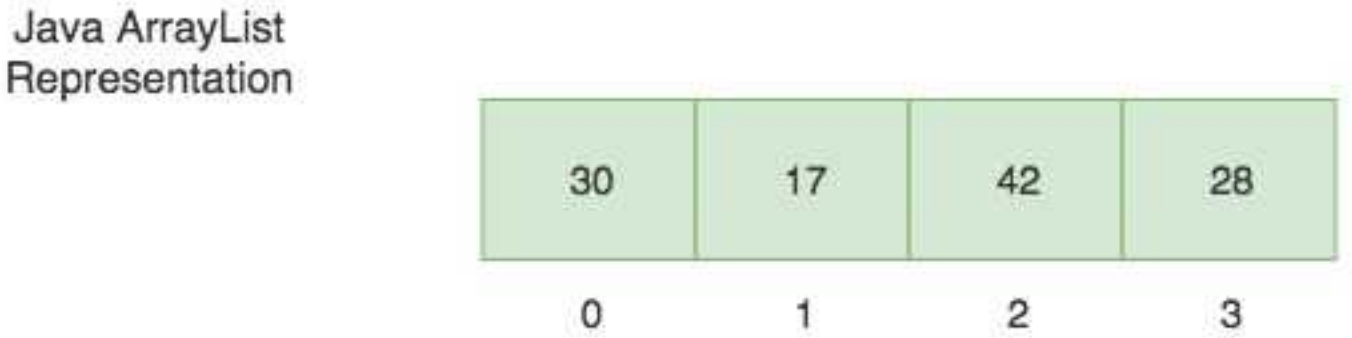
Les méthodes:

add(Object c) ; add(Object c, int index); remove(Object c); remove(int index); get(int index); toArray(); size(); ….

Java Collections Framework

Interface: Collection

La classe ArrayList :



Java Collections Framework

Interface: Collection

La classe LinkedList :

La classe LinkedList en Java fait partie du Java Collections Framework et implémente l'interface List (tout comme ArrayList), mais elle fonctionne différemment en termes de structure interne. Contrairement à ArrayList, qui utilise un tableau dynamique sous-jacent, LinkedList est une liste chaînée, ce qui signifie que chaque élément de la liste contient une référence (ou un lien) vers l'élément suivant et, dans certains cas, vers l'élément précédent..

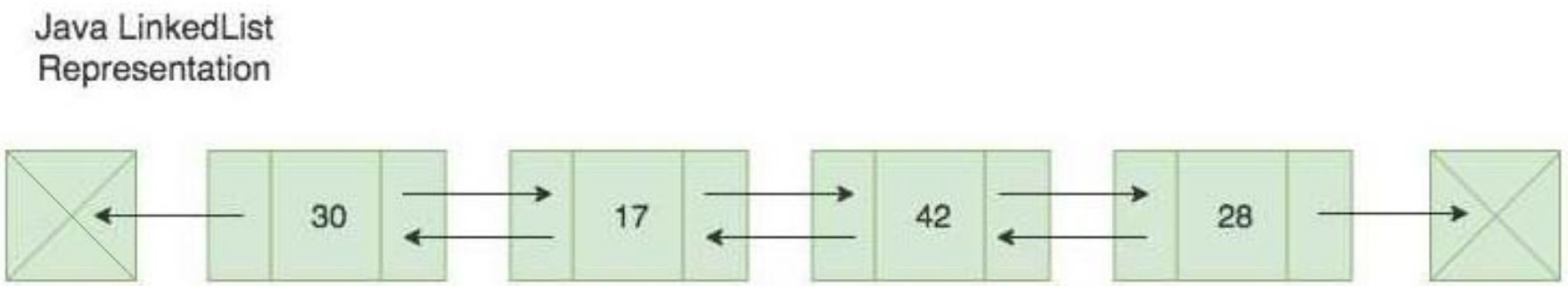
Les méthodes:

add(Object c) ; add(Object c, int index); remove(Object c); remove(int index); get(int index); toArray(); addFirst(Object c); addLast(Object c); ….

Java Collections Framework

Interface: Collection

La classe LinkedList :



Java Collections Framework

Interface: Collection

La classe HashSet:

La classe **HashSet** fait partie de la bibliothèque Java Collections Framework et implémente l'interface Set. Un Set est une collection d'éléments qui ne permet pas les doublons : cela signifie que chaque élément ajouté dans un HashSet doit être unique.

Les méthodes:

add(Object c) ; remove(Object c); toArray();length(); ….

Java Collections Framework

Interface: Collection

La classe TreeSet:

La classe **TreeSet** fait partie du Java Collections Framework et implémente l'interface Set. Elle est basée sur un arbre binaire de recherche (un arbre rouge-noir). Contrairement aux autres implémentations de Set comme HashSet ou LinkedHashSet, qui ne garantissent pas l'ordre des éléments, TreeSet trie automatiquement ses éléments..

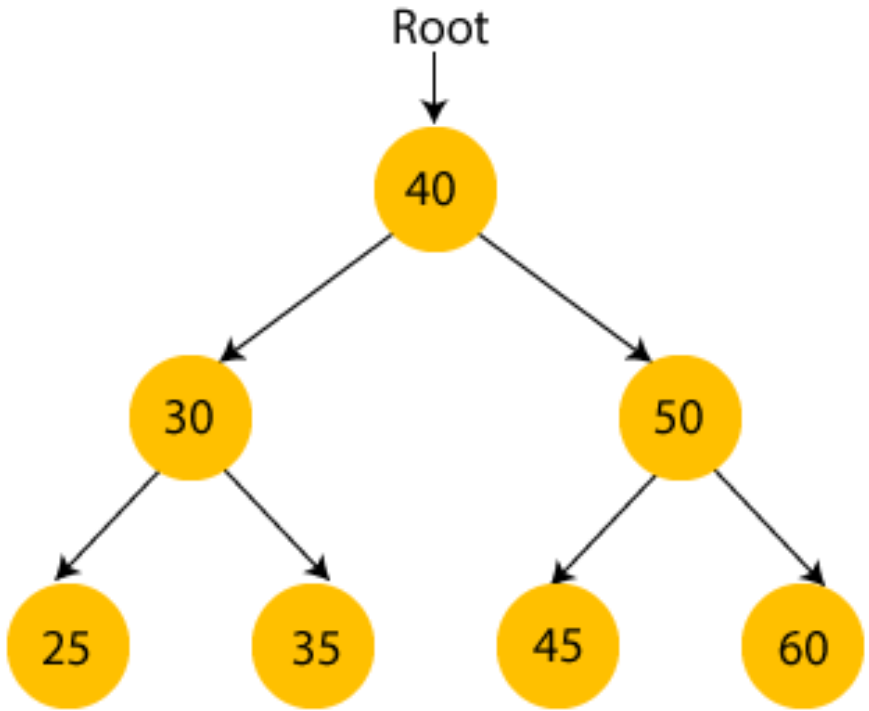
Les méthodes:

add(Object c) ; remove(Object c); toArray();first(); last();….

Java Collections Framework

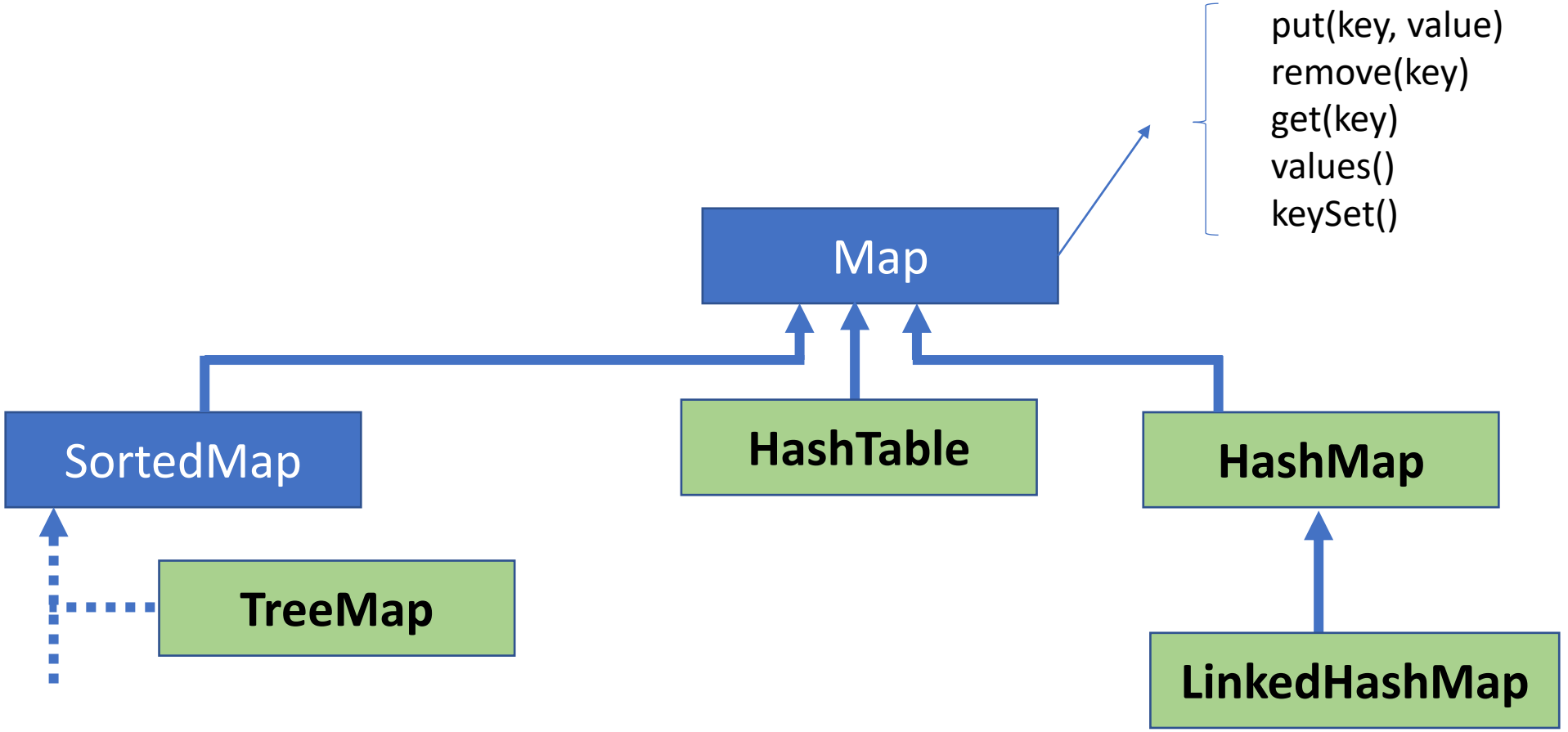
Interface: Collection

La classe TreeSet:



Java Collections Framework

Interface: Map



Java Collections Framework

Interface: Collection

La classe HashMap:

La classe HashMap fait partie de Java Collections Framework et implémente l'interface Map. Elle permet de stocker des paires clé-valeur, où chaque clé est unique et associée à une valeur. Les éléments dans une HashMap sont stockés dans des structures de données appelées tables de hachage. L'idée principale derrière une HashMap est d'offrir un accès rapide aux valeurs via des clés.

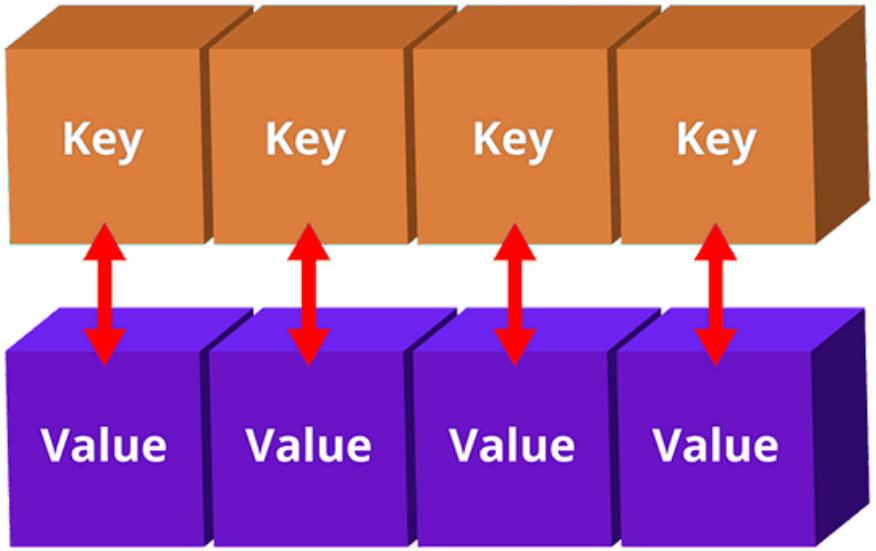
Les méthodes:

put(Object key, Object value) ; remove(Object key); …

Java Collections Framework

Interface: Map

- HashMap :



Java Collections Framework

La comparaison des objets dans les collections

Deux interfaces :

- Comparable
  - Fournit une méthode de comparaison au sein d'une classe
  - Impose de redéfinir une seule méthode `public int compareTo(Object o)` qui renvoie un entier:
    1 si l'objet courant > l'objet « o » fourni dans la méthode
    0 si l'objet courant = l'objet « o » fourni dans la méthode
    -1 si l'objet courant < l'objet « o » fourni dans la méthode

- Comparator
  - Permet de définir une classe servant de comparateur à une autre
  - Impose de redéfinir une seule méthode
    `public int compare(Object o1, Object o2)` qui renvoie un entier:
    1 si *o1* > *o2*
    0 si *o1* = *o2*
    -1 si *o1* < *o2*

Java Collections Framework

La comparaison des objets dans les collections

Exemple d'implémentation de *Comparable*

```
import java.util.*;
public class Personne implements Comparable {
    private String  firstName, lastName;
    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public int compareTo(Object o) {
        Name n = (Name) o;
        int lastCmp = lastName.compareTo(n.lastName);
        if(lastCmp!=0)
            lastCmp = firstName.compareTo(n.firstName);
        return lastCmp;
    }
}
```

Java Collections Framework

La comparaison des objets dans les collections

- Les types primitifs contiennent toujours un ordre naturel

Class	Natural Ordering
`Byte`	signed numerical
`Character`	unsigned numerical
`Long`	signed numerical
`Integer`	signed numerical
`Short`	signed numerical
`Double`	signed numerical
`Float`	signed numerical
`BigInteger`	signed numerical
`BigDecimal`	signed numerical
`File`	system-dependent lexicographic on pathname.
`String`	lexicographic
`Date`	chronological

Java Collections Framework

Algorithmes : Tri

On peut trier un tableau/collection au moyen de méthodes simples:

- Trier un tableau ➔ méthode « sort » de la classe « Arrays »
  - `Arrays.sort(<type>[])`
- Trier une collection ➔ méthodes « sort » de la classe « Collections »
  - Ne fonctionne qu'avec les collections dérivant de « List »
  - Si les éléments de la collection sont comparables (implémentent l'interface « Comparable »)
    ➔ `Collections.sort(List)`
  - Si les éléments de la collection ne sont pas comparables, il faut alors indiquer quelle classe servira de comparateur
    ➔ `Collections.sort(List, Comparator)`

Java Collections Framework

Algorithmes : Autre

- D'autres opérations sont fournies par le Java Collections Framework:
  - Mélange
    - *Collections.shuffle(liste)*
  - Inversion de l'ordre
    - *Collections.reverse(liste)*
  - Réinitialisation des éléments (remplace tous les éléments par l'objet spécifié)
    - *Collections.fill(liste, objetParDefaut)*
  - Copie des éléments d'une liste dans une autre
    - *Collections.copy(listeSource, listeDestination)*
  - Recherche d'extrema
    - Sur base de la position des éléments *min(liste)* et *max(liste)*
    - Sur base d'un comparateur *min(liste, comparateur)* et *max(liste, comparateur)*

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Gestion des exceptions

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Gestion des exceptions

1. Le mécanisme des exceptions en Java (try, catch, finally).
2. Les classes d'exceptions (Exception, RuntimeException).
3. Lever des exceptions (throw) et créer des exceptions personnalisées.
4. Différence entre les exceptions vérifiées et non vérifiées.

Gestion des exceptions

Plan du chapitre :

- Introduction
- Hiérarchie des exceptions
- Traitement des exceptions
  - Interception d'exceptions: bloc *try – catch – finally*
  - Lancement (génération) par une méthode: *throws* et *throw*

Introduction

La gestion des exceptions en Java

Définition :

Une exception est un événement qui perturbe le flux normal d'un programme.

Une exception peut être causée par une erreur de programmation (par exemple, division par zéro) ou une erreur externe (par exemple, un fichier non trouvé).

Gestion des exceptions

Introduction

La gestion des exceptions en Java

Pourquoi gérer les Exceptions ?

Sécurité du programme : Éviter l'arrêt brutal du programme en cas d'erreur.

Améliorer l'expérience utilisateur : Fournir des messages d'erreur utiles.

Maintenance du code : Aider à repérer les erreurs et anomalies de manière efficace.

Gestion des exceptions

Introduction

La gestion des exceptions en Java

Types d'exceptions :

- **Exceptions vérifiées (Checked Exceptions)** : Doivent être explicitement gérées (par exemple, IOException, SQLException).

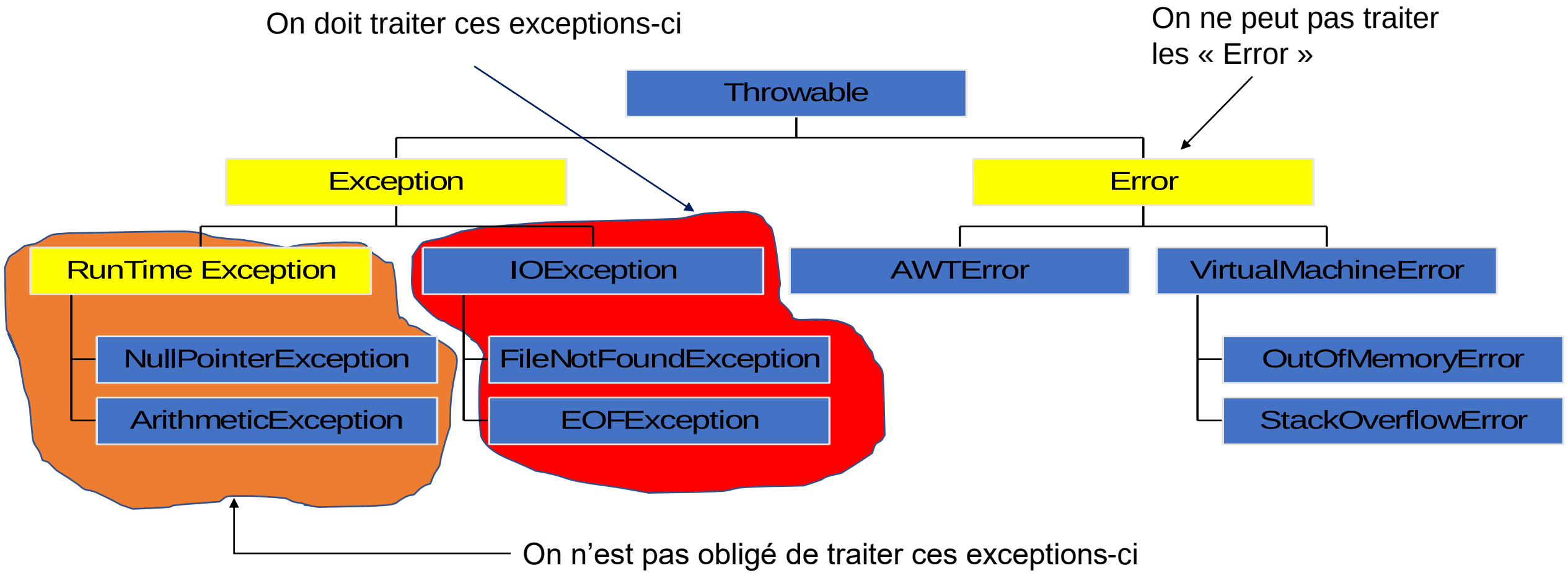
- **Exceptions non vérifiées (Unchecked Exceptions)** : Héritent de RuntimeException, elles ne sont pas obligatoires à gérer (par exemple, NullPointerException, ArithmeticException).

- **Errors :** On ne peut pas les traiter.

Gestion des exceptions

Introduction

Hiérarchie des exceptions



Traitement des exceptions

Principe

- Le traitement des exceptions contient deux aspects:
  - L'interception des exceptions
    - Utilisation du bloc *try – catch – finally* pour récupérer les exceptions
    - Et réaliser les actions nécessaires
  - Le lancement (la génération) d'exceptions
    - Automatiquement par l'environnement run-time ou la machine virtuelle pour certaines exceptions prédéfinies par Java
    - Explicitement par le développeur dans une méthode avec « throws » et « throw » (en tout cas pour les exceptions créées par le développeur

Traitement des exceptions

Interception par bloc try – catch :

```
try
{
  // quelques actions potentiellement risquées
}
catch(SomeException se)
{
  // que faire si une exception de ce type survient
}
```

Traitement des exceptions

Interception de plusieurs types d'exceptions

```
try
{
  // quelques actions potentiellement risquées
}
catch(SomeException se)
{
  // que faire si une exception de ce type survient
}
catch(Exception e)
{
  // que faire si une exception d'un autre type survient
}
```

Traitement des exceptions

Le block finally (1/2)

```
try
{
  // quelques actions potentiellement risquées
}
catch(SomeException se)
{
  // que faire si une exception de ce type survient
}
catch(Exception e)
{
  // que faire si une exception d'un autre type survient
}
finally
{
  // toujours faire ceci, quelle que soit l'exception
}
```

Traitement des exceptions

Exemple :

Implémentation

```
public ExampleException() {
  for(int i=0;i<3;i++) {
    test[i]=Math.log(i);
  }
  try {
    for(int i=0;i<4;i++) {
      System.out.println("log("+i+") = "+test[i]);
    }
  } catch(ArrayIndexOutOfBoundsException ae) {
     System.out.println("Arrivé à la fin du tableau");
  }
  System.out.println("Continuer le constructeur");
}
```

Traitement des exceptions

Lancement avec les mots-clés throws et throw (1/4)

- Si une exception peut survenir, mais que la méthode n'est pas censée la traiter elle-même, il faut en « lancer » une instance
- Il faut préciser que la méthode peut lancer ces exceptions
  ➔ ajouter une clause throws à la signature de la méthode

```
public void writeList() {
  PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
  for (int i = 0; i < vector.size(); i++)
       out.println("Valeur = " + vector.elementAt(i));
}
```

- Peut lancer une IOException ➔ doit être attrapée
- Peut lancer une ArrayIndexOutOfBoundsException

```
public void writeList() throws IOException,
                    ArrayIndexOutOfBoundsException {
```

Traitement des exceptions

Lancement avec les mots-clés throws et throw (2/4)

- Une méthode avec une propriété *throws* peut lancer des exceptions avec *throw*
- Toute exception ou erreur lancée pendant l'exécution provient d'un throw
- Fonctionne avec les exceptions qui héritent de Throwable (la classe de base)
- Le développeur peut créer de nouvelles classes d'exceptions et les lancer avec throw

```
class MyException extends Exception {
  MyException(String msg){
    System.out.println("MyException lancee, msg =" + msg);
  }
}
void someMethod(boolean flag)throws MyException {
        if(!flag) throw new MyException(«someMethod»);
        …
}
```

Traitement des exceptions

Lancement avec les mots-clés throws et throw (3/4)

- Une fois l'exception lancée avec throw, il faut soit l'attraper (avec catch), soit la re-lancer. Si vous la re-lancez, votre méthode doit le déclarer

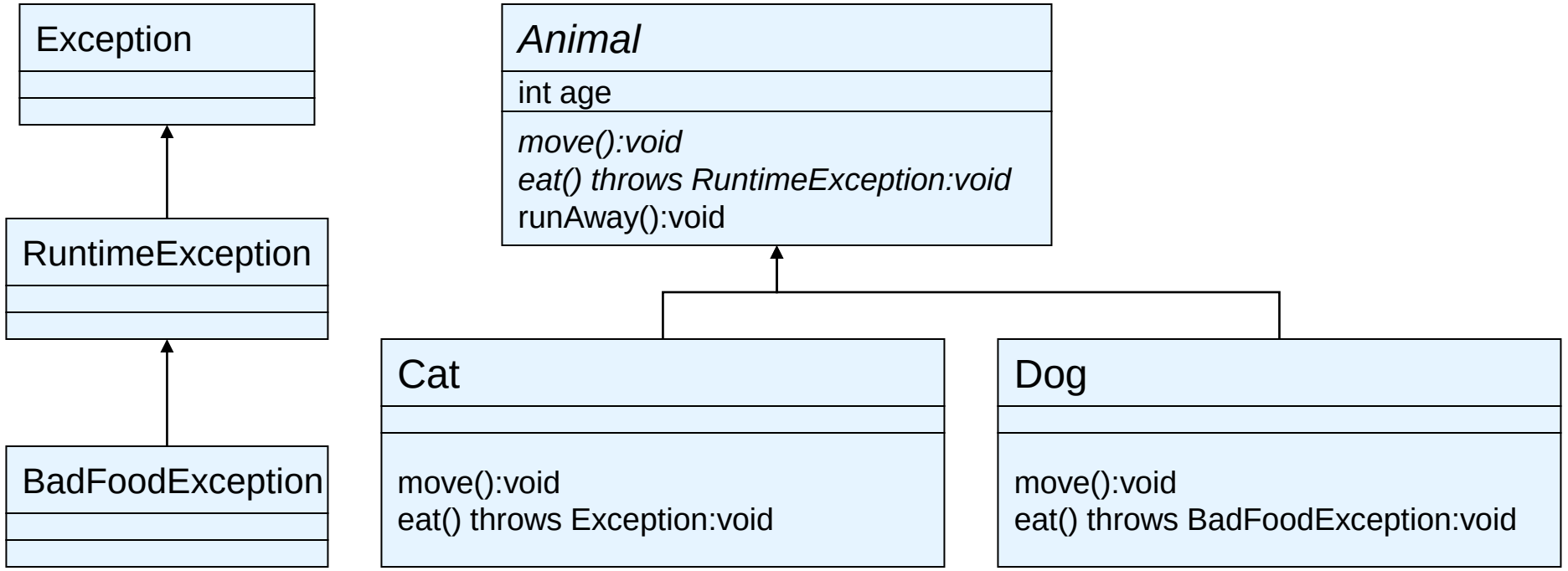
```
public void connectMe(String serverName)  throws ServerTimedOutException {
   boolean success;
   int portToConnect = 80;
   success = open(serverName, portToConnect);
   if (!success) {
     throw new ServerTimedOutException("Connection impossible", 80);
  }
}
```

```
public void connectMe(String serverName)  {
   boolean success;
   int portToConnect = 80;
   success = open(serverName, portToConnect);
   if (!success) {
     try{  throw new ServerTimedOutException("Connection impossible ", 80);
     } catch(ServerTimedOutException stoe){ }
  }
}
```

Traitement des exceptions

Lancement avec les mots-clés throws et throw (4/4)

- Si la méthode à redéfinir lance une exception
  - ➔On ne peut pas lancer une « autre » exception
  - ➔Mais on peut utiliser l'héritage



FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Gestion des fichiers et IO

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Plan Chapitre 5: Gestion des fichiers et IO

1. Lire et écrire dans des fichiers avec les classes File, FileReader, FileWriter.
2. Introduction aux flux (streams) : flux d'entrée/sortie, flux bufferisés.
3. Utilisation des classes Scanner et PrintWriter.

Les entrées / sorties

- Dans la plupart des langages de programmation les notions d'entrées/sorties sont considérées comme une technique de base, car les manipulations de fichiers, notamment, sont très fréquentes.

- En Java, et pour des raisons de sécurité, on distingue deux cas :
  - le cas des applications Java autonomes, où, comme dans n'importe quel autre langage, il est généralement fait un usage important de fichiers,
  - le cas des applets Java qui, ne peuvent pas, en principe, accéder, tant en écriture qu'en lecture, aux fichiers de la machine sur laquelle s'exécute le navigateur (machine cliente).

La gestion des fichiers (1)

- La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.
- Un objet de la classe File peut représenter un fichier ou un répertoire.

La gestion des fichiers (2)

Voici un aperçu de quelques constructeurs et méthodes de la classe File :

Méthodes	Description
File(String pathname)	Ce constructeur crée une nouvelle instance de File en fonction du chemin spécifié sous forme de chaîne de caractères (String). Le chemin peut être absolu ou relatif.
File(URI uri)	Ce constructeur crée une instance de File en fonction d'un objet URI. Il est particulièrement utile lorsque vous travaillez avec des chemins sous forme d'URI.
File(String parent, String child)	Ce constructeur permet de spécifier un chemin parent et un nom de fichier ou de répertoire enfant. Le chemin complet sera le résultat de la concaténation du chemin parent et du nom de l'enfant.
File(File parent, String child)	Ce constructeur est similaire au précédent, mais le chemin parent est représenté par un objet File au lieu d'une chaîne de caractères.

La gestion des fichiers (3)

Méthodes de création et suppression de la classe File :

Méthodes	Description
boolean **createNewFile**()	Crée un nouveau fichier vide s'il n'existe pas déjà.
boolean **mkdir**()	Crée un répertoire unique
boolean **mkdirs**()	Crée les répertoires manquants dans le chemin spécifié.
boolean **delete**()	Supprime le fichier ou le répertoire. Pour les répertoires, il doit être vide.

La gestion des fichiers (4)

Méthodes d'information de la classe File :

Méthodes	Description
boolean **exists**()	Vérifie si le fichier ou le répertoire existe
String **getName**()	Renvoie le nom du fichier ou du répertoire
String **getPath**()	Renvoie le chemin spécifié lors de la création de l'objet File
String **getAbsolutPath**()	Renvoie le chemin absolu
long **length**()	Renvoie la taille du fichier en octets
long **lastModified**()	Renvoie la date de la dernière modification du fichier

La gestion des fichiers (5)

Méthodes d'information et permission de la classe File :

Méthodes	Description
boolean **exists**()	Vérifie si le fichier ou le répertoire existe
String **getName**()	Renvoie le nom du fichier ou du répertoire
String **getPath**()	Renvoie le chemin spécifié lors de la création de l'objet File
String **getAbsolutPath**()	Renvoie le chemin absolu
long **length**()	Renvoie la taille du fichier en octets
long **lastModified**()	Renvoie la date de la dernière modification du fichier
boolean **isFile**()	Vérifie si l'objet est un fichier
boolean **isDirectory**()	Vérifie si l'objet est un répertoire
boolean **canRead**()	Vérifient les permissions de lecture
boolean **canWrite**()	Vérifient les permissions d’écriture

La gestion des fichiers (6)

Méthodes pour lister les fichiers et répértoires de la classe File :

Méthodes	Description
String[] **list**()	Renvoie un tableau de chaînes contenant les noms des fichiers dans le répertoire
File[] **listFiles**()	Renvoie un tableau d'objets File représentant les fichiers et sous-répertoires dans le répertoire

La gestion des fichiers (7)

```
import java.io.*;

public class Listeur
{
    public static void main(String[] args)
    {
        litrep(new File("."));
    }
    public static void litrep(File rep)
    {
        if (rep.isDirectory())
        {   //liste les fichier du répertoire
            String t[]=rep.list();
            for (int i=0;i<t.length;i++)
                System.out.println(t[i]);
        }
    }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire



La gestion des fichiers (8)

```
import java.io.*;
public class Listeur
{
      public static void main(String[] args)
      {   litrep(new File( "c:\\"));  }

      public static void litrep(File rep)
     {
           File r2;
           if (rep.isDirectory())
          {    String t[]=rep.list();
               for (int i=0;i<t.length;i++)
               {
                    r2=new File(rep.getAbsolutePath()+"\\"+t[i]);
                    if (r2.isDirectory())
                              litrep(r2);
                    else
                              System.out.println(r2.getAbsolutePath());
               }
          }
     }
}
```

Si le fichier est un répertoire litrep s'appelle récursivement elle-même

Notion de flux (1)

- Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).
- Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.
- Un flux peut être :
  - Soit une source d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
  - Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.

Notion de flux (2)

Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :

- les fichiers,
- les tableaux de données en mémoire,
- les lignes de communication (connexion réseau)

Notion de flux (3)

- L'intérêt de la notion de flux est qu'elle permet une gestion homogène :
  - quelle que soit la ressource associée au flux de données,
  - quel que soit le flux (entrée ou sortie).
- Certains flux peuvent être associés à des filtres
  - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

Notion de flux (4)

- Les flux sont regroupés dans le paquetage java.io
- Il existe de nombreuses classes représentant les flux
  - il n'est pas toujours aisé de se repérer.
- Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :
  - E / S bufferisées, traduction de données, …
- Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.

Flux d'octets et flux de caractères

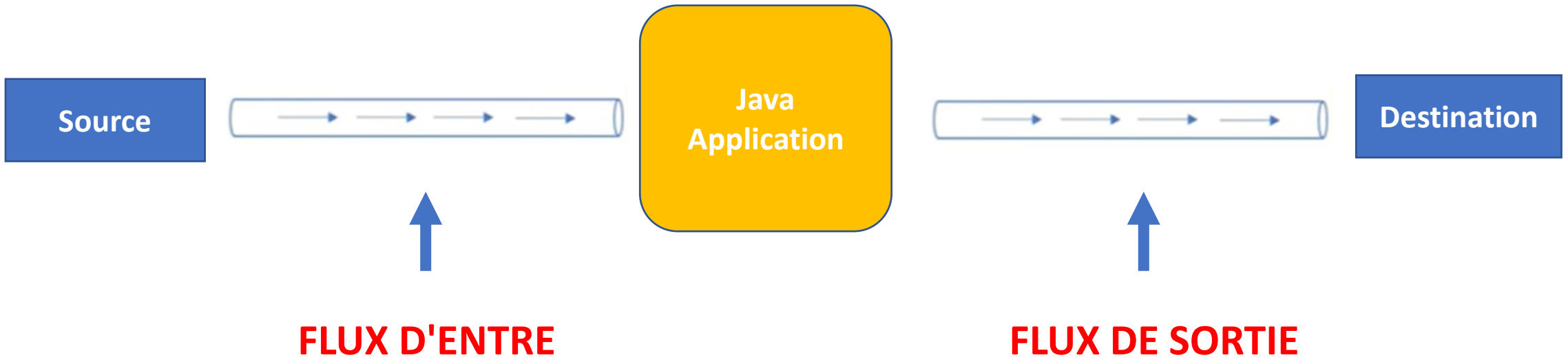
En Java, pour exploiter un flux, les étapes générales consistent à ouvrir, lire (ou écrire), puis fermer le flux. Voici les étapes détaillées :

1. Ouverture d'un flux
2. Lecture/Ecriture depuis ou dans le flux
3. Fermeture du flux

! Prévoir une gestion des exceptions

Gestion des fichiers et IO

Notion de flux :



Flux d'octets et flux de caractères

Les flux sont décomposés en deux grandes familles:

- Les flux de caractères
  - classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.

- Les flux d'octets
  - classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives,

Flux d'octets et flux de caractères

La différence principale entre les classes **Reader** et **InputStream** en Java repose sur le type de données qu'elles gèrent :

- Utilisation :

  - Utilisez **InputStream** pour les données binaires (par exemple, des fichiers audio, des images, des fichiers binaires).

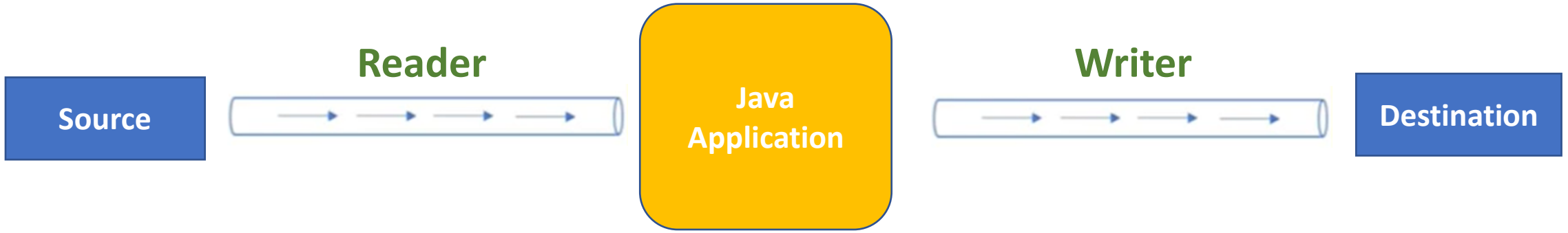
  - Utilisez **Reader** pour des données textuelles (par exemple, des fichiers texte, des flux de caractères).

- Encodage :

  - **InputStream** ne gère pas l'encodage, il lit des octets bruts.

  - **Reader** gère l'encodage des caractères, ce qui est important pour la lecture de texte dans des formats comme UTF-8 ou UTF-16.

Flux de caractères (1)



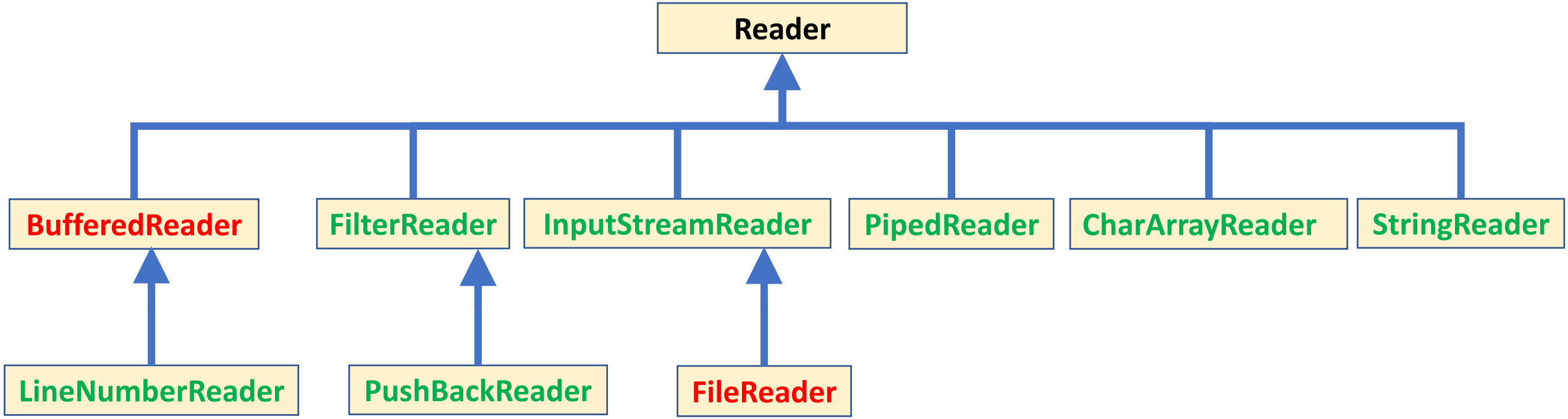
Méthodes de lecture :

int read() : Lit un caractère à la fois et retourne un entier représentant la valeur du caractère, ou -1 si la fin du flux est atteinte.
int read(char[] cbuf) : Lit des caractères dans un tableau et retourne le nombre de caractères lus.
void close() : Ferme le flux et libère les ressources associées.

Méthodes d'écriture:

void write(int c) : Écrit un seul caractère.
void write(char[] cbuf): Écrit un tableau de caractères.
void write(String str): Écrit une chaîne de caractères.
void close() : Ferme le flux d'écriture et libère les ressources.

Flux de caractères - Flux d'entré



Flux de caractères : La classe FileReader

FileReader est une classe de la bibliothèque Java qui permet de lire des fichiers texte. Elle fait partie du package java.io et est utilisée pour lire des caractères à partir d'un fichier.

Flux de caractères : La classe FileReader

```
try {
	// Ouvrir le fichier avec FileReader
	FileReader reader = new FileReader("example.txt");
	int character;
	// Lire chaque caractère du fichier
	while ((character = reader.read()) != -1) {
		// Afficher le caractère (converti en char)
		System.out.print((char) character);
	}
} catch (IOException e) {
	e.printStackTrace();
}
```

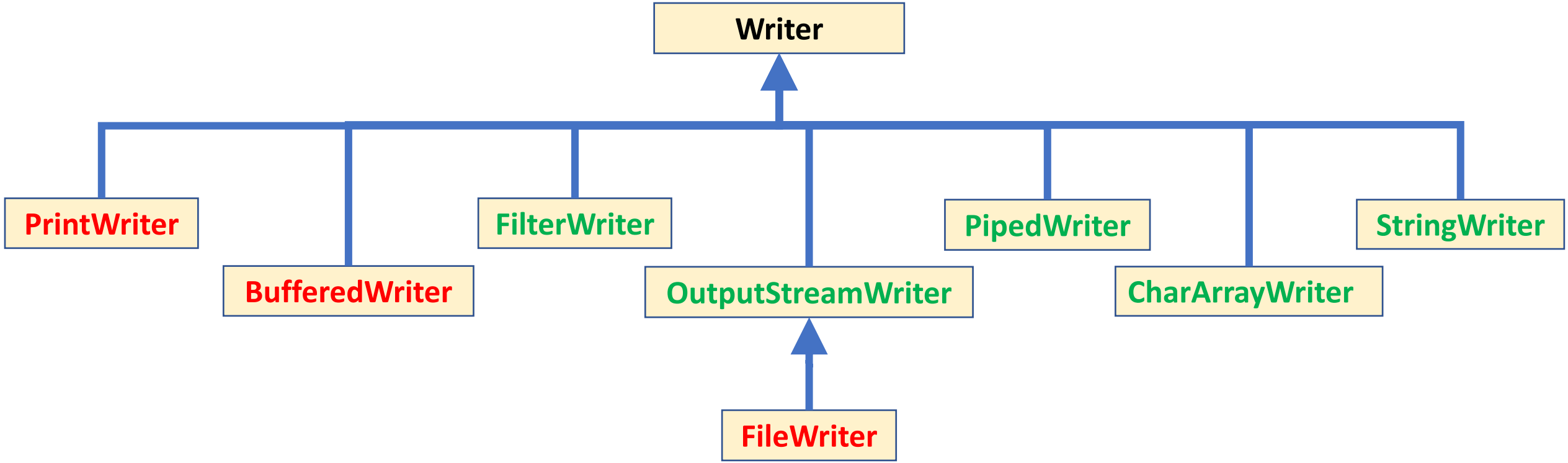
Flux de caractères : La classe BufferedReader

BufferedReader est une classe de la bibliothèque Java qui permet de lire des caractères à partir d'un flux d'entrée de manière plus efficace, en utilisant un tampon (buffer). Elle fait partie du package java.io et est utilisée pour lire des données texte de manière rapide et optimisée, notamment pour les fichiers texte ou les flux de données.

Flux de caractères : La classe BufferedReader

```
try {
        // Ouvrir le fichier
        BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
        String line;
        // Lire chaque ligne du fichier
        while ((line = reader.readLine()) != null) {
                // Traiter la ligne
                String[] values = line.split(",");
                for (String value : values) {
                        System.out.print(value + " ");
                }
                System.out.println();
        }
} catch (IOException e) {
        e.printStackTrace();
}
```

Flux de caractères - Flux de sortie



Flux de caractères : La classe FileWriter

FileWriter est la manière la plus basique pour écrire dans un fichier texte. Si le fichier n'existe pas, il sera créé. Si le fichier existe, son contenu sera écrasé.

java

Flux de caractères (5) : La classe FileWriter

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un objet FileWriter pour écrire dans le fichier
        FileWriter writer = new FileWriter("output.txt");
        // Écrire du texte dans le fichier
        writer.write(text);
        // Fermer le FileWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie dans le fichier.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Flux de caractères : La classe BufferedWriter

BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux de caractères : La classe BufferedWriter

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Flux de caractères : La classe PrintWriter

PrintWriter est une autre classe pratique pour écrire dans un fichier. Elle permet d'écrire des chaînes de caractères et d'autres types de données, tout en offrant une gestion facile des nouvelles lignes.

Flux de caractères : La classe PrintWriter

```
String text = "Ceci est écrit dans le fichier avec PrintWriter.";
try {
        // Créer un PrintWriter pour écrire dans le fichier
        PrintWriter writer = new PrintWriter("outputPrintWriter.txt");

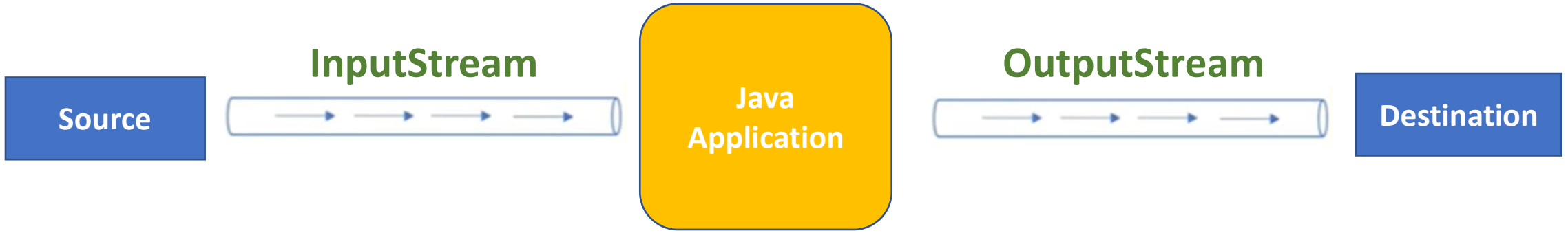
        // Écrire du texte dans le fichier
        writer.println(text);
        writer.println("Une nouvelle ligne de texte.");

        // Fermer le PrintWriter après utilisation
        writer.close();

        System.out.println("Écriture réussie avec PrintWriter.");
} catch (FileNotFoundException e) {
        System.out.println("Erreur lors de la création du fichier : " + e.getMessage());
}
```

Gestion des fichiers et IO

Flux d'octets



Méthodes de lecture :

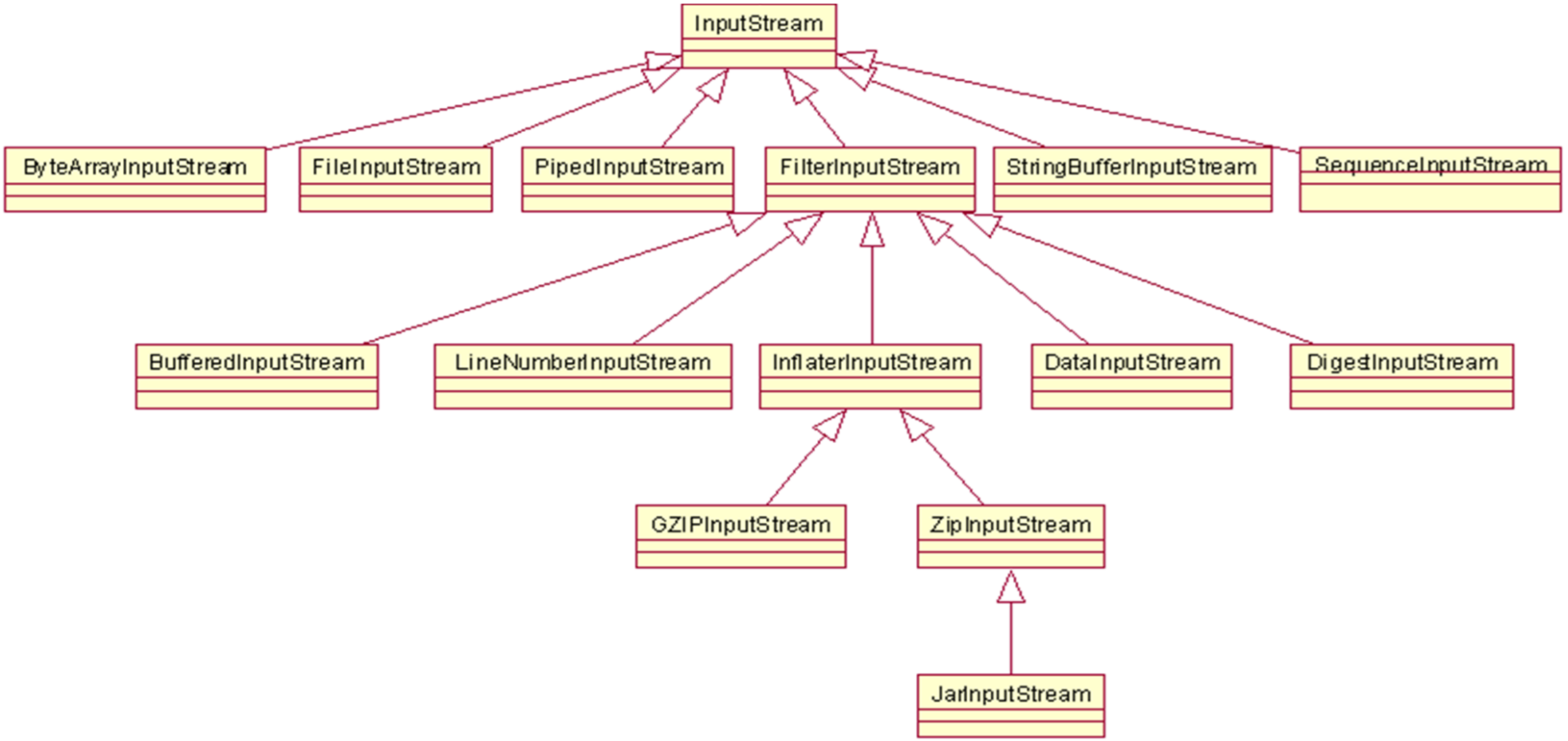
int read() : Lit et renvoie un octet du flux de données, ou -1 si la fin du flux est atteinte.
int read(byte[] b) : Lit des octets dans un tableau et retourne le nombre d'octets lus.
long skip(long n) : Ignore (saute) jusqu'à n octets de données dans le flux d'entrée. Renvoie le nombre d'octets ignorés.
void close() : Ferme le flux et libère les ressources associées.

Méthodes d'écriture:

void write(int b) : Écrit un seul octet.
void write(byte[] b): Écrit un tableau d'octets.
void write(byte[] b, int off, int len): Écrit les octets du tableau b à partir de l'index off jusqu'à len octets dans le flux de sortie.
void close() : Ferme le flux d'écriture et libère les ressources.
flush(): qui permet de purger le tampon en cas d'écritures bufferisées.

Gestion des fichiers et IO

Flux d'octets - Flux d'entré



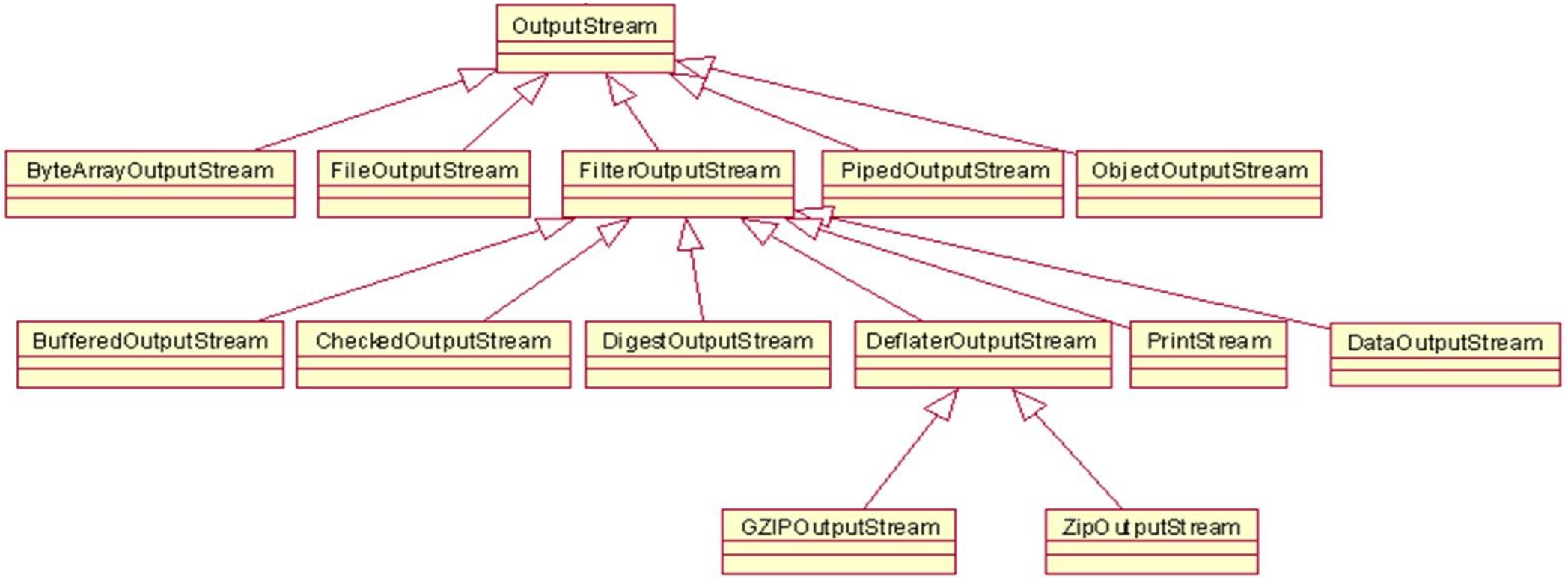
Flux de caractères : La classe FileInputStream

BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux de caractères : La classe FileInputStream

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Flux d'octets - Flux de sortie



Flux d'octets : La classe FileOutputStream

BufferedWriter est une version plus performante de FileWriter car il utilise un tampon (buffer) pour écrire les données. Cela permet d'écrire plus efficacement, en particulier lorsqu'on travaille avec de grandes quantités de données.

Flux d'octets : La classe FileOutputStream

```
String text = "Ligne 1 : Bonjour\nLigne 2 : Ceci est un exemple.";
try {
        // Créer un BufferedWriter pour écrire dans le fichier avec un tampon
        BufferedWriter writer = new BufferedWriter(new FileWriter("outputBuffered.txt"));
        // Écrire du texte dans le fichier
        writer.write(text);
        // Ajouter une nouvelle ligne
        writer.newLine();
        writer.write("Ceci est une nouvelle ligne.");
        // Fermer le BufferedWriter après utilisation
        writer.close();
        System.out.println("Écriture réussie avec BufferedWriter.");
} catch (IOException e) {
        System.out.println("Erreur lors de l'écriture dans le fichier : " + e.getMessage());
}
```

Empilement de flux:

- En Java, chaque type de flux est destiné à réaliser une tâche.
- Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe, il "empile", à la façon des poupées russes, plusieurs flux ayant des comportements plus élémentaires.
- On parle de « flux filtrés ».
- Concrètement, il s'agit de passer, dans le constructeur d'un flux, un autre flux déjà existant pour combiner leurs caractéristiques.

Empilement de flux:

- FileInputStream
  - permet de lire depuis un fichier mais ne sait lire que des octets.
- DataInputStream
  - permet de combiner les octets pour fournir des méthodes de lecture de plus haut niveau (pour lire un double par exemple), mais ne sait pas lire depuis un fichier.
- Une combinaison des deux permet de combiner leurs caractéristiques :

```
FileInputStream fic = new FileInputStream ("fichier");
DataInputStream din = new DataInputStream (fic);
double d = din.readDouble ();
```

Empilement de flux:

Lecture bufferisée de nombres depuis un fichier

```
DataInputStream din = new DataInputStream(new BufferedInputStream(
                                        new FileInputStream ("monfichier")));
```

Lecture de nombre dans un fichier au format zip

```
ZipInputStream zin = new ZipInputStream (
                                    new FileInputStream ("monfichier.zip"));
DataInputStream din = new DataInputStream (zin);
```

La sérialisation

La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).

Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.

La sérialisation peut donc être considérée comme une forme de persistance des données.

La sérialisation

2 classes **ObjectInputStream** et **ObjectOutputStream** proposent, respectivement, les méthodes **readObject** et **writeObject**

Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface **java.io.Serializable**.

Il faut que la classe n’ait pas supprimé le constructeur par défaut

Exemple de sérialisation

```
void sauvegarde(String s) {
try  {FileOutputStream f = new FileOutputStream(new File(s));
            ObjectOutputStream oos = new ObjectOutputStream(f);
            oos.writeObject(this);
        oos.close();}
      catch (Exception e)
       { System.out.println("Erreur "+e);}
}

static Object relecture(String s) {
try  {FileInputStream f = new FileInputStream(new File(s));
            ObjectInputStream oos = new ObjectInputStream(f);
            Object o=oos.readObject();
        oos.close();
            return o;}
      catch (Exception e)
       { System.out.println("Erreur "+e);
        return null;}
}
```

Les flux de données prédéfinis (1)

Il existe 3 flux prédéfinis :

- l'entrée standard **System.in** (instance de InputStream)
- la sortie standard **System.out** (instance de PrintStream)
- la sortie standard d'erreurs **System.err**(instance de PrintStream)

La class Scanner

- Une classe injustement méconnue du JDK est Scanner (depuis la verison 5 de Java)
- fonctionnalités très intéressantes pour parser des chaînes de caractères, et en extraire et convertir les composants.
- Un Scanner peut se brancher sur à peu près n'importe quelle source : InputStream, Readable (et donc Reader), File... et bien sûr une simple String.
- Ensuite utiliser les méthodes de type hasNext...() / next...(), ou alors les méthodes de type find...() / match() / group().

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

La class Scanner

- Méthode hasNext() / next()
- 1. découper la chaîne de caractères en *tokens* grâce à un délimiteur ; il s'agit par défaut d'un caractère "blanc" (espace, tabulation, retour à la ligne...), mais il est évidemment possible de fournir sa propre expression via la méthode useDelimiter(expression).
- 2. utiliser les méthodes de type hasNext...() et next...() pour parcourir, récupérer et convertir ces tokens.
- Les méthodes de type hasNext...() (hasNextInt(), hasNextFloat()...) fonctionnent sur le même principe qu'un Iterator.

La class Scanner

A l'aide de ces méthodes, il est très facile de parser une chaîne dont vous maîtrisez parfaitement le format, par exemple un fichier .csv :

```
String s =
"Dalton;Joe;1.4\n" +
"Dalton;Jack;1.6\n" +
"Dalton;William;1.8\n" +
"Dalton;Averell;2.0";
Scanner scan = new Scanner(s);
scan.useDelimiter(";|\n");
scan.useLocale(Locale.US); // Pour les floats
 while(scan.hasNextLine()) {
  System.out.printf("%2$s %1$s : %3$.1f m %n", scan.next(), scan.next(),
  scan.nextFloat());
}
```

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Multithreading

Licence Professionnelle : Développement Informatique

Pr: Youssef Ouassit

Plan du chapitre

- Introduction
  - Définition
  - Raison d’être
- Création de Thread
  - Par implémentation
  - Par héritage
- Gestion de Thread
  - Méthodes de gestion
  - Diagrammes d’état

Introduction

Qu’est-ce qu’un Thread?

- Un ordinateur qui exécute un programme :
  - Possède un CPU
  - Stocke le programme à exécuter
  - Possède une mémoire manipulée par le programme
  - ➔ Multitasking géré par l’OS

- Un thread (« file ») a ces mêmes capacités
  - A accès au CPU
  - Gère un processus
  - A accès à la mémoire, qu’il partage avec d’autres files
  - ➔ Multithreading géré par la JVM

Introduction

Pourquoi le multithreading?

- Un programme moderne est composé de
  - Une interface graphique
  - Quelques composantes pouvant agir de manière autonome
- Sans multithreading
  - Les composantes ne pourraient agir que lorsque l'interface est suspendue
- Plus généralement, le multithreading
  - Permet de réaliser plusieurs processus indépendants en parallèle
  - Permet de gérer les files d'attente
  - Permet au besoin de synchroniser les différents processus entre eux

Création de Thread

Mise en œuvre (1/3)

- Par implémentation de l'interface
  - Usage
    - ➔ `public void MaClasse implements Runnable`
  - Avantages et inconvénients
    - ☺ Meilleur sur le plan orienté objet
    - ☺ La classe peut hériter d'une autre classe
    - ☺ Consistance
- Par héritage de la classe Thread elle-même
  - Usage
    - ➔ `public void MaClasse extends Thread`
  - Avantages et inconvénients
    - ☺ Code simple (l'objet est un Thread lui-même)
    - ☹ La classe ne peut plus hériter d'une autre classe

Création de Thread

Mise en œuvre (2/3)

```
public class MaFile implements Runnable {
  public void run(){
    byte[] buffer=new byte[512];
    int i=0;
    while(true){
      if(i++%10==0)System.out.println(""+i+" est divisible par 10");
      if (i>101) break;
    }
  }
}

public class LanceFile {
  public static void main(String[]arg){
    Thread t=new Thread(new MaFile());
    t.start();
  }
}
```

Le constructeur de la classe Thread attend un objet Runnable en argument

Création de Thread

Mise en œuvre (3/3)

```
public class MyThread extends Thread {
  public void run(){
    byte[] buffer=new byte[512];
    int i=0;
    while(true){
      if(i++%10==0) System.out.println(""+i+" est divisible par 10");
      if(i>101) break;
    }
  }
}

public class LaunchThread{
  public static void main(String[]arg){
    MyThread t=new MyThread();
    t.start();
  }
}
```

Grâce à l'héritage, un objet de type MyThread est lui-même Runnable, on peut donc appeler un constructeur sans argument

Création de Thread

Méthodes de gestion (1/4)

- `t.start()`
  - Appeler cette méthode place le thread dans l'état "runnable"

    ➔ Eligible par le CPU
- `t.yield() throws InterruptedException`
  - La VM arrête la file active et la place dans un ensemble de files activables. (runnable state)
  - La VM prend une file activable et la place dans l'état actif (running state)
- `t.sleep(int millis) throws InterruptedException`
  - La VM bloque la file pour un temps spécifié (état « d'attente »)
- `t.join() throws InterruptedException`
  - Met la file en attente jusqu'au moment où la file t est terminée (a fini sa méthode run()). Le thread appelant redevient alors activable.

Gestion des Thread

Méthodes de gestion (2/4)

- `Thread.yield() throws InterruptedException`
  - La VM arrête la file active et la place dans un ensemble de files activables. (runnable state)
  - La VM prend une file activable et la place dans l'état actif (running state)
- `Thread.sleep(int millis) throws InterruptedException`
  - La VM bloque la file pour un temps spécifié (état « d' attente »)
- `t.join() throws InterruptedException`
  - Met la file en attente jusqu'au moment où la file t est terminé (a fini sa méthode run()). Le thread appelant redevient alors activable.

Gestion des Thread

Méthodes de gestion (3/4)

```
public class ExJoin {
  public static void main(String[]arg){
    Thread t=new Thread(new FileSecondaire());
    t.start();
    for(int i=0;i<20;i++){
      System.out.println("File principale en cours d'exécution "+i);
      try{
        Thread.sleep(10);
      } catch(InterruptedException ie){}
    }
    try {
      t.join();
    } catch (InterruptedException ie) {}
    System.out.println("t termine son exécution, fin programme");
  }
}
```

Gestion des Thread

Méthodes de gestion (4/4)

```
class FileSecondaire implements Runnable{
  public void run(){
    for(int i=0;i<40;i++){
      System.out.println("File secondaire en execution "+i);
      try{
        Thread.sleep(10);
      } catch (InterruptedException ie) {}
    }
    System.out.println("Fin de file secondaire");
  }
}
```

Gestion des Thread

Diagrammes d'état

