Programmation orientée objet

En JAVA

Pr. Hicham Lakhlef , *hicham.lakhlef@bordeaux-inp.fr*

Contenu du cours :

Chapitre 1 – Introduction à la POO et à Java

- Rappel sur la programmation
- Concepts fondamentaux de la POO (classe, objet, attribut, méthode)
- Pourquoi Java ?
- Exemple simple

Chapitre 2 – Classes et Objets

- Définition et syntaxe d'une classe en Java
- Constructeurs, attributs, méthodes
- Création et utilisation d'objets
- Exemple pratique

Chapitre 3 – Héritage et Polymorphisme

- Héritage et spécialisation
- Redéfinition de méthodes
- Polymorphisme statique/dynamique
- Classes abstraites

Chapitre 4– Gestion avancée

- Gestion des exceptions
- Packages et organisation du code
- Quelques classes utilitaires de la bibliothèque Java

Chapitre 5– Programmation concurrente avec les Threads

- Qu'est-ce qu'un thread ?
- Création de threads en Java (héritage de Thread, implémentation de Runnable)
- Cycle de vie d'un thread
- Synchronisation entre threads
- Exemple d'application multi-threads

Organisation du cours :

- 5 séances de cours en EI
  - Manipulation en fin de séances
- 2 TPs
  - A finir à la maison
  - Tests et corrections disponibles
- Un projet
  - Séances de suivi
  - Binômes
- Note finale : 0.5* QCM + 0,5 * Projet

Chapitre 1

Chapitre 1 – Introduction à la POO et à Java

- Rappel sur la programmation
- Concepts fondamentaux de la POO (classe, objet, attribut, méthode)
- Pourquoi Java ?
- Exemple simple

Rappel sur la programmation

• Un **programme** = suite d'instructions exécutées par un ordinateur.

• La POO (Programmation Orientée Objet) est un paradigme de programmation basé sur la notion d'objets.

• **Paradigmes de programmation :**

- Procédural → instructions séquentielles (ex : C)
- Orienté Objet → manipulation d'**objets** (ex : Java, Python, C++)

• Objectif : rendre le code **modulaire, réutilisable et maintenable**.

Concepts fondamentaux de la POO

- **Classe** : modèle (plan) qui définit les caractéristiques d'un objet.
- **Objet** : instance d'une classe, entité concrète.
- **Attributs** : variables décrivant l'état de l'objet.
- **Méthodes** : fonctions définissant le comportement de l'objet.

Exemple :

•Classe : Voiture
•Attributs : couleur, vitesse
•Méthodes : demarrer(), freiner()
•Objet : maVoiture = new Voiture();

Pourquoi Java ?

- **Langage multiplateforme** : « Write Once, Run Anywhere »
- **Sécurisé** (gestion mémoire, exceptions, sandbox)
- **Orienté Objet** (tout est basé sur des classes/objets)
- **Bibliothèque standard riche** (API Java)
- **Très utilisé** dans l'industrie : applications web, mobiles (Android), serveurs, IoT

Exemple simple en Java


```java
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Bonjour, bienvenue en Java !");

    }
}
```

Quels avantages voyez-vous à organiser un programme en objets plutôt qu'en suite d'instructions séquentielles ?

Chapitre 2

Chapitre 2 – Classes et Objets

- Définition et syntaxe d'une classe en Java
- Constructeurs, attributs, méthodes
- Création et utilisation d'objets
- Exemple pratique

Classes et objets

Class

- **Type défini par l'utilisateur** : Une classe est une structure créée par l'utilisateur pour définir des types de données personnalisés.
- **Données (attributs)** : Les attributs sont des variables qui décrivent les propriétés des objets d'une classe.
- **Comportement (méthodes)** : Les méthodes sont des fonctions définies dans une classe qui déterminent le comportement des objets. Elles manipulent les attributs et définissent les actions possibles.
- **Instances (objets)** : Un objet est une instance d'une classe, ayant ses propres valeurs pour les attributs.

Classes et objets

Déclaration de Class

- **Syntaxe :**

```
<modifier>* class <class_name>{
        <attribute_declaration>*
        <constructor_declaration>*
        <method_declaration>*
}
```

- **Exemple :**

```
public class Compteur {
        private int valeur;
        public void inc(){
                ++valeur;
        }
        public int getValue(){
                return valeur;
        }
}
```

Classes et objets

Déclaration des attributs

- **Syntaxe :**

<modifier>* <type> <attribute_name>[= <initial_value>];

- **Exemple :**

```
public class Test {
        private int x;
        private float f = 0.0f;
        private String nom = "Anonyme";
}
```

Déclaration des méthodes

- **Syntaxe :**

<modifier>* <return_type> <method_name>(<argument>*){ <statement>*}

- **Exemple :**

```
public class Compteur {
        public static final int MAX = 100;
        private int valeur;

        public void inc() {
        if (valeur < MAX) {
                ++valeur;
                }
        }

        public int getValue() {
                return valeur;
        }
}
```

Classes et objets

Accès aux membres d'objet

- **Syntaxe :**

  <object>.<member>

- **Exemple :**

```
public class Counter {
        public static final int MAX = 100;
        private int valeur;

        public void inc() {
        if (valeur < MAX) {
                ++ valeur;
                }
        }

        public int getValue() {
                return valeur;
        }
}
```

```
Counter c = new Counter();
c.inc();
int i = c.getValue();
```

Masquage d'informations

- **Le problème:**

Le code client a un accès direct aux données internes

```
/* C language */

struct Date {
        int year, month, day;
};
```

```
/* C language */

Date d;
d.day = 32; //jour incorrect
d.month = 2; d.day = 30;
// 30 fevrier ?
d.day = d.day + 1;
// pas de verification
```

Masquage d'informations

- **La solution:**

Le code client doit utiliser des setters et des getters pour accéder aux données internes

```
// Java language
public class Date {
        private int year, month, day;
        public void setDay(int d){..}
        public void setMonth(int m){..}
        public void setYear(int y){..}
        public int getDay(){...}
        public int getMonth(){...}
        public int getYear(){...}
}
```

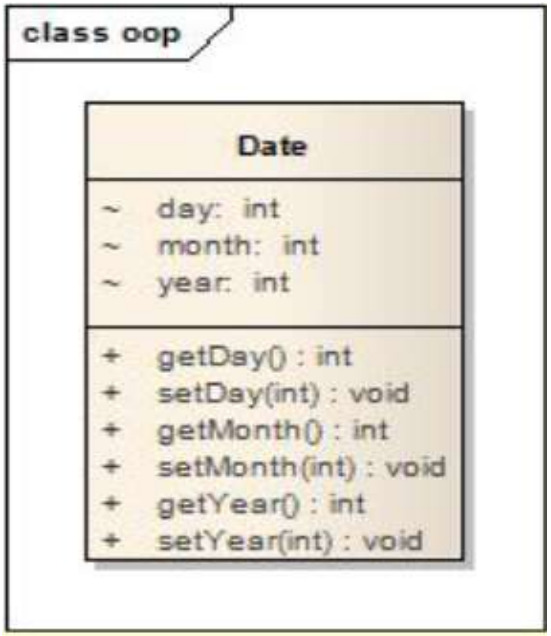
```
Date d = new Date();
// no assignment
d.setDay(32);
// month is set
d.setMonth(2);
// no assignment
d.day = 30;
```

Classes et objets

Encapsulation

- **Regroupement** de **données** avec les **méthodes** qui opèrent sur ces données

(restreinte de l'accès direct à certains composants d'un objet)

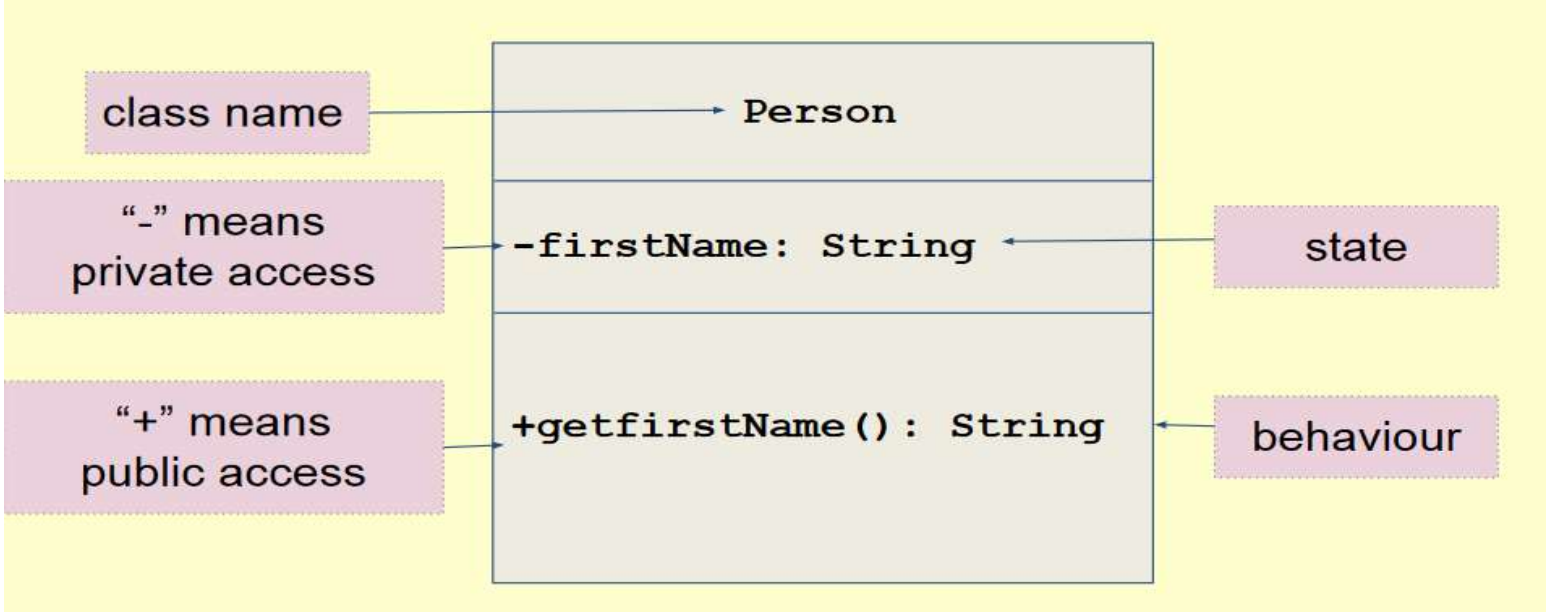


- Masque l'implémentation détails d'une class
- Forcer l'utilisateur à utiliser un interface pour accéder aux données
- Rend le code plus maintenable

Classes et objets

UML (Unified Modeling Language) - Représentation graphique des classes

- Les diagrammes de classes UML aident à visualiser la structure d'une classe, en montrant ses attributs et méthodes, ainsi que leurs niveaux d'accès.



Classes et objets

Les types de diagrammes UML

1.Diagrammes structurels (statique) :

1. Représentent la structure du système
2. Exemples :
    1. **Diagramme de classes**
    2. Diagramme d'objets
    3. Diagramme de composants

2.Diagrammes comportementaux (dynamique) :

1. Montrent le comportement du système au fil du temps
2. Exemples :
    1. Diagramme de séquence
    2. Diagramme d'activités
    3. Diagramme d'états-transitions

Déclaration des constructeurs

- **Syntaxe :**

[<modifier>]<class_name>(<argument>*){ <statement>*}

```
public class Date {
        private int year, month, day;
        public Date(int y, int m, int d) {
                if (verify(y, m, d)) {
                        year = y;
                        month = m;
                        day = d;
                }
        }
        private boolean verify(int y, int m, int d) {
                // ...
        }
}
```

Les constructeurs

- Rôle : initialisation de l'objet
- Le nom du constructeur doit être le même que celui du nom de la classe.
- Ne doit pas avoir de type de retour.
- Chaque classe doit avoir **au moins un constructeur**.
  - Si vous n'écrivez pas de constructeur, le compilateur générera un constructeur par défaut.
- Les constructeurs sont généralement déclarés publics.
  - Le constructeur peut être déclaré privé → Vous ne pouvez pas l'utiliser à l'extérieur la classe.
- Une classe peut avoir plusieurs constructeurs : *Surcharge du constructeur.*

Constructeur par défaut

Il y a toujours au moins un constructeur dans chaque classe.

- Si le programmeur ne fournit aucun constructeur, le constructeur par défaut est généré par le compilateur
  - Le constructeur par défaut ne prend aucun argument
  - Le corps du constructeur par défaut est toujours vide

```
public class Date {
        private int year, month, day;
        public Date( ){
        }
}
```

Classes et objets

Les objets

- Les objets sont des instances de classes
- Sont alloués sur le tas à l'aide du nouvel opérateur
- Le tas est une zone de mémoire dynamique où Java alloue tous les objets créés avec *new*.
- Le constructeur est invoqué automatiquement sur le nouveau objet

```
Counter c = new Counter();
Date d1 = new Date( 2050, 9, 23);
Person p = new Person("John","Smith");
```

Packages

- Aider à gérer de grands systèmes logiciels
- Contenir : classes, sous-packages

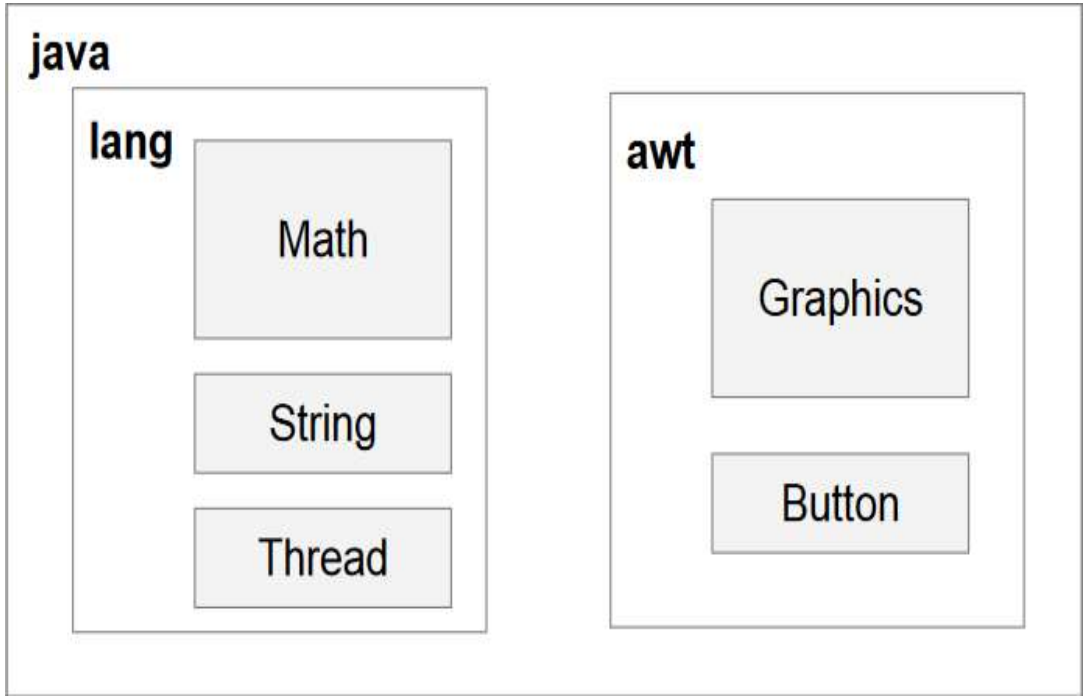
● **Syntaxe :**

package <top_pkg_name>[.<sub_pkg_name>]* ;

● **Exemples :**

```
package java.lang;
        public class String{
            //...
        }
```

- *Déclaration au début du fichier source*
- *Une seule déclaration de package par fichier source*
- *Si aucun nom de package n'est déclaré → la classe est placée dans le package par défaut*



Classes et objets

Construction et initialisation d'objets

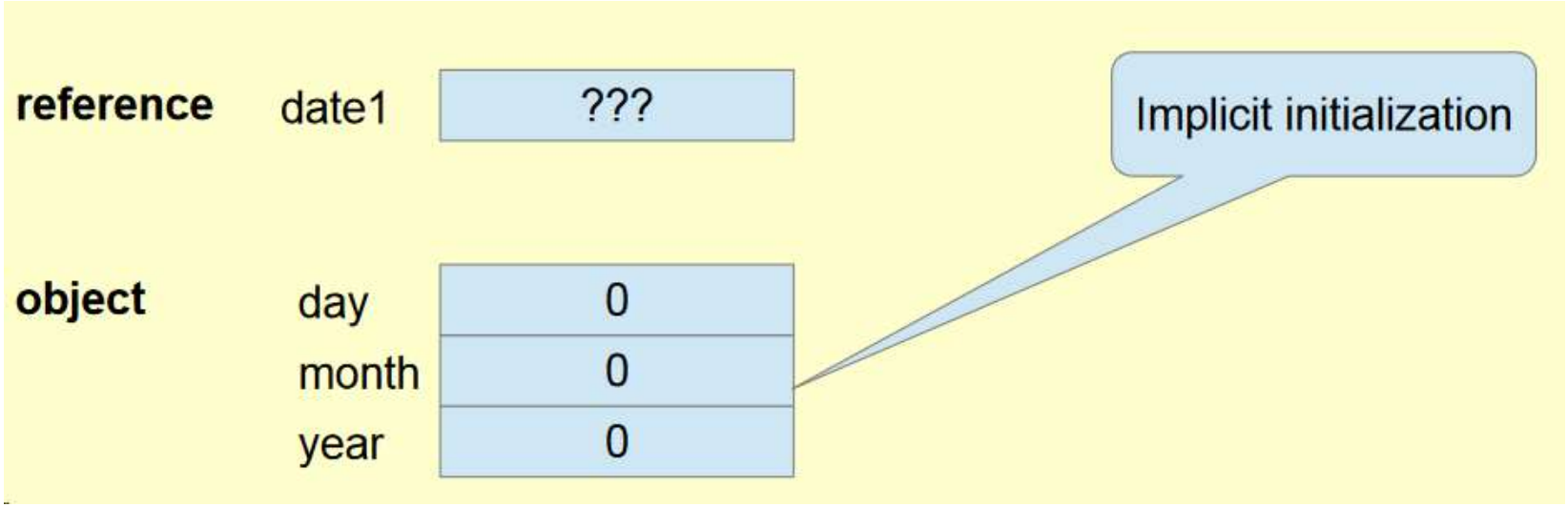
```
MyDate date1 = new MyDate(20, 6, 2016);
```

1) La mémoire est allouée à l'objet
2) L'initialisation explicite des attributs est effectuée
3) Un constructeur est exécuté
4) La référence de l'objet est renvoyée par le nouvel opérateur
5) La référence est affectée à une variable (dans cette exemple **date1**)

Construction et initialisation d'objets

MyDate date1 = new MyDate(20, 6, 2016);

1) La mémoire est allouée à l'objet



Classes et objets

Construction et initialisation d'objets

MyDate date1 = new MyDate(20, 6, 2016);

2) L'initialisation explicite des attributs est effectuée


```
public class MyDate {
      private int day = 26;
      private int month = 9;
      private int year = 2016;
}
```

Classes et objets

Construction et initialisation d'objets

MyDate date1 = new MyDate(20, 6, 2016);

3) Un constructeur est exécuté

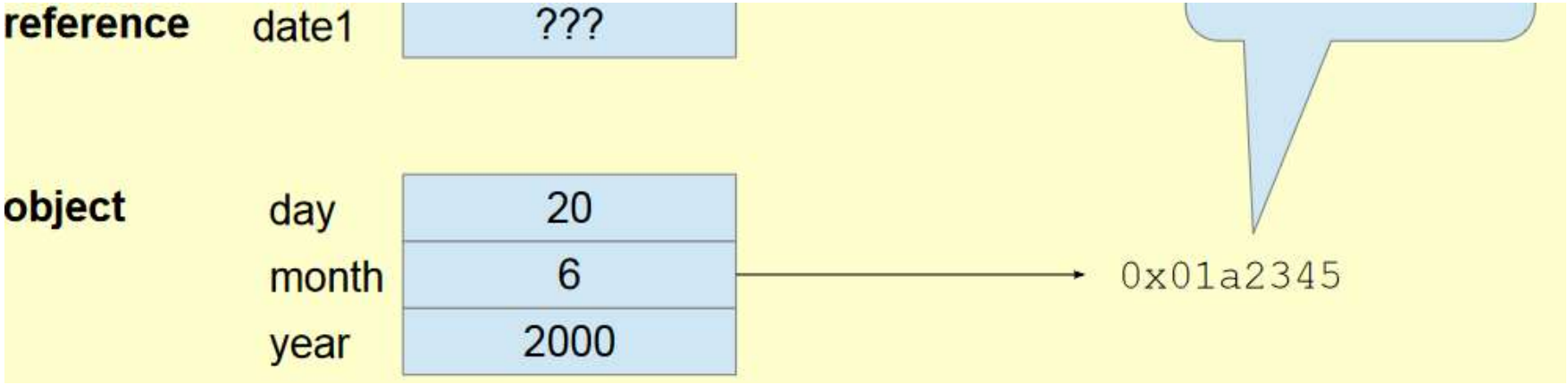

```
public class MyDate {
      private int day = 26;
      private int month = 9;
      private int year = 2016;
}
```

Classes et objets

Construction et initialisation d'objets

MyDate date1 = new MyDate(20, 6, 2016);

4) La référence de l'objet est renvoyée par le nouvel opérateur



Construction et initialisation d'objets

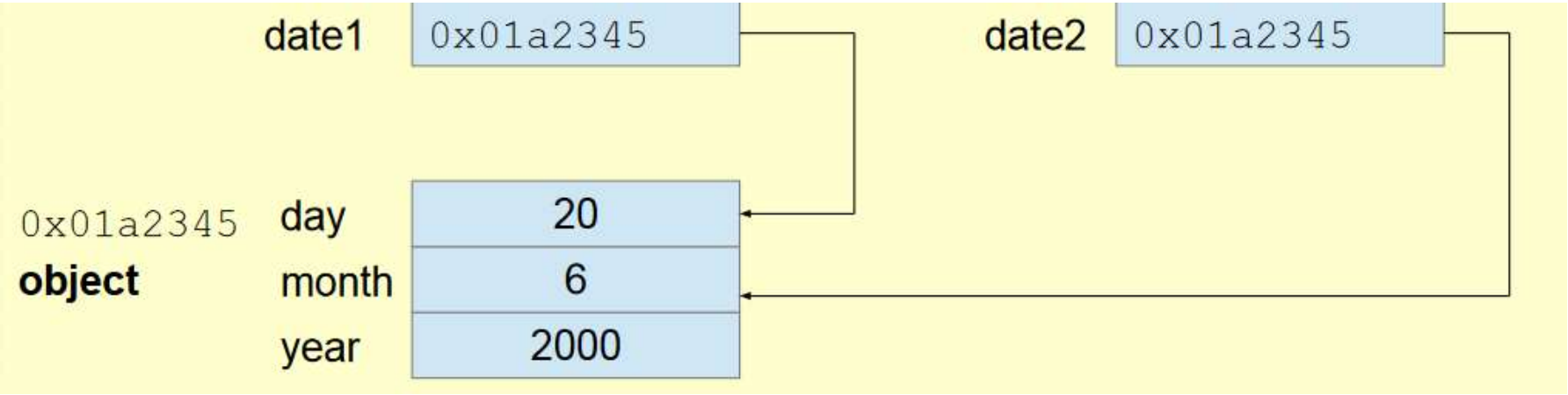
MyDate date1 = new MyDate(20, 6, 2000);

5) La référence est affectée à une variable

Attribution de références

```
MyDate date1 = new MyDate(20, 6, 2000);
MyDate date2 = date1;
```

- Deux variables font référence à un seul objet
- L'initialisation explicite des attributs est effectuée



Passage de paramètres

Passage par Valeur

```
public class PassTest {
        public void changePrimitive(int value) {
                ++value;
        }

        public void changeReference(MyDate from, MyDate to) {
                from = to;
        }

        public void changeObjectDay(MyDate date, int day) {
                date.setDay(day);
        }
}
```

Passage de paramètres

Passage par Valeur

```
PassTest pt = new PassTest();
int x = 100;
pt.changePrimitive( x );
System.out.println( x );
MyDate oneDate = new MyDate(3, 10, 2050);
MyDate anotherDate = new MyDate(3, 10, 2020);
pt.changeReference( oneDate, anotherDate );
System.out.println( oneDate.getYear() );
pt.changeObjectDay( oneDate, 12 );
System.out.println( oneDate.getDay() );
```

Résultat :

100

2050

12

Classes et objets

La référence this

```
public class MyDate{
        private int day = 26;
        private int month = 9;
        private int year = 2050;
        public MyDate( int day, int month, int year){
                this.day = day; this.month = month; this.year = year;
        }
        public MyDate( MyDate date){
                this.day = date.day; this.month = date.month; this.year = date.year;
        }
        public MyDate creteNextDate(int moreDays){
                MyDate newDate = new MyDate(this);
                //... add moreDays
                return newDate;
        }
}
```

This pour résoudre l'ambiguïté entre les variables d'instance et les paramètres, et pour passer l'objet actuel comme paramètre à une autre méthode.

Classes et objets


```
class Point {
   int x, y;

   Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Questions :
1.Que fait this.x = x; et this.y = y; ?
2.Que se passerait-il si on écrivait simplement x = x; y = y; ?

```
class Demo {
   void print(Demo d) { System.out.println(d); }
   void call() { print(this); }
}
```

Questions :
1.Que représente this dans call() ?
2.Que sera imprimé ?

Classes et objets

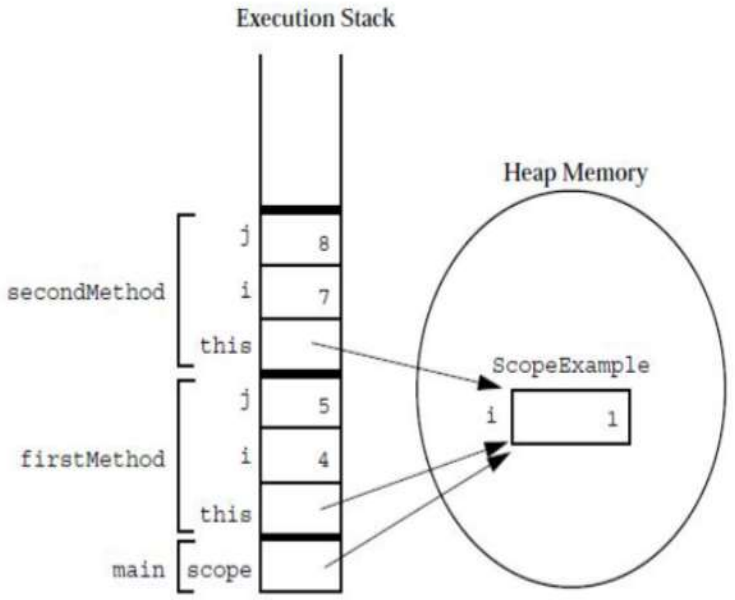
Variables et portée

Les variables locales sont :

- Défini dans une méthode
- Créé lorsque la méthode est exécutée et détruite quand la méthode est quittée
- Non initialisé automatiquemen
- Créé sur la pile d'exécution

```
public class ScopeExample {
  private int i=1;

  public void firstMethod() {
    int i=4, j=5;

    this.i = i + j;
    secondMethod(7);
  }
  public void secondMethod(int i) {
    int j=8;
    this.i = i + j;
  }
}
```

```
public class TestScoping {
  public static void main(String[] args) {
    ScopeExample scope = new ScopeExample();

    scope.firstMethod();
  }
}
```



Les tableaux

- Qu'est-ce qu'un tableau ? Groupe d'objets de données du même type
- Tableaux de types **primitifs** :

```
int t[];
int [] t;
```

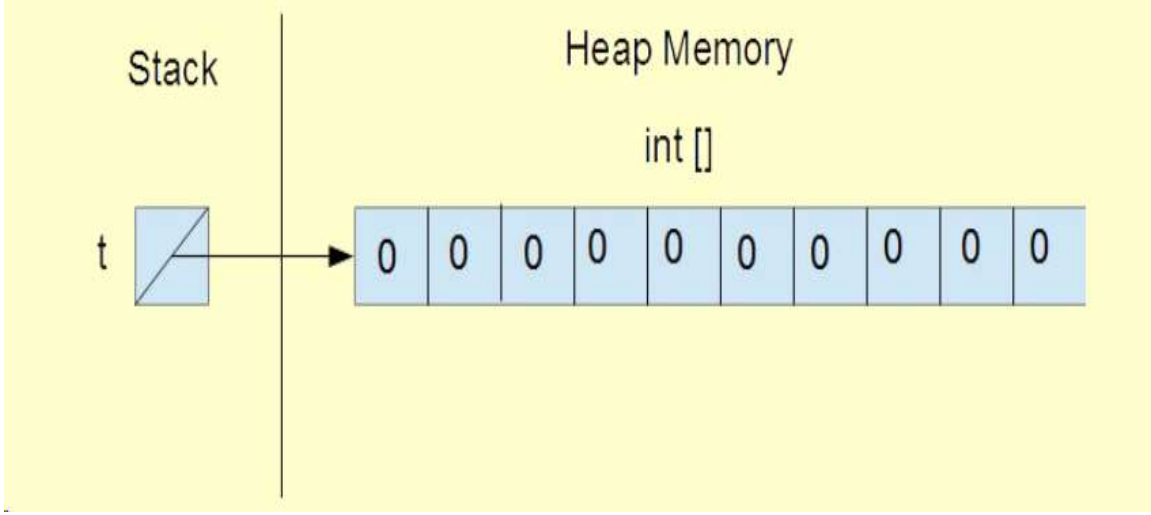
- Tableaux de types de **référence** :

```
Point p[];
Point[] p;
```

Création de tableaux

- Les tableaux sont des objets → sont créés avec **new**

```
//tableau declaration
int [] t;
//tableau creation
t = new int[10];
//print the array – enhanced for loop
for( int v: t ){
        System.out.println( v );
}
void printElements( int t[] ){
        for( int i=0; i < t.length; ++i){
                System.out.println( t[i] );
        }
}
```



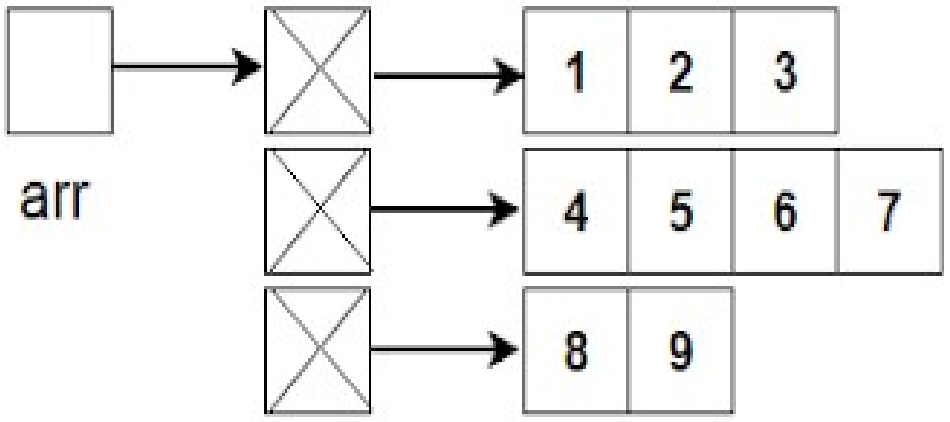
Tableaux multidimensionnels

- Les tableaux sont des objets → sont créés avec **new**
- Tableaux **rectangulaires** :

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

- Tableaux **non rectangulaires** :

```
int [][] tableau ;
tableau = new int[2][];
tableau[0] = new int[2];
tableau[1] = new int[3];
tableau[2] = new int[1];
```



Membres statiques

Accéder aux membres statiques

- **Recommandé** : Utiliser *nom_classe.nom_membre* pour accéder aux membres statiques de la classe. Cela rend le code plus clair et montre explicitement que le membre appartient à la classe plutôt qu'à une instance spécifique.

```
public class Exemple {
        public static int staticValue = 10;
        public static void main(String[] args) {
            System.out.println(Exemple.staticValue); // Recommandé
        }
}
```

Membres statiques

Accéder aux membres statiques

- **Non recommandé (mais fonctionnel)** : Utiliser *référence_instance.nom_membre* pour accéder aux membres de l'instance. Bien que cela fonctionne, cela peut réduire la clarté du code, en particulier lorsqu'il y a des conflits de noms entre les variables locales et les variables d'instance.

```
public class Exemple {
        public int instanceValue = 5;
        public void displayValue() {
                Exemple anotherExemple = new Exemple();
                anotherExemple.instanceValue = 10; // Fonctionnel mais moins clair
                System.out.println(this.instanceValue); // Utiliser 'this' pour plus de clarté
        }
}
```

Membres statiques

Membres statiques

- Données statiques + méthodes statiques = membres statiques
- Les données sont allouées au moment du chargement de la classe → peuvent être utilisé sans instances
- Les méthodes d'instance peuvent utiliser des données statique.
- Les méthodes statiques ne peuvent pas utiliser de données d'instance.

Membres statiques

Membres statiques

```
public class InstanceCounter {
        private static int counter = 0;

        public InstanceCounter() {
                ++counter;
        }

        public static int getCounter() {
                return counter;
        }
}
System.out.println(InstanceCounter.getCounter());
InstanceCounter ic = new InstanceCounter();
System.out.println(InstanceCounter.getCounter());
```

Qu'affiche ce programme ?

Membres statiques

Singleton design pattern

- Le modèle Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

```
public class Singleton {
	private static Singleton instance;
	private Singleton(){
	}
	public static Singleton getInstance(){
		if( instance == null ){
			instance = new Singleton();
		}
	return instance;
	}
}
```

Membres statiques

Le mot-clé final

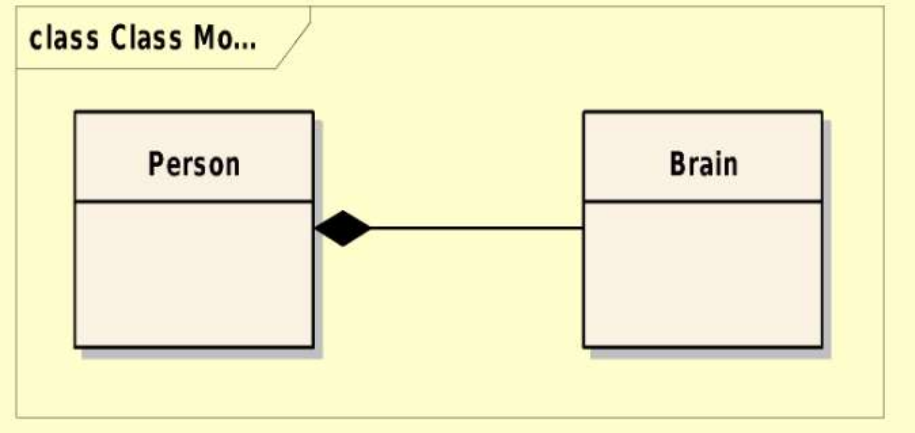
- **Class**
  - Vous ne pouvez pas sous-classer (héritage) une classe finale.
- **Méthode**
  - Vous ne pouvez pas remplacer (redéfinir) une méthode finale.
- **Variable**
  - Une variable finale est une constante.
  - Vous ne pouvez définir une variable finale qu'une seule fois.
  - L'affectation se fait indépendamment de la déclaration (variable finale vide).

Variables Finales Vides : Ce sont des variables finales qui ne sont pas initialisées lors de leur déclaration. Elles doivent être initialisées avant leur utilisation. Une fois initialisées, leur valeur ne peut plus être modifiée.

Relations entre les classes

Composition

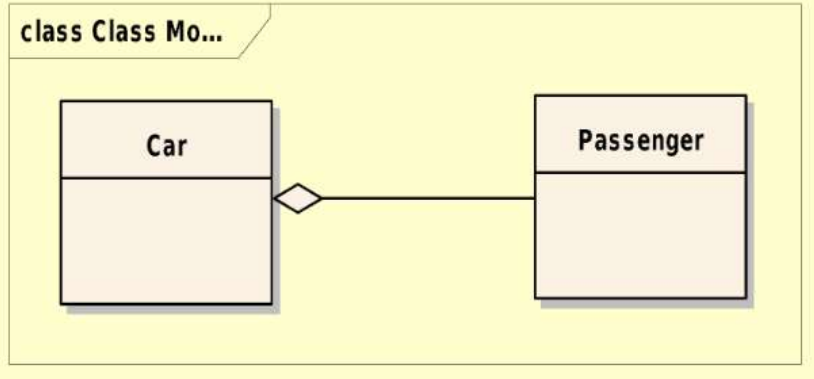
- La **composition** permet de construire des objets complexes à partir de composants plus simples.
- **Association forte** : Dans une relation de composition, l'objet contenu ne peut exister sans l'objet conteneur. L'objet contenu fait partie intégrante de l'objet conteneur.
- **Propriété totale** : L'objet conteneur a la responsabilité complète de la durée de vie de l'objet contenu. Si l'objet conteneur est détruit, l'objet contenu l'est également.



Relations entre les classes

Agrégation

- L'**agrégation** décrit une relation où un objet ("conteneur") est lié à un ou plusieurs objets ("contenus"), mais ces objets contenus peuvent exister indépendamment du conteneur.
- **Association faible** : Dans l'agrégation, l'objet contenu peut exister indépendamment de l'objet conteneur. L'objet contenu est lié à l'objet conteneur, mais il n'en fait pas partie intégrante.
- **Propriété partielle** : L'objet conteneur ne possède pas la responsabilité complète de la durée de vie de l'objet contenu. Si l'objet conteneur est détruit, l'objet contenu peut continuer à exister indépendamment.

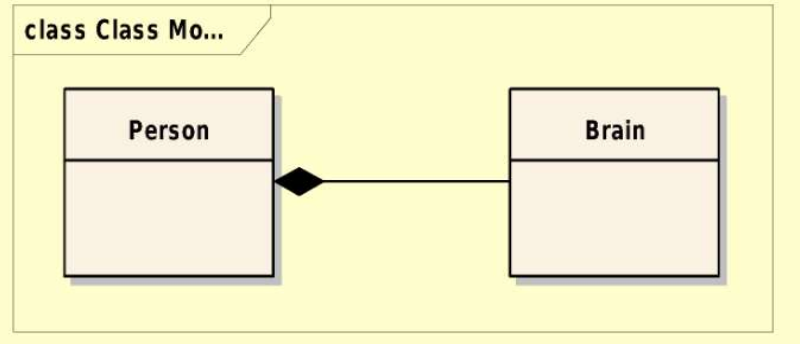


Relations entre les classes

Implémentation des associations

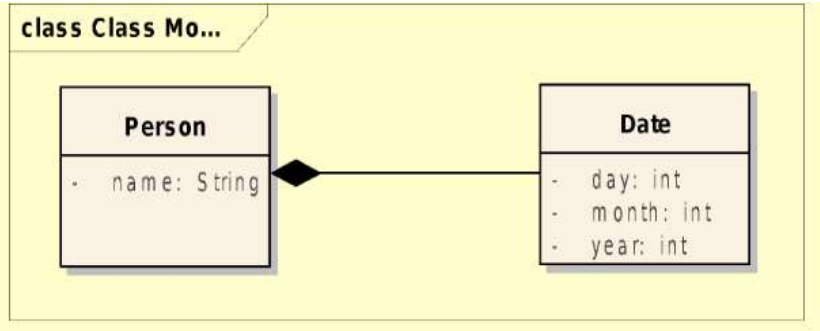
```
public class Brain{
//...
}
```

```
public class Person{
private Brain brain;
//...
}
```




```
public class Date{
private int day;
private int month;
private int year;
//...
}
```

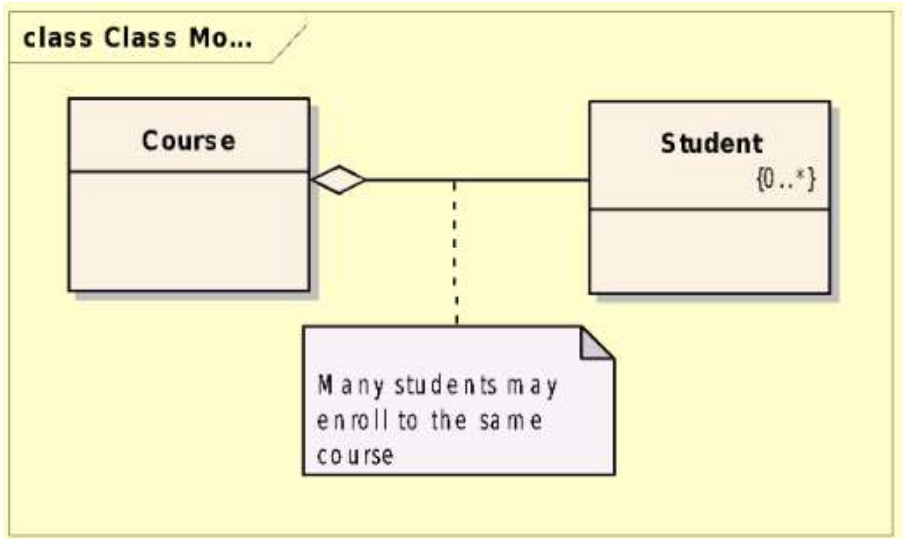
```
public class Person{
  private String name;
  private Date birthDate
  public Person(String name, Date
birthDate){
  this.name = name;
  this.birthDate = birthDate; }
//...
}
```



Relations entre les classes

Implémentation d'une relation *one-to-many*

```
public class Student{
        private final long ID;
        private String firstname;
        private String lastname;
        //...
}
```




```
public class Course{
    private final long ID;
    private String name;
    public static final int MAX_STUDENTS=100;
    private Student[] enrolledStudents;
    private int numStudents;
    //...
}
```

Relations entre les classes

Implémentation d'une relation *one-to-many*

```
public class Course{
 private final long ID;
 private String name;
 public static final int MAX_STUDENTS = 100;
 private Student[] enrolledStudents;
 private int numStudents;
 public Course( long ID, String name ){
    this.ID = ID;
    this.name = name;
    enrolledStudents = new Student[
MAX_STUDENTS ];
}
public void enrollStudent( Student student ){
    enrolledStudents[ numStudents ] = student;
    ++numStudents;
}
//...
}
```

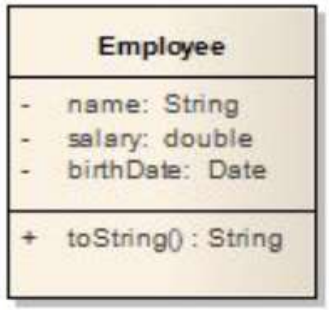
```
public class Course{
 private final long ID;
 private String name;
 private ArrayList<Student> enrolledStudents;
 public Course( long ID, String name ){
   this.ID = ID;
   this.name = name;
  enrolledStudents = new ArrayList<Student>();
 }
 public void enrollStudent(Student student ){
   enrolledStudents.add(student);
 }
//...
}
```

Chapitre 3

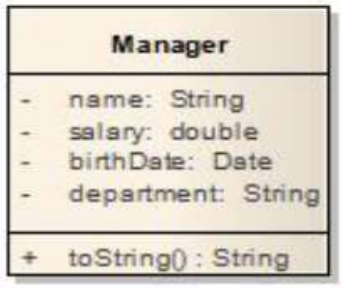
Chapitre 3 – Héritage et Polymorphisme

- Héritage et spécialisation
- Redéfinition de méthodes
- Polymorphisme statique/dynamique
- Classes abstraites
- Interfaces en Java Exemple illustratif

Problème : répétition dans les implémentations




```
public  class Employee{
    private String name;
    private double salary;
    private Date birthDate;

    public String toString(){
       //...
    }
}
```

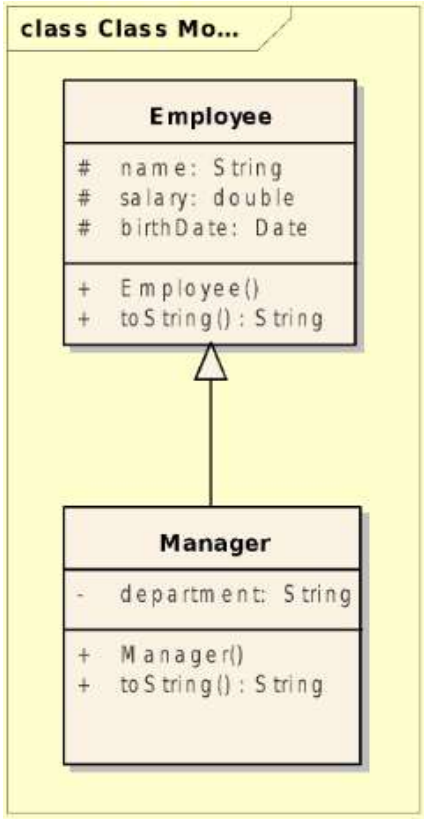



```
public  class Manager{
    private String name;
    private double salary;
    private Date birthDate;
    private String department;

    public String toString(){
       //...
    }
}
```

Héritage, Polymorphisme

Solution : l'héritage

```
public class Employee{
        protected String name;
        protected double salary;
        protected Date birthDate;
        public Employee( … ){
                // …
        }
        public String toString(){
                //...
        }
}
```




```
public class Manager extends Employee{
        private String department;
        public Manager( … ){
                // …
        }
        public String toString(){
                // …
        }
}
```

Héritage, Polymorphisme

Héritage

- **Syntaxe :**

```
<modifier> class <name> extends <superclass>{<declaration*> }
```

```
public class Manager extends Employee{
}
```

- **La sous classe hérite des données et des méthodes du classe parent**
- **N'hérite pas des constructeurs du classe parent**
- **Opportunités :**

1) **ajouter de nouvelles données**
2) **ajouter de nouvelles méthodes**
3) **remplacer les méthodes héritées (polymorphisme)**

Héritage, Polymorphisme

Appel de constructeurs de classe parent

```
public class Employee{
        protected String name;
        protected double salary;
        protected Date birthDate;
        public Employee( String name, double
salary, Date birthDate){
                this.name = name;
                this.salary = salary;
                this.birthDate = birthDate;
        }
//...
}
```

```
public class Manager extends Employee{
        private String department;
        public Manager(String name,double
salary, Date birthDate,
String department){
            super(name, salary, birthDate);
            this.department = department;
        }
 //...
}
```

Héritage, Polymorphisme

Contrôle d'accès

Modifier	Same Class	Same Package	Subclass	Universe
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Polymorphisme – Méthodes de remplacement

- Une sous-classe peut modifier le comportement hérité d'une classe parent
- Une sous-classe peut créer une méthode avec fonctionnalité différente de celle du parent méthode mais avec le :
  - même nom
  - même liste d'arguments
  - presque le même type de retour

```
public class Employee{
        protected String name;protected double salary;
        protected Date birthDate;
        public Employee( … ){        // …        }
        public String toString(){
        return "Name: "+name+" Salary: "+salary+" B. Date:"+birthDate;
        }
}
public class Manager extends Employee{
        private String department;
        public Manager( … ){ // …   }
        @Override
        public String toString(){
        return "Name: "+name+" Salary: "+salary+" B.
        Date:"+birthdate+" department: "+department;
        }
}
```

Polymorphisme – Appel de méthodes remplacées

```
public class Employee{
        protected String name;
        protected double salary;
        protected Date birthDate;
        public Employee( … ){
                // …
        }
        public String toString(){
                return “Name: “+name+” Salary:
”+salary+” B. Date:”+birthDate;
        }
}
```

```
public class Manager extends Employee{
        private String department;
        public Manager( … ){
                // …
        }
        public String toString(){
                return super.toString() + ”
department: ”+department;
        }
}
```

En Java, une méthode surchargée ne peut pas avoir une visibilité plus restreinte que la méthode qu'elle remplace.

```
public class Parent{
public void foo(){}
}
public class Child extends Parent{
private void foo(){} //illégal
}
```

Polymorphisme – Substitution de Liskov

- Le **principe de substitution de Liskov** stipule qu'une instance d'une classe dérivée doit pouvoir remplacer une instance de sa classe de base sans altérer le bon fonctionnement du programme.
- En d'autres termes, une sous-classe doit être substituable à sa classe de base sans introduire de comportements incorrects ou inattendus.

```
// Classe de base
public abstract class Shape {
public abstract double getArea();
}
```

```
// Sous-classe Rectangle
public class Rectangle extends Shape {
private double width;
private double height;

public Rectangle(double width, double height)
 {
this.width = width;
this.height = height;
}
@Override
public double getArea() {
return width * height;
}
}
```

Polymorphisme – Substitution de Liskov

```java
// Sous-classe Square (Carré)
public class Square extends Rectangle {
public Square(double side) {
super(side, side);
}
}
```

```java
public class MainApp {
public static void main(String[] args) {
Shape rectangle = new Rectangle(5, 10);
Shape square = new Square(5);
System.out.println("Surface du rectangle : " +
rectangle.getArea());
System.out.println("Surface du carré : " +
square.getArea());
}
}
```

Le principe de substitution de Liskov (LSP) impose qu'une sous-classe puisse remplacer sa super-classe sans modifier le comportement attendu.

- Les sous-classes doivent respecter les règles et invariants définis par la classe de base.
- Si une sous-classe change le comportement de manière imprévue (ex : un Square qui casse la logique de Rectangle), cela viole le LSP et provoque des erreurs.

Héritage, Polymorphisme

Supposons que on a ces classes en Java :

```java
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int w) { this.width = w; }
    public void setHeight(int h) { this.height = h; }

    public int area() {
        return width * height;
    }
}

class Square extends Rectangle {

    @Override
    public void setWidth(int w) {
        this.width = w;
        this.height = w;  // Maintient la propriété carré
    }

    @Override
    public void setHeight(int h) {
        this.height = h;
        this.width = h;  // Maintient la propriété carré
    }
}
```

Si une méthode attend un Rectangle et fait :

```java
public static void resize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    System.out.println(r.area());
}
```

1) Que se passe-t-il si on passe à *resize* un Rectangle ?

2) Que se passe-t-il si on passe à *resize* un Square ?

3) **Le Square respecte-t-il le principe de Substitution de Liskov ?** Pourquoi ?

Héritage, Polymorphisme

Tableaux hétérogènes

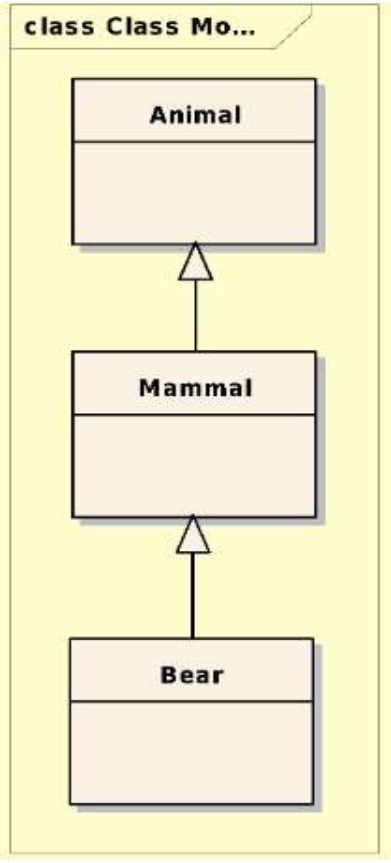
```
Employee emps[] = new Employee[100];
emps[0]=new Employee();
emps[1]=new Manager();
emps[2]=new Employee();
// …
// print employees
for(Employee e:emps)
        {
                System.out.println(e.toString());
        }
// count managers
int counter = 0;
for(Employee e:emps){
        if( e instanceof Manager ){
                ++counter;
        }
}
```

Héritage, Polymorphisme

Type statique ou dynamique d'une référence

```
// static (compile time) type est : Employee
Employee e;
// dynamic (run time) type est : Employee
e = new Employee();
// dynamic (run time) type est : Manager
e = new Manager();

 Employee e = new Manager(“Johann”,3000,
 new Date(10,9,1988),”sales”);
 System.out.println( e.getDepartment() );// ERROR
 //Solution
 System.out.println( ((Manager) e).getDepartment() );// CORRECT
 //Better Solution
 if( e instanceof Manager ){
 System.out.println( ((Manager) e).getDepartment() );
 }
```

L'opérateur instanceof



- L'opérateur **instanceof** est utilisé pour tester si un objet est une instance d'une classe spécifique ou d'une sous-classe de cette classe.

- C'est un opérateur binaire qui renvoie un booléen : true si l'objet est une instance de la classe spécifiée, et false sinon.

```
Animal a = new Bear();
//expressions
a instanceof Animal → true
a instanceof Mammal → true
a instanceof Bear → true
a instanceof Date → false
```

Polymorphisme - Méthodes de surcharge

- Polymorphisme : la capacité d'avoir de nombreuses formes différentes
- Surcharge de méthodes :
  - méthodes ayant le même nom,
  - la liste d'arguments doit différer,
  - les types de retour peuvent être différents.
- Exemple :

public void println (int i)
public void println (float f)
public void println (String s)

```
public class Employee {
protected String name; protected double salary;
protected Date birthDate;
public Employee(String name, double salary, Date
birthDate) {
        this.name = name;
        this.salary = salary;
        this.birthDate = birthDate;
}

public Employee(String name, double salary) {
        this(name, salary, null);
        }

public Employee(String name, Date birthDate) {
        this(name, 1000, birthDate);
        }
        // ...
}
```

Polymorphisme

La capacité à avoir de nombreuses formes différentes

- **Surcharge de méthodes**
  - même nom, signature différente
  - par exemple, une classe ayant plusieurs constructeurs
  - polymorphisme à la compilation (polymorphisme statique)

- **Redéfinition de méthodes**
  - même nom, même signature
  - par exemple, toString()
  - polymorphisme à l'exécution (polymorphisme dynamique)

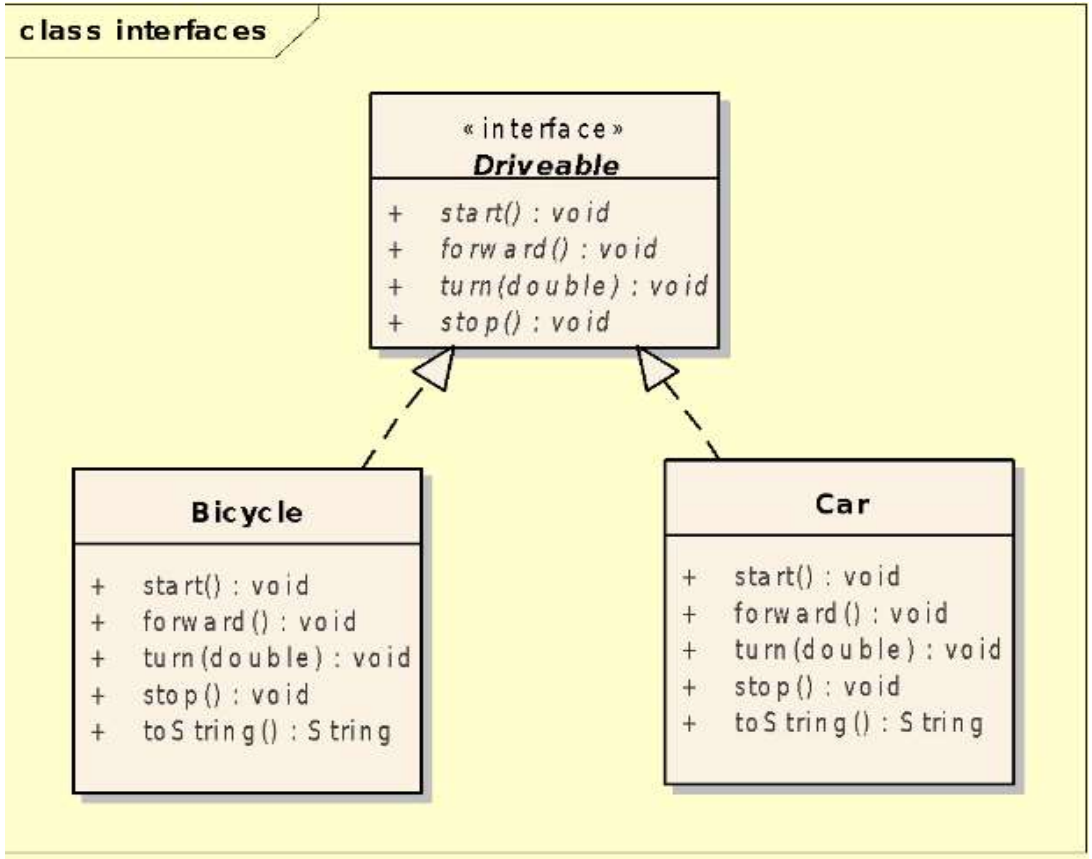
Les interfaces

Les interfaces

- **Définit des types**
- **Déclare un ensemble de méthodes (sans implémentation !)**

  TAD - Type Abstrait de Données

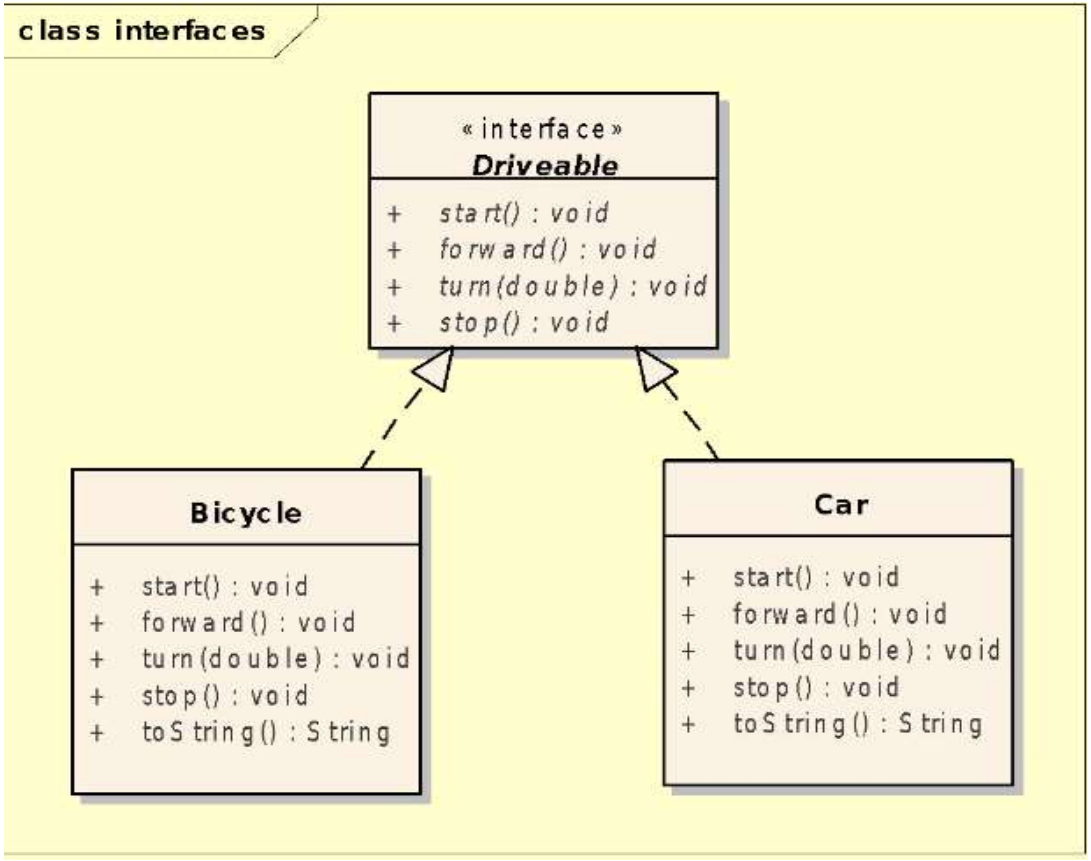
- **Sera implémentée par des classes**



Les interfaces

Les interfaces

- Contient **des déclarations de méthode** et peut contenir **des constantes**
- Toutes les **méthodes sont publiques**
- Les interfaces sont **de pures classes abstraites** → ne peuvent pas être instancié
- Les classes d'implémentation doivent **implémenter tous les méthodes** déclarées dans l'interface
- Une classe peut étendre une seule classe mais peut implémenter n'importe quel nombre d'interfaces



Les interfaces

Interface Itérateur

- Permet de parcourir séquentiellement une collection
- Déclare un ensemble de méthodes pour la navigation dans une collection
  - **hasNext():** Vérifie s'il y a un élément suivant dans la collection.
  - **next():** Renvoie l'élément suivant dans la collection.
  - (Optionnel) **remove():** Supprime l'élément courant de la collection (peut ne pas être implémenté dans toutes les collections).
- Utilisée dans de nombreuses classes de collections en Java
- Par exemple : ArrayList, LinkedList, HashSet, etc

```
List<String> l1 = new ArrayList<>();
l1.add("Welcome");
l1.add("to");
l1.add("Java");
Iterator<String> it = l1.iterator();
while(it.hasNext())
 {
    System.out.print( it.next() + "
");
 }
-----------------------------------
for(String str:l1)
 {
    System.out.print( str + " ");
 }
```

Les interfaces

Questions/réponses

Sélectionnez les affirmations correctes !

a) Driveable a;

b) Driveable a = new Driveable();

c) Driveable t[] = new Driveable[3];

d) public void conduire(Driveable d);

e) Driveable a = new Voiture();

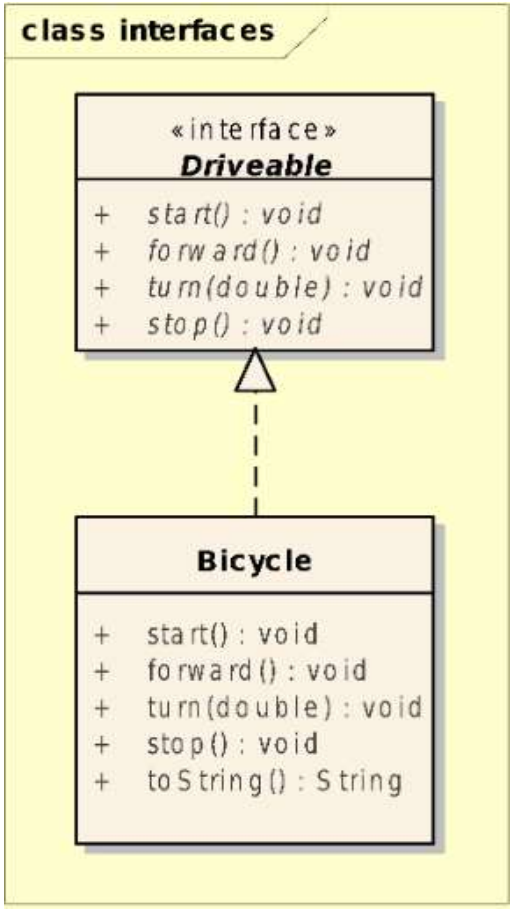
f) interface Driveable extends Voiture {}

g) public class Test implements Driveable { public void drive() {} }

Les interfaces

Interfaces vs classes

- **Interface :**
  - Type défini par l'utilisateur
  - Ensemble de méthodes
  - Aucune implémentation fournie
  - Ne peut pas être instancié
- **Classe :**
  - Type défini par l'utilisateur
  - Ensemble de données et méthodes
  - Toutes les méthodes sont mises en œuvre
  - Peut être instancié



Les interfaces

Argument polymorphe

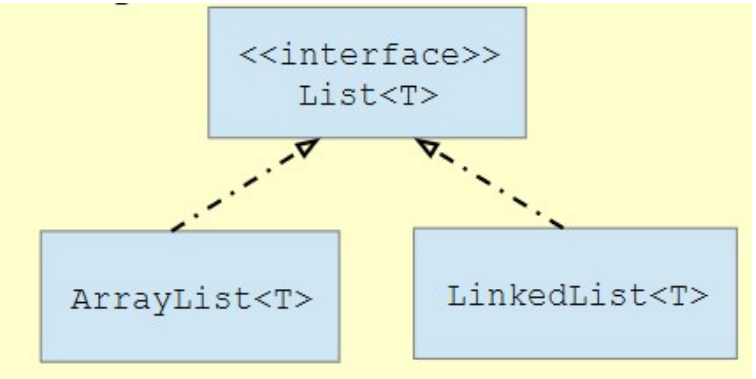
- Permet à une méthode de recevoir un argument qui peut être de plusieurs types différents, tant que ces types sont liés par un héritage ou une interface commune.
- Permet de concevoir des systèmes flexibles et extensibles, où de nouveaux types d'objets peuvent être introduits sans modifier le code existant, à condition qu'ils respectent le contrat défini par l'interface.

```
public class Utils{
  public static void moveMe(Driveable v)
        {
        v.start();
        for( int i=0; i<12; ++i)
              {
              v.turn(15);
              }
        v.stop();
        }
}
Utils.moveMe(new Bicycle());
Utils.moveMe(new Car());
```

Les interfaces

Argument polymorphe

```
public class Utils{
	public static void printIt(List<String> list){
	for( String s: list ){
		System.out.println( s );
		}
	}
}
List<String> l1 = new ArrayList<>();
//add elements to l1
Utils.printIt(l1);
List<String> l2 = new LinkedList<>();
//add elements to l2
Utils.printIt(l2);
```



Les interfaces

Méthode par défaut de l'interface Java

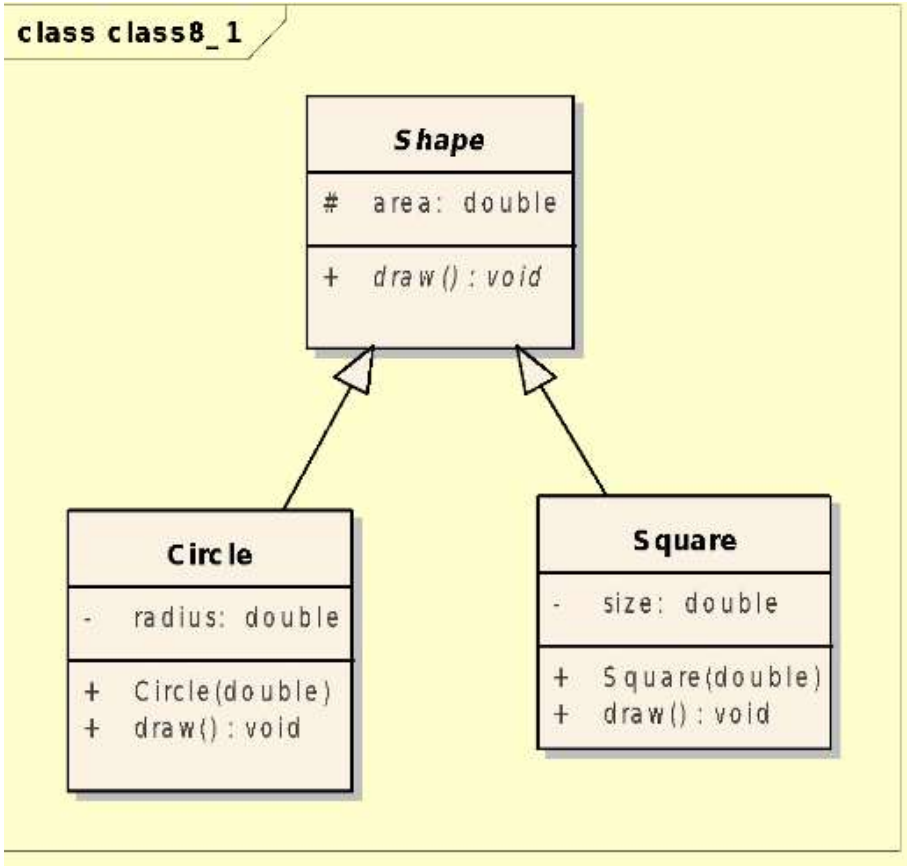
```
public interface Animal {
// Abstract method
void eat();

// Implemented method
default void log( String str ){
System.out.println(
“Animal log: “+str);
}
}
```

```
public class Bear implements Animal {
// Mandatory!!!
void eat(){
System.out.println(“Bear eats”);
}
// It is not mandatory to provide
// implementation for the log method
}
```

Les interfaces abstraits

Abstract Classes

- Peut contenir du résumé et implémenté les méthodes aussi
- Peut contenir des données
- Ne peut pas être instancié
- Sont conçus pour le sous-classement



Les interfaces abstraits

Abstract Classes

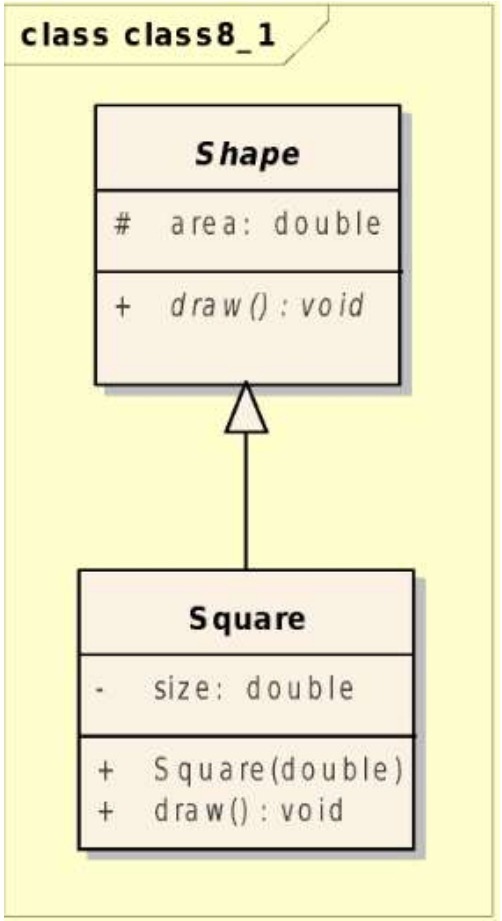
```
public abstract class Shape {
        protected double area;

        public abstract void draw();
        }

        public class Square extends Shape {
        private double size;

public Square(double size) {
        this.size = size;
        this.area = size * size;
        }
@Override
        public void draw() {
        System.out.println("I am a square");
        }
}
```

Les interfaces abstraits

Abstract Classes vs. Classes

Classe abstraite :
- Type défini par l'utilisateur
- Ensemble de données et méthodes
- Méthodes abstraites et implémentées
- Ne peut pas être instancié
- Conçu pour être sous-classé

Classe:
- Type défini par l'utilisateur
- Ensemble de données et méthodes
- Toutes les méthodes sont implémentées
- Peut être instancié



Les interfaces abstraits

Abstract Classes vs. Classes vs. Interfaces

	Interface	Abstract class	Class
Abstract method	Yes	Yes	No
Implemented method	No Yes(since Java 8)	Yes	Yes
Attribute (data)	No	Yes	Yes
Constants (final)	Yes	Yes	Yes

Les interfaces

Adaptez le programme du slide 62 afin de créer une classe abstrait (Shape) pour respecter le principe de Substitution de Liskov (LSP).

Les interfaces


```java
// Classe abstraite Shape
abstract class Shape {
    public abstract int area();
}

// Classe Rectangle qui étend Shape
class Rectangle extends Shape {
    protected int width;
    protected int height;

    public void setWidth(int w) { this.width = w; }
    public void setHeight(int h) { this.height = h; }

    public int getWidth() { return width; }
    public int getHeight() { return height; }

    @Override
    public int area() {
        return width * height;
    }
}
```

```java
// Classe Square qui étend Shape
class Square extends Shape {
    private int side;

    public void setSide(int side) { this.side = side; }
    public int getSide() { return side; }

    @Override
    public int area() {
        return side * side;
    }
}

// Méthode resize adaptée pour respecter LSP
public static void resize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    System.out.println(r.area());
}

// Méthode pour redimensionner un carré
public static void resizeSquare(Square s, int newSide) {
    s.setSide(newSide);
    System.out.println(s.area());
}
```

Les interfaces

Tri et interfaces, Tri collections

- Tri des chaînes, primitives
  - Tableaux.sort()
  - Collections.sort()

- Trier les types définis par l'utilisateur
  - L'interface Comparable
  - L'interface du Comparator

- Trier les objets selon leur ordre naturel
  - L'interface Comparable

- Trier des objets à l'aide d'un comparateur
  - L'interface Comparator

Les interfaces

L'interface Comparable<T>

- **Comparable** est utilisée pour définir un ordre naturel pour les objets d'une classe, elle contient une seule méthode :

```
interface Comparable {
        int compareTo(Object o);
}
```

x.compareTo(y) retourne un entier :

- 0 : **x** est considéré égal à **y** dans l'ordre défini
- positif : **x** est considéré comme "plus grand" que **y** dans l'ordre défini
- négatif : **x** est considéré comme "plus petit" que **y** dans l'ordre défini

- La méthode ***compareTo*** compare l'objet courant (x) avec un autre objet (y) passé en paramètre. Les résultats de cette comparaison indiquent comment **x** se situe par rapport à **y** dans l'ordre défini.

Les interfaces

L'interface Comparable<T>

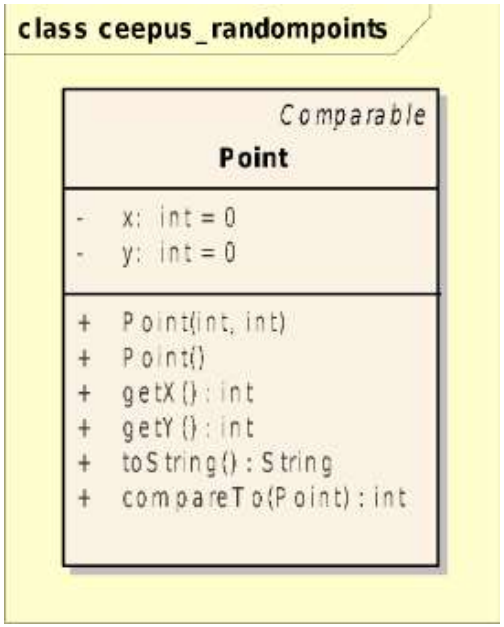
```
interface Comparable<T> {
        int compareTo(T o);
 }
```

- Les tentatives d'utiliser un type différent sont capturées au moment de la compilation!!!
- Cette méthode permet de comparer l'objet courant avec un autre objet du même type passé en paramètre et retourne un entier représentant la relation entre les deux objets dans l'ordre défini.
- Les tentatives d'utiliser un type différent sont capturées au moment de la compilation, ce qui permet de détecter les erreurs de typage dès la phase de développement plutôt qu'à l'exécution.

Les interfaces

L'interface Comparable<T>

```
public class Point implements Comparable<Point> {
        // ...
        @Override
        public int compareTo(Point o) {
                if (o == null)
                        throw new NullPointerException();
        if (this.x == o.x && this.y == o.y) {
                return 0;
}
        if (this.x == o.x) {
                return Integer.compare(this.y, o.y);
        }
        return Integer.compare(this.x, o.x);
}
}
```



Les interfaces

Différence entre equals et comprable<T>

equals() : Déterminer si deux objets sont égaux d'un point de vue logique, c'est-à-dire qu'ils représentent la même valeur.

```
String s1 = new String("Bonjour");
String s2 = new String("Bonjour");

System.out.println(s1.equals(s2)); // true (même contenu)
System.out.println(s1 == s2); // false (objets différents en mémoire)
equals() compare le contenu logique, pas la référence mémoire
```

compareTo() : Déterminer l'ordre relatif entre deux objets (utilisé pour le tri, par exemple dans TreeSet, TreeMap, Collections.sort()).

```
String s1 = "Apple";
String s2 = "Banana";
System.out.println(s1.compareTo(s2)); // < 0 car "Apple" vient avant "Banana"
System.out.println(s2.compareTo(s1)); // > 0
System.out.println(s1.compareTo("Apple")); // 0
```

Les interfaces

L'interface Comparable<T> - cohérence

- Si une classe redéfinit la méthode ***equals***, il est conseillé (mais non obligatoire) que ***a.equals(b)*** soit vrai exactement quand ***a.compareTo(b) == 0***

- Si ***equals*** et ***compareTo*** ne sont pas cohérents, cela peut entraîner des comportements inattendus lors de l'utilisation de collections triées (comme **TreeSet** ou **TreeMap**) ou d'algorithmes de tri. Par exemple :

    - Un **TreeSet** utilise **compareTo** pour maintenir ses éléments dans l'ordre. Si deux objets sont considérés égaux selon **equals** mais pas selon **compareTo**, le **TreeSet** pourrait contenir des doublons.
    - Inversement, si deux objets sont considérés égaux selon compareTo mais pas selon **equals**, des collections basées sur **equals** (comme **HashSet**) pourraient se comporter de manière incohérente

Les interfaces

L'interface Comparable<T> - cohérence

```
import java.util.*;
public class Person implements Comparable<Person> {
private int id;
private String name;
public Person(int id, String name) {
this.id = id;
this.name = name; }
public int getId() {
return id; }
public String getName() {
return name; }
@Override
public boolean equals(Object o) {
if (this == o) return true;
if (o == null || getClass() != o.getClass()) return
false;
Person person = (Person) o;
return id == person.id; }
@Override
public int hashCode() {
return Objects.hash(id);
}
```

```
@Override
public int compareTo(Person other) {
return this.name.compareTo(other.name); }
@Override
public String toString() {
return "Person{id=" + id + ", name='" + name +
"'}";}
public static void main(String[] args) {
Person p1 = new Person(1, "Alice");
Person p2 = new Person(1, "Bob");
Person p3 = new Person(2, "Charlie");
Set<Person> hashSet = new HashSet<>();
hashSet.add(p1); hashSet.add(p2);
hashSet.add(p3);
System.out.println("HashSet: " + hashSet);
Set<Person> treeSet = new TreeSet<>();
treeSet.add(p1); treeSet.add(p2);
treeSet.add(p3);
System.out.println("TreeSet: " + treeSet);
}}
```

HashSet: [Person{id=1, name='Alice'}, Person{id=2, name='Charlie'}]
TreeSet: [Person{id=1, name='Alice'}, Person{id=1, name='Bob'}, Person{id=2, name='Charlie'}]

Les interfaces

L'interface Comparator<T>

Que faire si nous avons besoin de plusieurs critères de tri ?

- Classe Point
  - Tri par x puis par y
  - Tri par y puis par x
  - Tri par la distance à l'origine (0,0)
- Pour chaque classe, nous ne pouvons définir qu'un seul ordre naturel via l'interface Comparable
- Nous pouvons définir un nombre illimité d'ordres en utilisant l'interface Comparator

```
interface Comparator<T> {
        int compare (T x, T y);
}
```

Les interfaces

L'interface Comparator<T>

Méthodes Clés de Comparator<T>

- **compare(T o1, T o2)** : Compare deux objets et retourne un entier.
- **reversed()** : Retourne un Comparator avec l'ordre inversé.
- **thenComparing(Comparator<? super T> other)** : Combine plusieurs comparateurs pour un tri secondaire.
- **naturalOrder()** : Retourne un Comparator pour l'ordre naturel des éléments (si applicable).

Les interfaces

L'interface Comparator<T> - Anonymous inner class

```
import java.util.*;
class Person {
private String name;
private int age;
// Constructeur
public Person(String name, int age) {
this.name = name;
this.age = age; }
public int getAge() {
return age;
}
public String getName() {
return name; }
// Méthode toString pour afficher les informations
@Override
public String toString() {
return "Person{name='" + name + "', age=" + age +
'}';
}}
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
public class Main {
public static void main(String[] args) {
// Création d'une liste de personnes
List<Person> people = new ArrayList<>();
people.add(new Person("Alice", 30));
people.add(new Person("Bob", 25));
people.add(new Person("Charlie", 35));
// Tri de la liste par âge en utilisant un
Comparator anonyme
Collections.sort(people, new Comparator<Person>() {
@Override
public int compare(Person p1, Person p2) {
return Integer.compare(p1.getAge(), p2.getAge());
} });
// Affichage des personnes triées
for (Person person : people) {
System.out.println(person);
}}}
```

Les interfaces

L'interface Comparator<T> - Lambda

```
import java.util.*;
class Person {
private String name;
private int age;
public Person(String name, int age) {
this.name = name;
this.age = age;}
public int getAge() {
return age;
}
public String getName() {
return name;
}
@Override
public String toString() {
return "Person{name='" + name + "', age=" +
age + '}';
}
}
```

```
public class Main {
public static void main(String[] args) {
ArrayList<Person> people = new
ArrayList<>();
people.add(new Person("Alice", 30));
people.add(new Person("Bob", 25));
people.add(new Person("Charlie", 35));
// Tri par âge avec une lambda
people.sort((p1, p2) ->
Integer.compare(p1.getAge(), p2.getAge()));
// Affichage des personnes triées
for (Person person : people) {
System.out.println(person);
}
}
}
```

Chapitre 4

Chapitre 4 – Gestion avancée

- Gestion des exceptions
- Packages et organisation du code
- Quelques classes utilitaires de la bibliothèque Java

Les Exceptions

Les Exceptions

- Définir des exceptions
- Gestion des exceptions : ***try, catch***, et ***finally***
- Lancer des exceptions : ***throw, throws***
- Catégories d'exceptions
- Exceptions définies par l'utilisateur

```
public class AddArguments {
        public static void main(String[] args) {
                int sum = 0;
                for (String arg : args) {
                  sum += Integer.parseInt(arg);
                }
                System.out.println("Sum: " + sum);
}
```

java AddArguments 1 2 3
Sum: 6
java AddArguments 1 foo 2 3
Exception in thread "main" java.lang.NumberFormatException: For input string: "foo"
atjava.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at adarguments.AddArguments.main(AddArguments.java:line_number)
Java Result: 1

Les Exceptions

Les Exceptions *try-catch*

```
public class AddArguments2 {
        public static void main(String[] args) {
                try{
                        int sum = 0;
                        for( String arg: args ){
                                sum += Integer.parseInt(arg);
                }
                System.out.println( "Sum: "+sum );
        } catch( NumberFormatException e ){
                System.err.println("Non-numeric argument");
                }
        }
}
```

java AddArguments2 1 foo 2 3

Non-numeric argument

Les Exceptions

Les Exceptions *try-catch*

```
public class AddArguments3 {
public static void main(String[] args) {
int sum = 0;
for( String arg: args ){
try{
sum += Integer.parseInt( arg );
} catch( NumberFormatException e ){
System.err.println(arg+”is not an integer”);
}
}
System.out.println( "Sum: "+sum );
}
}
```

java AddArguments3 1 foo 2 3

foo is not an integer

Sum: 6

```
try{
// critical code block
// code that might throw exceptions
} catch( MyException1 e1 ){
// code to execute if a MyException1 is thrown
} catch( MyException2 e2 ){
// code to execute if a MyException2 is thrown
} catch ( Exception e3 ){
// code to execute if any other exception is
thrown
} finally{
// code always executed
}
}
```

Les Exceptions

Le Bloc *finally*

- Le bloc **finally** est une partie du mécanisme de gestion des exceptions. Il est utilisé pour exécuter du code qui doit être exécuté qu'une exception soit levée ou non dans le bloc ***try***.
- Le bloc **finally** suit toujours un bloc *try* et est souvent utilisé pour nettoyer les ressources, comme fermer des fichiers ou des connexions réseau.

Syntaxe :

```
try {
// Code qui peut lever une exception
} catch (ExceptionType1 e1) {
// Gérer l'exception de type ExceptionType1
} catch (ExceptionType2 e2) {
// Gérer l'exception de type ExceptionType2
} finally {
// Code à exécuter toujours, exception ou non
}
```

Les Exceptions

L'instruction *try-with-resources*

- L'instruction ***try-with-resources***, introduite dans Java 7, est un mécanisme de gestion des ressources qui assure la fermeture automatique des ressources, telles que les fichiers ou les connexions réseau, après leur utilisation.
- Cela simplifie la gestion des ressources et réduit le risque de fuites de ressources

```
try (ResourceType resource = new ResourceType())
        {
                // Code qui utilise la ressource
        } catch (ExceptionType e) {
        // Gérer l'exception
}
```

Les Exceptions

L'instruction *try-with-resources*

Exemple

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class ExempleTryWithResources {
public static void main(String[] args) {
// Le try-with-resources ferme automatiquement la ressource après utilisation
try (BufferedReader lecteur = new BufferedReader(new FileReader("fichier.txt")))
{
String ligne;
while ((ligne = lecteur.readLine()) != null) {
System.out.println(ligne);
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Les Exceptions

Catégories d'exceptions

Exceptions Vérifiées et Non Vérifiées

Exceptions vérifiées (checked exceptions)

• Ce sont les **exceptions que le compilateur oblige à gérer** (soit avec try/catch, soit en les déclarant avec throws).
• Elles héritent de la classe **Exception**, mais **pas** de **RuntimeException**.
👉 Exemples : IOException, SQLException, FileNotFoundException, ClassNotFoundException

Exceptions non vérifiées (unchecked exceptions)

• Ce sont les **exceptions que le compilateur n'oblige pas à gérer**.
• Elles héritent de **RuntimeException** (ou de ses sous-classes).
👉 Exemples :NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, IllegalArgumentException

Les Exceptions

Catégories d'exceptions

Exceptions Vérifiées : exemple

```
import java.io.*;

class Test {
    void lireFichier() throws IOException {
        FileReader f = new FileReader("test.txt");
    }

    void executer() {
        try {
            lireFichier(); // Obligé de gérer IOException
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Les Exceptions

Catégories d'exceptions

Exceptions non Vérifiées : exemple

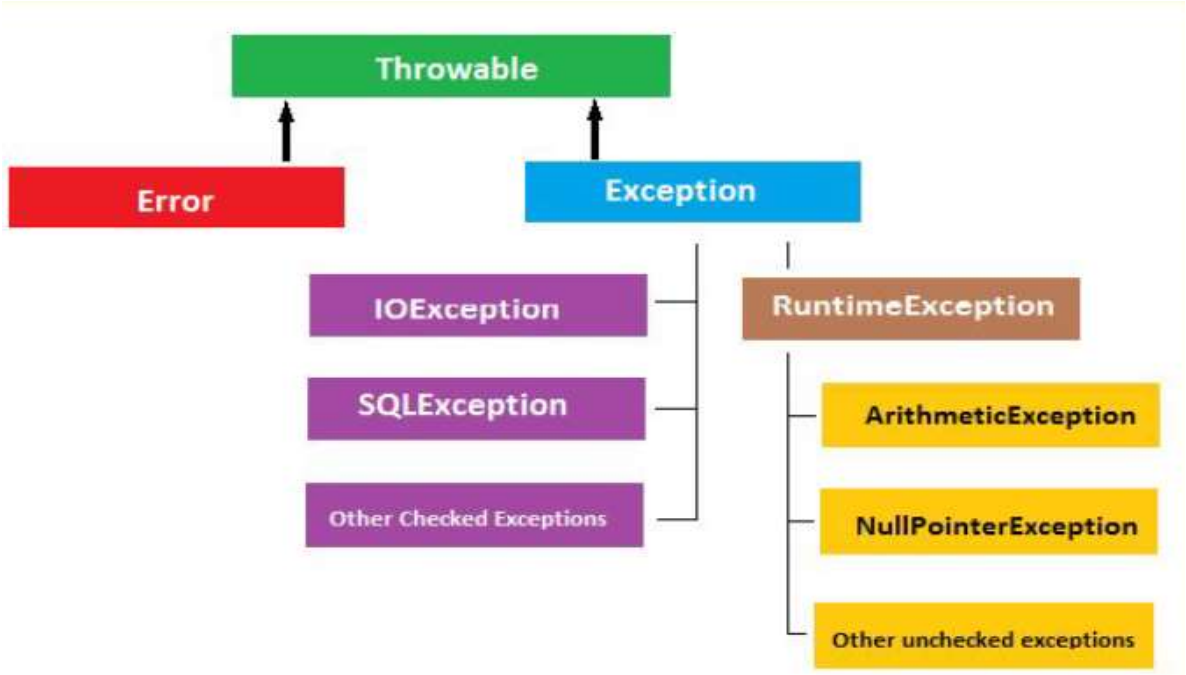
```
class Test {
    void division() {
        int x = 10 / 0; // ArithmeticException non vérifiée
    }

    void executer() {
        division(); // pas besoin de try/catch
    }
}
```

Ici, aucune erreur à la compilation,
mais l'erreur peut survenir à l'exécution.

Les Exceptions

Catégories d'exceptions

Exceptions Vérifiées et Non Vérifiées



Les Exceptions

La clause throw

- throw s'utilise dans le corps d'une méthode, quand on veut provoquer une exception.

```
public void divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Division par
zéro interdite !"); // le programme  s'arrete ici
    }
    System.out.println(a/b);
}
```

divide(10, 0);

Exception in thread "main" java.lang.ArithmeticException: Division par zéro interdite !

Les Exceptions

La clause throws

- La clause "throws" est utilisée pour déclarer les exceptions qui peuvent être levées par une méthode. Elle informe le compilateur des exceptions qui doivent être gérées soit par la méthode elle-même, soit par les méthodes appelantes.
- Vous n'avez pas besoin de déclarer les exceptions d'exécution (non vérifiées).
- Vous pouvez choisir de gérer les exceptions d'exécution (par exemple, IndexArrayOutOfBounds, NullPointerException).

```
public void myMethod() throws IOException {
// Code pouvant générer une IOException
}
```

Les Exceptions

Exemple : La clause throws

```java
import java.io.FileReader;
import java.io.IOException;

public class ExempleThrows {

    // Cette méthode PEUT lancer une IOException
    public static void lireFichier(String nomFichier) throws IOException {
        FileReader reader = new FileReader(nomFichier);
        System.out.println("Fichier ouvert avec succès !");
        reader.close();
    }

    public static void main(String[] args) {
        // Ici, on appelle la méthode qui "throws IOException"
        try {
            lireFichier("exemple.txt");
        } catch (IOException e) {
            System.out.println("Erreur : " + e.getMessage());
        }
    }
}
```

Les Exceptions

Règles pour la méthode substituée (override)

La méthode de substitution peut lancer :

- **Ne lancer aucune exception**, même si la méthode dans la classe parent en lançait.
- **Lancer une ou plusieurs exceptions** que la méthode de la classe parent lançait.
- **Lancer une ou plusieurs sous-classes des exceptions** que la méthode de la classe parent lançait.

La méthode substituée ne peut pas lancer :

- Des exceptions supplémentaires non lancées par la méthode substituée
- Des superclasses des exceptions lancées par la méthode substituée

```
public class StackException extends Exception {
        public StackException(String message) {
                super(message);
        }
}
```

Les Exceptions

Règles pour la méthode substituée (override)

```
class Parent {
    void lire() throws IOException {System.out.println("Lecture fichier parent");
    }
}
// Classe enfant
class Enfant extends Parent {
    // OK : ne lance rien
    @Override
    void lire() {System.out.println("Lecture enfant sans exception");
    }
    //  OK : lance la même exception
    // @Override
    // void lire() throws IOException { ... }

    // OK : lance une sous-classe d'IOException
    // @Override
    // void lire() throws FileNotFoundException { ... }

    // ✕ Erreur : lance une exception non déclarée dans le parent
    // @Override
    // void lire() throws SQLException { ... }
    // ✕ Erreur : lance une super-classe (Exception) de IOException
    // @Override
    // void lire() throws Exception { ... }
}
```

Les Exceptions

Exception définie par l'utilisateur

```
public class Stack {
        private Object elements[]; private int capacity;
        private int size;
        public Stack(int capacity) {
                this.capacity = capacity;
                elements = new Object[capacity];
        }
        public void push(Object o) throws StackException {
                if (size == capacity) {
                        throw new StackException("Stack is full");
                }
                elements[size++] = o;
        }

public Object top() throws StackException {
                if (size == 0) {
                        throw new StackException("Stack is empty");
                }
                return elements[size - 1];
        }
        // ...
}
```

```
Stack s = new Stack(3);
for (int i = 0; i < 10; ++i) {
try {
s.push(i);
} catch (StackException ex) {
ex.printStackTrace();
}
}
```

Classes Imbriquées

- Les classes imbriquées (nested classes) sont des classes définies à l'intérieur d'autres classes.
- Elles sont utilisées pour grouper des classes qui sont logiquement liées et pour améliorer l'encapsulation.
- Il existe quatre types de classes imbriquées en Java : les classes internes (inner classes), les classes internes statiques (static nested classes), les classes locales (local classes) et les classes anonymes (anonymous classes).

Classes Imbriquées

Classes Internes (Inner Classes)

- Ce sont des classes non statiques définies à l'intérieur d'une autre classe.
- Elles peuvent accéder aux membres (y compris les membres privés) de la classe englobante

```
public class OuterClass {
        private String message = "Hello from OuterClass!";
        // Inner class
        public class InnerClass {
        public void display() {
                System.out.println(message); // Accès au membre de la classe englobante
        }
}
        public static void main(String[] args) {
                OuterClass outer = new OuterClass();
                OuterClass.InnerClass inner = outer.new InnerClass();
                inner.display();
        }
}
```

Classes Imbriquées

Classes Internes Statiques (Static Nested Classes)

- Ce sont des classes statiques définies à l'intérieur d'une autre classe.
- Elles ne peuvent accéder qu'aux membres statiques de la classe englobante.

```
public class OuterClass {
        private static String message = "Hello from OuterClass!";
        // Static nested class
        public static class StaticNestedClass {
        public void display() {
                System.out.println(message);//Accès au membre statique de la classe englobante
        }
}
        public static void main(String[] args) {
                OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();
                nested.display();
        }
}
```

Classes Imbriquées

Classes Locales (Local Classes)

- Ce sont des classes définies à l'intérieur d'un bloc, comme une méthode ou un constructeur.
- Elles ne sont visibles que dans le bloc où elles sont définies.

```
public class OuterClass {
public void displayMessage() {
class LocalClass {
public void print() {
System.out.println("Hello from LocalClass!");
}
}
LocalClass local = new LocalClass();
local.print();
}
public static void main(String[] args) {
OuterClass outer = new OuterClass();
outer.displayMessage();
}
}
```

Classes Imbriquées

Classes Anonymes (Anonymous Classes)

- Ce sont des classes sans nom définies et instanciées dans une seule expression.
- Utilisées généralement pour remplacer des interfaces ou des classes avec une seule instance.

```
interface Greeting { void sayHello(); }

public class OuterClass {
        public void greet() {
                Greeting greeting = new Greeting() {
                        public void sayHello() {
                   System.out.println("Hello from Anonymous Class!");
                  }
                };
        greeting.sayHello();
}

        public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.greet();
        }
}
```

Chapitre 5

Chapitre 5 – Programmation concurrente avec les Threads

- Qu'est-ce qu'un thread ?
- Création de threads en Java (héritage de Thread, implémentation de Runnable)
- Cycle de vie d'un thread
- Synchronisation entre threads
- Exemple d'application multi-threads

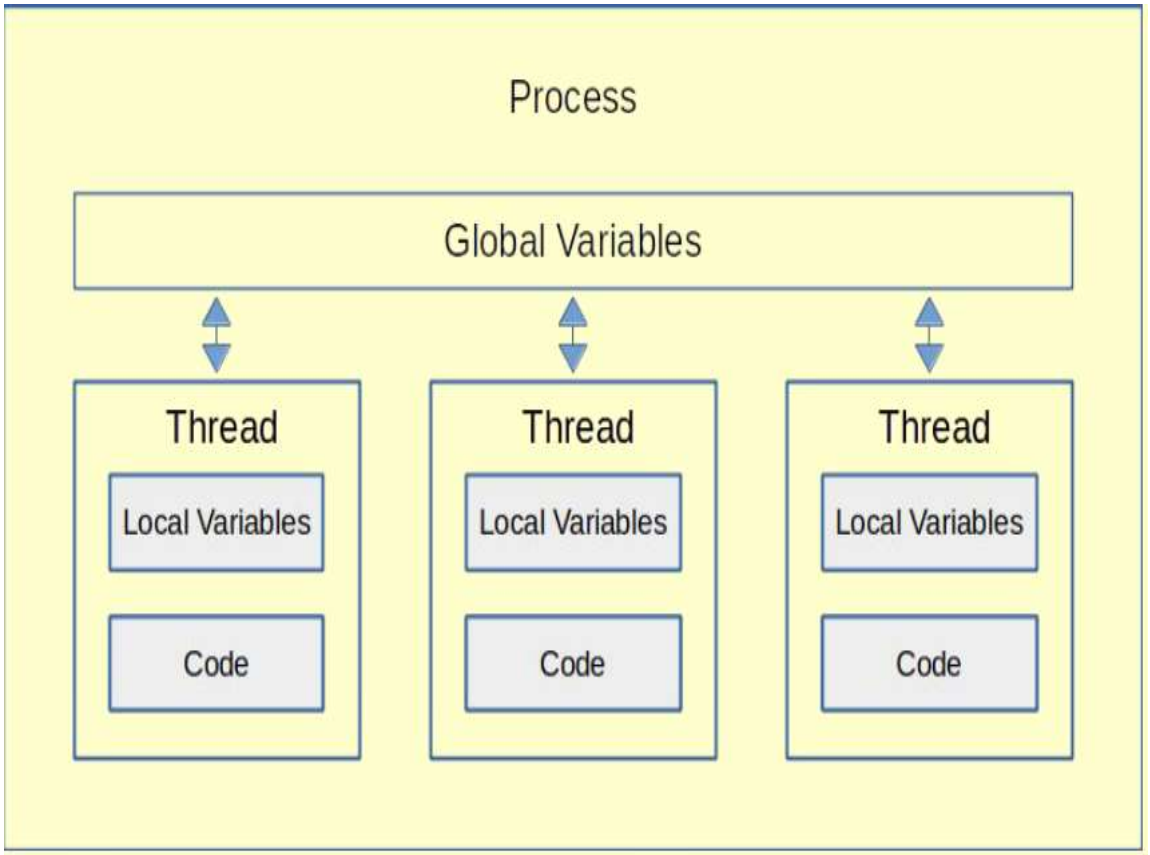
Threads

Systèmes d'Exploitation

- **Processus léger**
  - S'exécute dans l'espace d'adressage d'un processus
  - Possède son propre compteur de programme (PC) et sa propre pile
  - Partage le code et les données avec d'autres threads

Programmation Orientée Objet

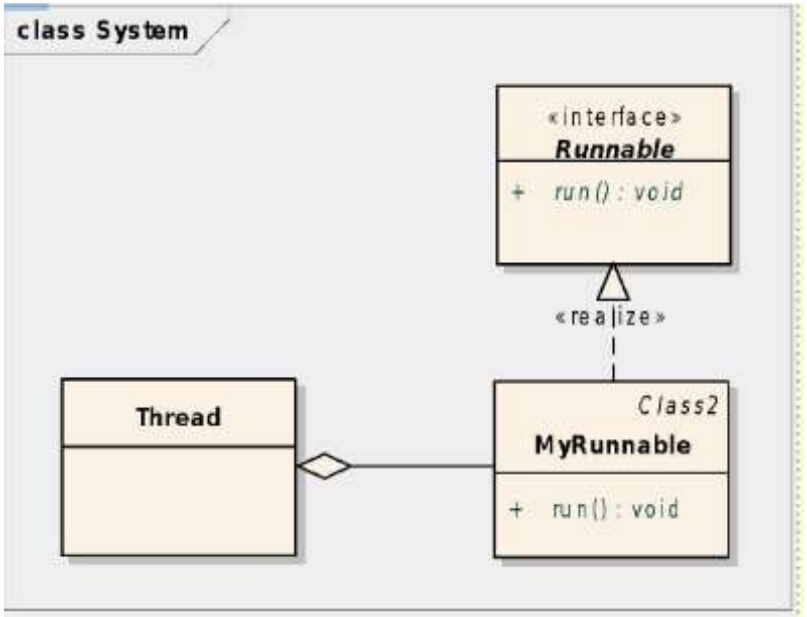
- Un objet : une instance de la classe Thread



Threads

Thread : Création

```
public class MyRunnable implements Runnable {
        private int id;

        public MyRunnable(int id) {
                this.id = id;
        }

        public void run(){
                for( int i=0; i<10; ++i){
        System.out.println("Hello"+id+" "+i);
                }
        }
}
…

MyRunnable r = new MyRunnable(1);
Thread t = new Thread( r );
t.start();
```



Thread : Création

Deux façons principales de créer un thread en Java :

1. En étendant la classe Thread

•Vous pouvez créer un thread en étendant la classe Thread et en redéfinissant la méthode run().

2. En implémentant l'interface Runnable

• Une autre façon consiste à implémenter l'interface Runnable et à passer une instance de la classe qui implémente cette interface à un objet Thread.

Différences entre les deux méthodes

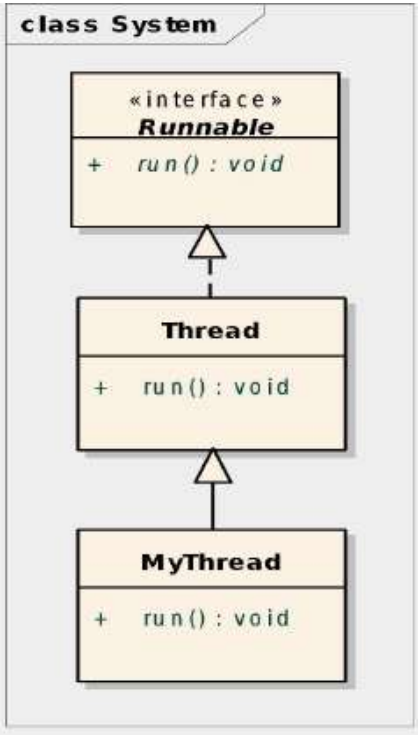
• Lorsque vous **étendez la classe Thread**, vous ne pouvez pas étendre une autre classe car Java ne supporte pas l'héritage multiple.

• En **implémentant Runnable**, vous avez la flexibilité d'étendre une autre classe si nécessaire.

Thread : Création

```
class MyThread extends Thread {
private int id;
public MyThread(int id) {
this.id = id;
}
@Override
public void run() {
for (int i = 0; i < 10; ++i) {
System.out.println("Hello" + id + " " + i);
}
}
}
…
Thread t = new MyThread(1);
t.start();
```

```
public class Test {
public static void main(String[]
args) {
Thread t1 = new MyThread(1);
Thread t2 = new MyThread(2);
t1.start();
t2.start();
}
}
```



Thread : Exemples

```
public class MyFirstRunnable implements
Runnable{
@Override
public void run() {
System.out.println("In a thread");
}
}
```

Usage:

```
Thread thread = new Thread(new
MyFirstRunnable());
thread.start();
System.out.println("In the main Thread");
```

```
public class MyFirstRunnable implements
Runnable{
@Override
public void run() {
System.out.println("In a thread");
}
}
Usage:
Runnable runnable = new MyFirstRunnable();
for(int i = 0; i<25; i++){
new Thread(runnable).start();
}
```

Thread : Exemples

```
public class MyFirstRunnable implements Runnable{
@Override
public void run() {
System.out.println("In a thread");
}
}
Usage:
Thread thread = new Thread(new MyFirstRunnable());
thread.run();
System.out.println("In the main Thread");
```

Thread : Exemples

```
class A {
// Classe avec des méthodes utiles
}
class MaClasse extends A implements Runnable { // Ici, on hérite de A et on implémente Runnable
public void run() {
System.out.println("Exécution du thread");
}
}
public class Test {
public static void main(String[] args) {
MaClasse obj = new MaClasse();
Thread t = new Thread(obj);
t.start();
}
}
```

Operation sur les Thread : sleep

- sleep met toujours en pause l'exécution du thread en cours.
- La durée réelle de mise en veille du thread dépend des minuteries et des planificateurs du système (pour un système occupé, la durée réelle de mise en veille est un peu supérieure à la durée de mise en veille spécifiée).

```
try {
Thread.sleep(1000);
} catch (InterruptedException e){
e.printStackTrace();
}
```

Operation sur les Thread : join()

- La méthode join() est utilisée pour faire en sorte que le thread courant (celui qui appelle join()) attende que le thread cible (celui sur lequel join() est appelé) ait terminé son exécution avant de continuer.

```
Thread t2 = new Thread(new R());
t2.start();
try {
t2.join();
} catch (InterruptedException e){
e.printStackTrace();
}
```

Operation sur les Thread : setPriority()/getPriority()

- Les priorités des threads sont utilisées pour indiquer aux planificateurs de threads quelles tâches devraient être exécutées en premier. Toutefois, les priorités ne garantissent pas un ordre d'exécution strict, car le planificateur de threads dépend de la plateforme sous-jacente et peut ignorer les priorités

```
public class ThreadPriorityRange {
        public static void main(String[] args) {
        System.out.println("Minimal priority : " +Thread.MIN_PRIORITY);
System.out.println("Maximal priority : " +Thread.MAX_PRIORITY);
System.out.println("Norm priority : " +Thread.NORM_PRIORITY);
}
```

Threads

Operation sur les Thread : interrupt()

- La méthode interrompu() peut être appelée sur un thread pour le signaler qu'il doit s'arrêter.
- Cette méthode ne force pas directement l'arrêt du fil, mais elle change son état d'interruption.
- Si le thread est en train d'exécuter certaines méthodes spécifiques comme sleep(), wait(), ou join(), une exception InterruptedException sera levée.

```
private static class ForeverRunnable implements Runnable {
        public void run() {
                while (true) {
System.out.println(Thread.currentThread().getName() +
": " + System.currentTimeMillis());
try {
Thread.sleep(5000);
} catch (InterruptedException e) {
        System.out.println(
        Thread.currentThread().getName() +
        "has been interrupted");
        }
        }
        }
}
```

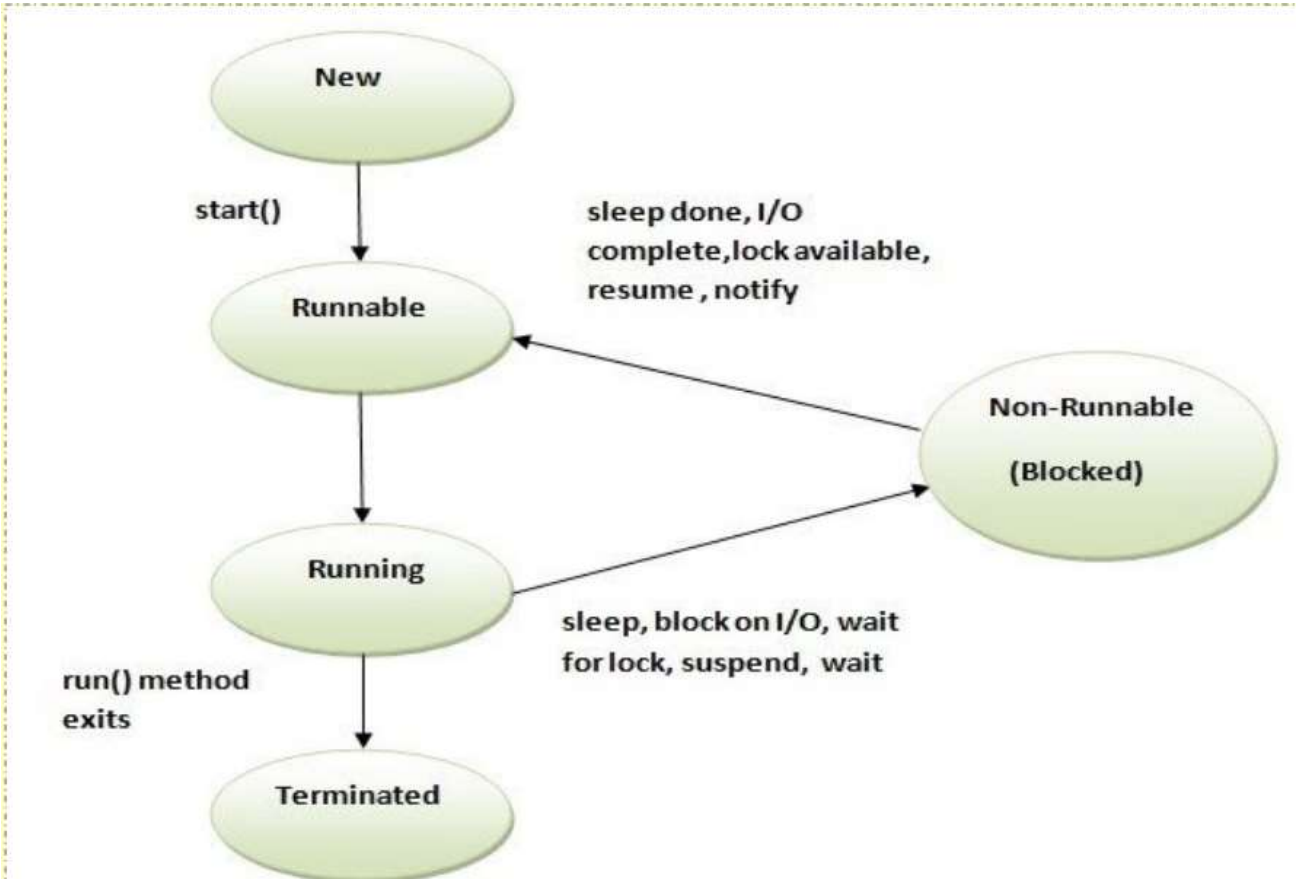
Operation sur les Thread : interrupt()

```
private static class ForeverRunnable implements Runnable {
public void run() {
while (true) {
System.out.println(Thread.currentThread().getName() + ": " + System.currentTimeMillis());
try {
Thread.sleep(5000);
} catch (InterruptedException e) {
System.out.println( Thread.currentThread().getName() +
"has been interrupted");
}
}
}
```

```
public static void main(String[] args) {
Thread t2 = new Thread(new ForeverRunnable());
System.out.println("Current time millis : " +
System.currentTimeMillis());
t2.start();
t2.interrupt();
```

Threads

etat des Threads



Thread : Section critique, synchronisation

```
class Counter {
private int value;
public int getNextValue() {
return ++value;
}
public int getValue(){
return value;
}
}
```

Thread : Section critique, synchronisation

```
Runnable task = new Runnable() {
@Override
public void run() {
for( int i=0; i<10000; ++i) {
counter.getNextValue();
}
}
```

```
Counter counter = new Counter();
Thread t1 = new Thread(task);
Thread t2 = new Thread(task);
t1.start();
t2.start();
try{
t1.join();
t2.join();
} catch( InterruptedException e ){
}
System.out.println("COUNTER: "
+counter.getValue());
```

value++ <--- Pas atomique ! 1. Lisez la valeur actuelle de « valeur » 2. Ajoutez-en un à la valeur actuelle 3. Écrivez cette nouvelle valeur dans « valeur »

Thread : Section critique, synchronisation

Solution

```
public class Counter {
private int value = 0;


public synchronized int getNextValue() {
return value++;
}
}
```

```
public class Counter {
private int value = 0;


public int getNextValue() {
synchronized (this) {
value++;
}
return value;
}
}
```

```
import
java.util.concurrent.atomic.AtomicInteger;


public class Counter {
private AtomicInteger value = new
AtomicInteger(0);


public int getNextValue() {
return value.incrementAndGet();
}


public int getValue(){
return value.intValue();
}
```

Threads

Blocs synchronisés

- Chaque objet contient un verrou unique.
- Le verrou est pris lorsque la section synchronisée est entrée.
- Si le verrou n'est pas disponible, le thread entre dans une file d'attente d'attente.
- Si le verrou est libéré, le thread est repris.

thread-safe

- Une classe est thread-safe si elle se comporte toujours de la même manière lorsqu'elle est accédée par plusieurs threads.
- Les objets sans état (classes immuables) sont toujours thread-safe :
  - *String*
  - *Long*
  - *Double*

Threads


```
class Ressource {
private final String nom;

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

public String getNom() { return nom; }
}

class X {
public static void main(String[] args) {
Ressource ressource1 = new Ressource("Ressource 1");
Ressource ressource2 = new Ressource("Ressource 2");
// Thread A essaie de verrouiller ressource1, puis
ressource2
Thread threadA = new Thread(() -> {
synchronized (ressource1) {
System.out.println("Thread A a verrouillé " +
ressource1.getNom());

try { Thread.sleep(100); } catch (InterruptedException e)
{}

System.out.println("Thread A essaie de verrouiller " +
ressource2.getNom());
synchronized (ressource2) {
System.out.println("Thread A a verrouillé " +
ressource2.getNom());
}
}
});

// Thread B essaie de verrouiller ressource2, puis
ressource1
Thread threadB = new Thread(() -> {
synchronized (ressource2) {
System.out.println("Thread B a verrouillé " +
ressource2.getNom());
try { Thread.sleep(100); } catch (InterruptedException e)
{}
System.out.println("Thread B essaie de verrouiller " +
ressource1.getNom());
synchronized (ressource1) {
System.out.println("Thread B a verrouillé " +
ressource1.getNom());
}
}
});

threadA.start(); threadB.start();
}
}
```