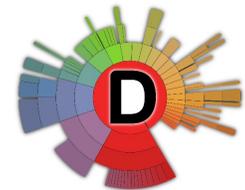


# PROG - ESIR 1

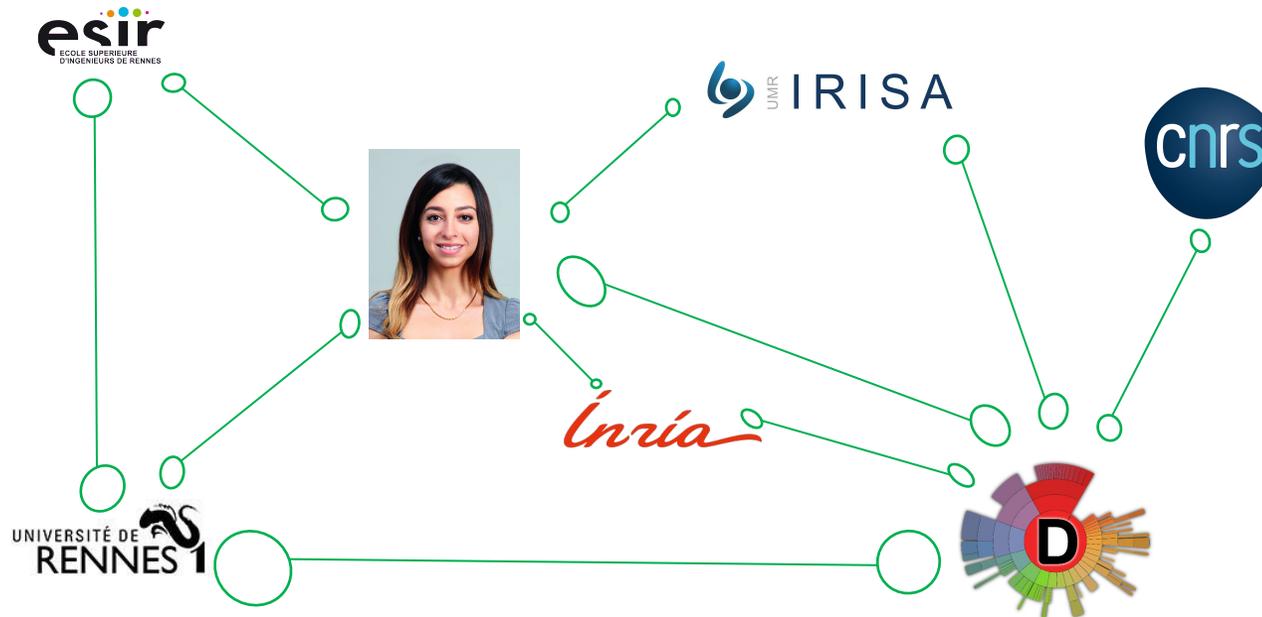
Cours programmation

Octobre 2024



# Qui suis-je ?

- **Maître de Conférences @ Université de Rennes**
  - DiverSE team (University of Rennes, IRISA, Inria)



 <https://stephaniechallita.github.io/>

 <https://www.diverse-team.fr/>

# DiverSE team



- Software engineering group in Rennes
- Software languages, architecture, simulation, variability, testing, resilience eng.
- Applied to smart, heterogeneous, and distributed systems
- ~ 30 members: 13 prof/assoc. prof/researchers, 15 PhD students, 3 postdocs, 1 RSE
- Empirical software approach
- Strong contractual activity with industry

# Introduction : objectifs du cours

- Concepts de la programmation orientée objet
- Mécanismes et fonctionnement des concepts objets : classes, abstraction, liaisons tardives, polymorphisme
- Structures de données standards
- Pratique et implémentation en Java

# Introduction : pourquoi est-ce important ?

- La programmation orientée objet est très utilisée et puissante
- Bonne réutilisation et une bonne extension du code existant
- Briques de base pour des prochains cours plus avancés : patrons de conceptions, méthodes de développement industrielles, architecture logicielle, etc.

# Introduction : organisation du module

- 7 Cours magistraux
- 4 TPs sur 16 séances
- 2 TDs sur 5 séances

# Introduction : modalités d'évaluation

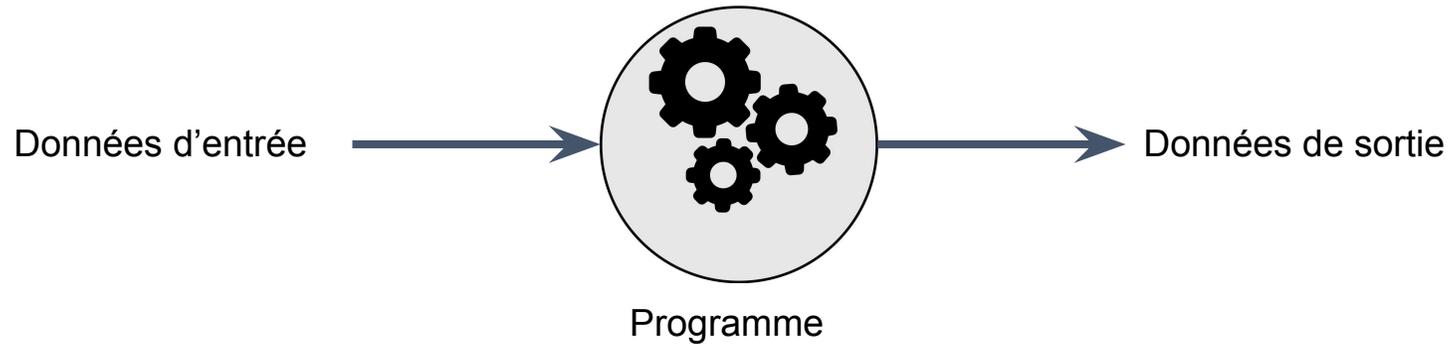
- 1 Contrôle :
  - 17/12/2024 (sur table)
- 2 Devoirs :
  - 28/11/2024 (sur machine)
  - 07/01/2025 (sur machine)
- 0.7CC + 0.3D

# Table des matières

- Rappel de programmation
- Classes et Objets
- Interfaces et Implémentations
- Généricité
- Structures de données
- Héritage

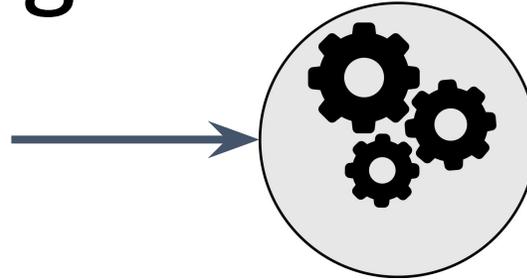
# Rappel de programmation

# Qu'est-ce qu'un programme ?



# Exemples de programmes

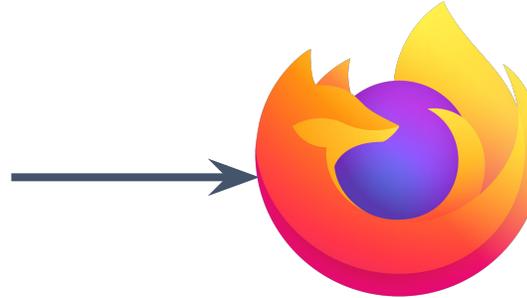
Données d'entrée :  $(3 + 4) * 5$



Calculatrice

"35" est la sortie du programme

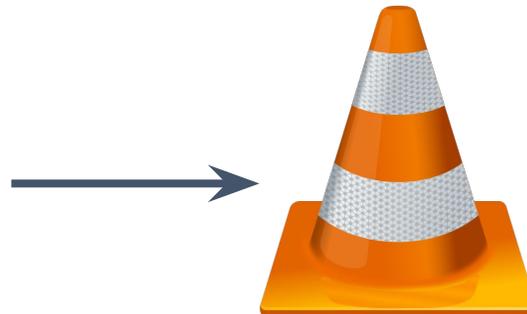
Données d'entrée : Clics de souris et saisies au clavier



Navigateur  
Firefox

Navigation sur des pages internet,  
Communication avec des services  
numérique distants

Données d'entrée : fichiers  
sons (mp3) ou vidéos (mp4)



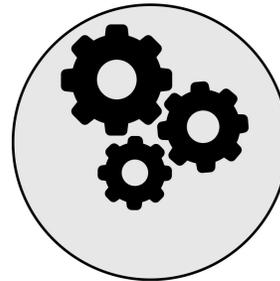
VLC media  
player

Jeu du son et des images vidéos en  
fonction du fichier donné en entrée

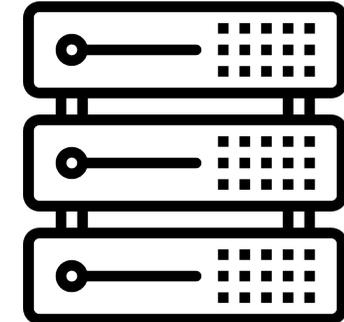
# Programmation



Fichiers sources



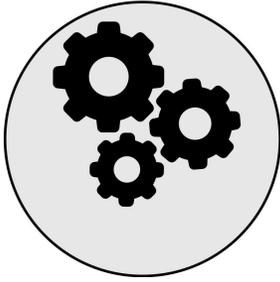
Compilateur ou  
Interpréteur



Machine

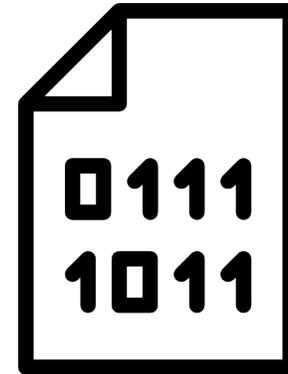
# Compilation

Compilateur



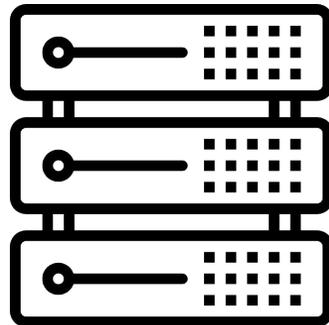
Fichiers sources

Compile



Fichiers binaires

Exécute

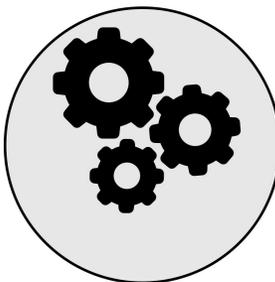


Machine

# Interprétation

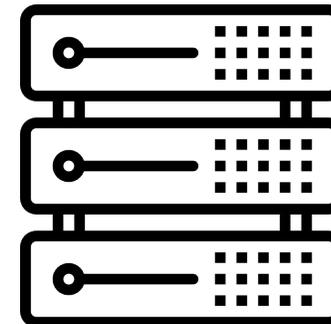


Fichiers sources



Interpréteur

Exécute

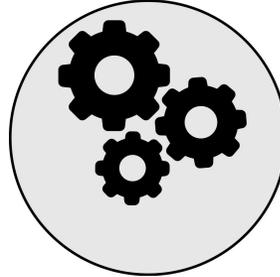


Machine

# Compilation en Java



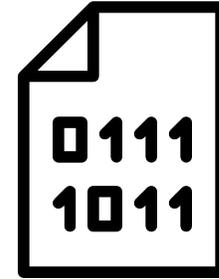
Fichiers sources  
(.java)



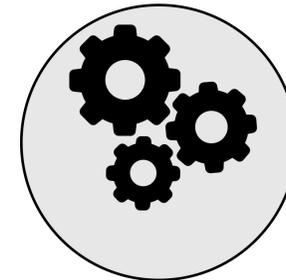
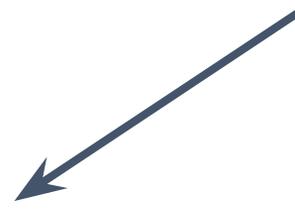
Compilateur  
Java



Compile



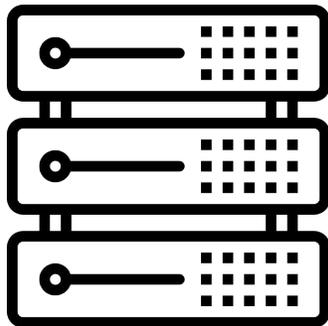
Fichiers binaires  
(.class)



Machine Virtuelle  
Java (JVM)



Exécute



Machine

# Lignes de commande

```
1 # Produit MyFile.class
2 $ javac MyFile.java
3 # Exécute MyFile.class
4 # (dans la ligne de commande, on ne précise pas le .class)
5 $ java MyFile
6 # Avec des dependances :
7 # le répertoire bin contient des fichiers .class
8 # requis pour compiler mon fichier
9 $ javac --classpath bin/ MyFile.java -d bin/
10 # Idem pour l'exécution (-cp est un raccourcis pour --classpath)
11 $ java -cp bin/ MyFile
```

# Rappel Langage Java : variables

```
1 public class MyFile {  
2  
3     public static void main(String[] args) {  
4         int x = 10;  
5         String s = "Message";  
6         int y = x + 10;  
7         System.out.println(s + y);  
8     }  
9  
10 }
```

■ 3 Variables :

x = 10

s = "Message"

y = 20

??

# Rappel Langage Java : variables

```
1 public class MyFile {
2
3     public static void main(String[] args) {
4         int x = 10;
5         String s = "Message";
6         int y = x + 10;
7         System.out.println(s + y);
8     }
9
10 }
```

■ 3 Variables :

x = 10

s = "Message"

y = 20

■ Affiche "Message20"

# Rappel Langage Java : tableaux

```
1 int[] tableau = new int[5];
2 // tableau = [0, 0, 0, 0, 0]
3 tableau[0] = 10;
4 // tableau = [10, 0, 0, 0, 0]
5 tableau[1] = tableau[0] * 2;
6 // tableau = [10, 20, 0, 0, 0]
7 tableau[10] = -1;
8 // Erreur : 10 >= tableau.length
9 tableau = new int[] {1}
10 // tableau = [1]
```

- Déclaration d'un nouveau tableau
- Taille du tableau entre [] -> 5
- Accède au premier élément avec [0]
- Les indices vont de 0 à taille - 1
- taille = tableau.length

# Rappel Langage Java : structure de contrôle if/then/else

```
1 int x = ?;
2 int y;
3 if (x == 5) {
4     y = x * 2;
5 } else if (x > 10) {
6     y = x;
7 } else {
8     y = -1;
9 }
10 System.out.println(x + ";" + y);
```

■ Si x est égale à 5, alors

■  $y = x * 2 = 5 * 2 = 10$

■ Sinon, si x est supérieur à 10, alors

■  $y = x$

■ Sinon  $y = -1$

■ Suivant x, on aura :

■ "5;10" si  $x == 5$

■ "x;x" si  $x > 10$

■ "x;-1" sinon

# Rappel Langage Java : structure de contrôle boucle pour i

```
1 for (int i = 0 ; i < tableau.length ; i++) {  
2     tableau[i] = i;  
3 }
```

■ Initialisation de la  
valeur de la boucle  
**i = 0**

■ Condition d'arrêt  
Tant que i est < à la  
taille du tableau

■ Mise à jour  
On augmente de 1 la  
valeur de i à chaque fin de  
tour de boucle

# Rappel Langage Java : structure de contrôle boucle tant que

```
1 while(i < tableau.length) {  
2     tableau[i] = i;  
3     i++;  
4 }
```

■ Condition d'arrêt

■ Mise à jour externe à la syntaxe de la boucle

# Rappel Langage Java : fonctions

```
1 public static void fonction() {
2     int x = twice(10);
3     // x = 20
4     System.out.println(parameter);
5     // error
6 }
7 public static int twice(int parameter) {
8     System.out.println(parameter);
9     return parameter * 2;
10 }
```

Appel

■ Appel à la fonction **twice()**

■ Déclaration de la fonction **twice()**

# Rappel Langage Java : paramètres et valeur de retour

```
1 public static void fonction() {  
2     int x = twice(10);  
3     // x = 20  
4     System.out.println(parameter);  
5     // error  
6 }  
7 public static int twice(int parameter) {  
8     System.out.println(parameter);  
9     return parameter * 2;  
10 }
```

- Affectation de la valeur de retour à une variable
- Type de retour
- Paramètre de la fonction
- Valeur retournée
- Signature de fonction : type de retour, nom et paramètres

# Rappel Langage Java : portée

```
1 public static void fonction() {
2     int x = twice(10);
3     // x = 20
4     System.out.println(parameter);
5     // error
6 }
7 public static int twice(int parameter) {
8     System.out.println(parameter);
9     return parameter * 2;
10 }
```

■ **parameter** n'est pas accessible depuis la fonction **fonction**

# Rappel Langage Java : point d'entrée

```
1 public class MyFile {  
2  
3     public static void main(String[] args) {  
4         int x = 10;  
5         String s = "Message";  
6         int y = x + 10;  
7         System.out.println(s + y);  
8     }  
9  
10 }
```

- Point d'entrée du programme :  
méthode "main"

# Rappel Langage Java : gestion des erreurs

```
1 public static int functionWithThrows(int parameter)
2     throws IllegalArgumentException {
3     if (parameter < 0) {
4         throw new IllegalArgumentException(
5             "parameter doit être >= 0"
6         );
7     }
8     // contenu de la fonction
9 }
10 public static void function() {
11     int x = ?
12     try {
13         functionWithThrows(x)
14     } catch (IllegalArgumentException e) {
15         // affichage du problème
16         // traitement de l'erreur
17         // procédure de rétablissement
18     }
19 }
```

▪ Déclaration d'Exception potentielle

▪ Lancement d'une nouvelle Exception

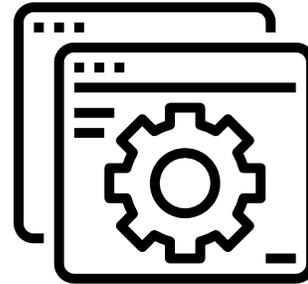
▪ L'appel de la fonction est enveloppé par **try / catch**

▪ Code de traitement de l'erreur

# Rappel : Principe des Tests Unitaires



Exécute



Tests Unitaires

Programme

- 1 test unitaire pour 1 fonctionnalité

- ✓ Test réussit -> Fonctionnalité codée sans bug

- ✗ Test échoue -> Fonctionnalité contenant un bug

# Rappel : JUnit Tests pour Java

```
1 public class CalculTest {  
2     @Test  
3     public void testTwice() {  
4         int x = 10;  
5         int result = twice(x);  
6         assertEquals(20, result);  
7     }  
8 }
```

- Déclaration d'une fonction de test
- Nommage pour indiquer la fonctionnalité testée
- Données d'entrée du test
- Vérification de la valeur retournée

# Classes et Objets

# Motivation : simulation de robot sur carte

```
1 public static void main(String[] args) {
2     // Coordonnées du robot
3     int x = rand.nextInt(N);
4     int y = rand.nextInt(N);
5
6     // Capacité de la batterie du robot
7     final int capBattery = 100;
8     int currentBattery = capBattery;
9
10    // Énergie consommée pour avancer et tourner
11    final int consForward = 5;
12    final int consTurn = 1;
13
14    // Direction du Robot
15    Direction dirRobot = Direction.NORTH;
16
17    // ... Traitement
18 }
```

■ Coordonnées initiales du robot

■ Énergie max et courante

■ Consommation d'énergie pour avancer et pour tourner

■ Direction initiale du robot

# Énumération

```
1 public enum Direction {
2     NORTH, EAST, SOUTH, WEST
3 }
4 //
5 Direction dir = Direction.NORTH;
6 dir = Direction.EAST;
7 dir = Direction.SOUTH;
8 dir = Direction.WEST;
9 switch (dir) {
10     case Direction.NORTH:
11         System.out.println("NORTH");
12         break;
13     case Direction.EAST:
14         System.out.println("EAST");
15         break;
16     //...
17 }
```

■ Définition d'énumération

■ Valeurs possibles

■ Déclaration et affectations

■ Switch sur une énumération

■ Pour chaque cas, un algo est défini

# Motivation : simulation de robot sur carte

```
1 // Si suffisamment d'énergie
2 if (currentBattery >= consForward) {
3     // On change les coordonnées en fonction
4     // de la direction du robot
5     switch dirRobot {
6         case Direction.NORTH:
7             y--;
8             break;
9         case Direction.EAST:
10            x++;
11            break;
12            // ...
13    }
14    // Mise à jour de l'énergie du robot
15    currentBattery -= consForward
16 }
```

■ Condition pour avancer : avoir suffisamment d'énergie

■ Test de la direction courante du robot

■ Mise à jour de la position du robot en fonction de sa direction

■ Mise à jour du niveau courant de batterie

# Motivation : critique du code

- Pas de séparation entre traitement et données
- Difficultés de découpage en fonctions
- Évolutivité compliquée
- Pas de lien entre les données : éparpillement des données, cohérence des données difficile
- Lourd à gérer : Manipulation de **N** robots => remplacer chaque variable par un tableau de taille **N**

# Solution : l'Objet !

- Encapsuler les données dans une même entité : **un objet**
- Fournir des opérations abstraites permettant le traitement et le maintien de la cohérence des données
- Les objets "Robot" regroupent, cachent et maintiennent la cohérence des données qui composent un robot
- Les objets "Robot" fournissent des opérations abstraites pour manipuler les Robots : avancer(), tourner(), status(), etc.

# C'est quoi un type ?

- Définit :

- L'ensemble des valeurs possibles pour le type
- L'ensemble des opérations possibles pour le type

- Exemples :

- `int` : valeur comprise entre  $-2^{32}$  et  $+2^{32}$ , `[+, -, *, %, /, ...]` sont les opérations possibles
- `int[]` : taille comprise entre 0 et  $+2^{32}$ , `[.length]` est une opération possible
- `String` : `([aA-zZ]|[0-9]*)*`, `[.size(), +, ...]` comme opérations possibles

# Les Classes : les types des objets

- Les classes définissent les types des objets
- Définition des **méthodes** et des traitements
  - Le **comportement** des objets
- Définition des données nécessaires à la réalisation des traitements
  - L'**état** des objets
- Le **comportement** agit sur l'**état** et l'**état** influence le **comportement**

# Class Robot : définition

```
1 public class Robot {  
2  
3     // ...  
4  
5 }
```

- Définition d'une nouvelle classe **Robot**
- Implémentation de la classe robot entre { } : données et comportement
- Une classe Java par fichier Java : ici la classe Robot est dans le fichier **Robot.java**

# Class Robot : attributs

```
1 // ...
2
3 private int x;
4 private int y;
5 private int currentBattery;
6 private Direction currentDirection;
7
8 // ...
```

- Définition des données composant l'état d'un robot : les attributs
- **private** définit la "visibilité" de l'attribut

# Class Robot : visibilité

- Visibilité : définition de la permission d'accès
- **sans visibilité** : accessible par les objets du package
  - Aussi appelé "Package private"
- **public** : accessible par tous les objets
- **private** : accessible par les objets du type
- **protected** : accessible par les objets du package et "les classes filles"
- Bonne pratique : attributs toujours en privé

# Class Robot : constructeurs

```
1 // ...
2
3 public Robot() {
4     this.x = 0;
5     this.y = 0;
6     this.currentDirection = Direction.NORTH;
7     this.currentBattery = CAP_BATTERY;
8 }
9
10 public Robot(int x, int y, Direction direction) {
11     this.x = x;
12     this.y = y;
13     this.direction = direction;
14     this.currentBattery = CAP_BATTERY;
15 }
16
17 // ...
```

▪ Porte le même nom que la classe

▪ Visibilité

▪  $\emptyset$  de type de retour (même pas void)

▪ Avec ou sans paramètre

# Class Robot : constructions

```
1 public static void main(String[] args) {  
2     Robot r2d2 = new Robot();  
3     Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
4     // ...  
}
```

- Variable de type Robot nommée **r2d2**
- Variable de type Robot nommée **c3p0**
- “**Instanciation**” d’un nouveau robot sans paramètre
- “**Instanciation**” d’un nouveau robot avec paramètres

# Class Robot : méthode `forward()`

```
1 // ...
2 public void forward() {
3     if (this.canForward()) {
4         switch this.dirRobot {
5             case Direction.NORTH:
6                 y--;
7                 break;
8             case Direction.EAST:
9                 x++;
10                break;
11            // ...
12        }
13        currentBattery -= consForward
14    }
15 }
16 public boolean canForward() {
17     return currentBattery >= consForward;
18 }
```

■ Nom de la méthode

■ Visibilité

■ Body (lignes 3 à 14 pour `forward()`)

■ Type de retour

■ Avec ou sans paramètre

# Class Robot : méthode forward()

```
1 // ...
2 public void forward() {
3     if (this.canForward()) {
4         switch (this.dirRobot) {
5             case Direction.NORTH:
6                 y--;
7                 break;
8             case Direction.EAST:
9                 x++;
10                break;
11            // ...
12        }
13        currentBattery -= consForward;
14    }
15 }
16 public boolean canForward() {
17     return currentBattery >= consForward;
18 }
```

Usage des attributs de l'objet -> les valeurs des attributs influent sur le comportement de l'objet

# Class Robot : méthode `forward()`

```
1 // ...
2 public void forward() {
3     if (this.canForward()) {
4         switch this.dirRobot {
5             case Direction.NORTH:
6                 y--;
7                 break;
8             case Direction.EAST:
9                 x++;
10                break;
11            // ...
12        }
13        currentBattery -= consForward
14    }
15 }
16 public boolean canForward() {
17     return currentBattery >= consForward;
18 }
```

Appel de méthode depuis  
une autre méthode

# Class Robot : utilisation

```
1 Robot r2d2 = new Robot();  
2 while (r2d2.canForward()) {  
3     r2d2.forward();  
4 }
```

Appel de méthode sur un objet avec le '.'

# Class Robot : accès aux attributs privés

```
1 r2d2.currentDirection;
```

- Impossible ! Erreur de compilation

# Class Robot : accesseurs ou getters

```
1 public Direction getCurrentDirection() {  
2     return this.currentDirection;  
3 }
```

Retourne l'attribut

Nom de la méthode :  
**get** + nom de l'attribut

# Class Robot : mutateurs ou setters

```
1 public void setCurrentDirection(Direction currentDirection) {  
2     this.currentDirection = currentDirection;  
3 }
```

■ Nom de la méthode :  
**set** + nom de l'attribut

■ Mise à jour de l'attribut

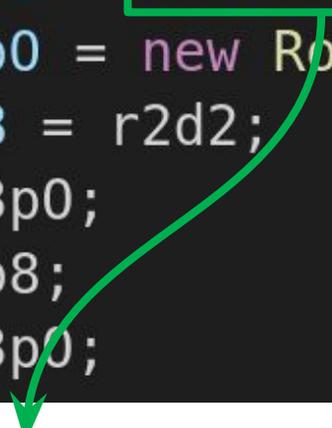
■ Paramètre :  
nouvelle valeur de  
l'attribut

# Concept Objet : références

```
1 Robot r2d2 = new Robot();  
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
3 Robot bb8 = r2d2;  
4 r2d2 = c3p0;  
5 c3p0 = bb8;  
6 r2d2 = c3p0;
```

# Concept Objet : références

```
1 Robot r2d2 = new Robot();  
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
3 Robot bb8 = r2d2;  
4 r2d2 = c3p0;  
5 c3p0 = bb8;  
6 r2d2 = c3p0;
```



Mémoire

x	0	
y	0	
currentBattery	100	
currentDirection	NORTH	

# Concept Objet : références

```
1 Robot r2d2 = new Robot();  
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
3 Robot bb8 = r2d2;  
4 r2d2 = c3p0;  
5 c3p0 = bb8;  
6 r2d2 = c3p0;
```



## Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

# Concept Objet : références

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

r2d2

c3p0

# Concept Objet : références

```
1 Robot r2d2 = new Robot();  
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
3 Robot bb8 = r2d2  
4 r2d2 = c3p0;  
5 c3p0 = bb8;  
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

r2d2

c3p0

# Concept Objet : références

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

r2d2, bb8

c3p0

# Concept Objet : références

```
1 Robot r2d2 = new Robot();  
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);  
3 Robot bb8 = r2d2;  
4 r2d2 = c3p0;  
5 c3p0 = bb8;  
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

bb8

c3p0, r2d2

# Concept Objet : références

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

bb8, c3p0

r2d2

# Concept Objet : références

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
```

Mémoire

x	0	x	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH

bb8, c3p0, r2d2

# Concept Objet : références

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
```

Mémoire

x	0	*	<del>23</del>
y	0	y	<del>32</del>
currentBattery	100	currentBattery	<del>100</del>
currentDirection	NORTH	currentDirection	<del>SOUTH</del>

bb8, c3p0, r2d2

# Questions !

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = r2d2;
4 r2d2 = c3p0;
5 c3p0 = bb8;
6 r2d2 = c3p0;
7 r2d2.setCurrentDirection(Direction.SOUTH);
8 System.out.println(c3p0.getCurrentDirection());
9 System.out.println(bb8.getCurrentDirection());
```

# Concept Objet : références null

```
1 c3p0 = null;  
2 r2d2 = null;  
3 bb8 = null;
```

Mémoire

x	0	*	<del>23</del>
y	0	y	<del>32</del>
currentBattery	100	currentBattery	<del>100</del>
currentDirection	NORTH	currentDirection	<del>SOUTH</del>

bb8, r2d2

# Concept Objet : références null

```
1 c3p0 = null;  
2 r2d2 = null;  
3 bb8 = null;
```

Mémoire

x	0	*	<del>23</del>
y	0	y	<del>32</del>
currentBattery	100	currentBattery	<del>100</del>
currentDirection	NORTH	currentDirection	<del>SOUTH</del>

bb8

# Concept Objet : références null

```
1 c3p0 = null;  
2 r2d2 = null;  
3 bb8 = null;
```

Mémoire

*	0	*	23
y	0	y	32
currentBattery	100	currentBattery	100
currentDirection	NORTH	currentDirection	SOUTH



# Concept Objet : appel de méthode sur référence null

```
1 bb8 = null;
2 while (bb8.canMoveForward()) {
3     bb8.forward();
4 }
```

■ Appel de méthode sur une variable null =>  
Erreur:

**NullPointerException!**

# Programmation objet : vocabulaire

- Une **classe** est un type d'**objet**
- Un **objet** ou une **instance** est une “occurrence” d'une **classe**
- Les **attributs** sont les données d'un **objet**, définis par la **classe**
- Les **méthodes** implémentent le comportement des objets
- Les **membres** d'une classe sont ses attributs et ses méthodes
- Une **variable** objet est une référence vers un **objet**

# Le mot-clé **this**

```
1 private Direction currentDirection;  
2  
3 public void setCurrentDirection(Direction currentDirection) {  
4     this.currentDirection = currentDirection;  
5 }
```

- Préfixer le nom avec **this** pour lever l'ambiguïté et spécifier que l'on veut l'attribut

- Attribut et Paramètre ont le même nom :  
“**currentDirection**”

# Méthodes standards des objets

- Les méthodes standards :
  - **boolean equals(Object o)** : permet de tester si deux objets égaux
  - **String toString()** : permet d'afficher sur la sortie standard l'objet

# Méthode standard : equals()

```
1 public boolean equals(Object that) {
2     if (that == null || !that instanceof Robot) {
3         return false;
4     }
5     Robot thatRobot = (Robot)that;
6     return this.x == thatRobot.x &&
7         this.y == thatRobot.y &&
8         this.currentDirection == thatRobot.currentDirection &&
9         this.currentBattery == thatRobot.currentBattery;
10 }
```

- Renvoie true quand les objets sont égaux
- Par défaut, Java compare les adresses mémoire
- En implémentant la méthode equals(Object that), on spécifie comment comparer deux objets
- Classiquement, on compare tous les champs

# Méthode standard : equals()

```
1 public boolean equals(Object that) {  
2     if (that == null || !that instanceof Robot) {  
3         return false;  
4     }  
5     Robot thatRobot = (Robot)that;  
6     return this.x == thatRobot.x &&  
7         this.y == thatRobot.y &&  
8         this.currentDirection == thatRobot.currentDirection &&  
9         this.currentBattery == thatRobot.currentBattery;  
10 }
```

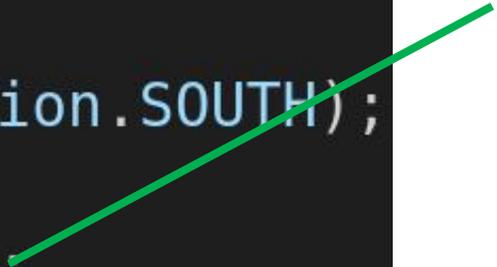
- Si le paramètre est null, ou n'est pas un Robot
  - Alors les objets ne sont pas égaux
- Sinon, tous les attributs doivent être égaux pour que les objets soient égaux

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```



# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

■ vrai

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

■ vrai

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

■ vrai

■ vrai

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

■ vrai

■ vrai

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux  
■ vrai  
■ vrai  
■ faux

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux

■ vrai

■ vrai

■ faux

# Méthode standard : equals() usage

```
1 Robot r2d2 = new Robot();
2 Robot c3p0 = new Robot(23, 32, Direction.SOUTH);
3 Robot bb8 = new Robot();
4 System.out.println(r2d2.equals(c3p0));
5 System.out.println(r2d2.equals(r2d2));
6 System.out.println(r2d2.equals(bb8));
7 System.out.println(r2d2.equals(null));
8 System.out.println(r2d2.equals(""));
```

■ faux  
■ vrai  
■ vrai  
■ faux  
■ faux

# Méthode standard : toString()

```
1 public String toString() {  
2     return "(" + this.x + ";" + this.y +  
3         "): " + this.currentDirection +  
4         " [" + this.currentBattery + "]"  
5 }
```

- Formate et transforme les attributs en chaîne de caractères

# Méthode standard : `toString()` usage

```
1 Robot r2d2 = new Robot();  
2 String r2d2AsString = r2d2.toString();  
3 System.out.println(r2d2AsString);  
4 System.out.println(r2d2);
```

■ Affiche l'objet sous forme de chaîne

■ Appel à `toString()` implicite et géré par Java

■ Output:

- L3 ?
- L4 ?

# Méthode standard : `toString()` usage

```
1 Robot r2d2 = new Robot();  
2 String r2d2AsString = r2d2.toString();  
3 System.out.println(r2d2AsString);  
4 System.out.println(r2d2);
```

■ Affiche l'objet sous forme de chaîne

■ Appel à `toString()` implicite et géré par Java

- Output:

“(0;0) : NORTH [100]”

“(0;0) : NORTH [100]”

# Le mot-clé `static`

```
1 private static final int CAP_BATTERY = 100;
2 public static int generateRandomBattery() {
3     return new Random().nextInt(CAP_BATTERY);
4 }
5 public Robot(boolean random) {
6     // ...
7     if (!random) {
8         this.currentBattery = CAP_BATTERY;
9     } else {
10        this.currentBattery = generateRandomBattery();
11    }
12 }
13 public static void main(String[] args) {
14     new Robot(23, 32, Direction.SOUTH,
15         Robot.generateRandomBattery());
16 }
```

- Déclaration de constante avec `static`
- Déclaration de méthode de classe
- Usage de la constante et de la méthode `static` dans une instance
- Usage hors de l'instance

# Interfaces et Implémentations

# Problématique

- On a des objets de la vie de tous les jours :
  - **Papier, Bouteille, Pile, etc.**
- Ces objets ont des comportements différents :
  - **déchirer le Papier**
  - **écraser la Bouteille**
  - **etc...**
- **Mais !** Tous ces objets sont recyclables :
  - **recycler le Papier**
  - **recycler la Bouteille,**
  - **etc...**

# Problématique : en classe Java

```
1 public class Paper {
2     public void tear() { /* ... */ }
3     public void recycling() {
4         System.out.println("Recycling Paper");
5     }
6 }
7 public class Bottle {
8     public void crush() { /* ... */ }
9     public void recycling() {
10        System.out.println("Recycling Bottle");
11    }
12 }
```

# Problématique : ce que l'on souhaite faire

- On souhaite par exemple, pouvoir programmer le recyclage de l'ensemble des objets contenus dans une “poubelle”
- Cette poubelle est représentée par un tableau d'objet
  - **Papier, Bouteille, Pile, etc.**
- En Java :

```
1  for (int i = 0 ; i < trash.length ; i++) {  
2      trash[i].recycling();  
3  }
```

# Problématique

- En Java :

```
1 for (int i = 0 ; i < trash.length ; i++) {  
2     trash[i].recycling();  
3 }
```

- Comment coder cet ensemble d'objets de type **T**, qui permet d'écrire :

```
1 T[] trash = new T[/** ... **/];
```

# Comment faire ? Il faut définir T

```
1 T[] trash = new T[2];
2 trash[0] = new Paper();
3 trash[1] = new Bottle();
4 for (int i = 0 ; i < trash.length ; i++) {
5     trash[i].recycling();
6 }
```

- Le Type **T** doit pouvoir :
  - Recevoir des références d'objet de type **Paper** et de type **Bottle**
  - Doit permettre l'appel de la méthode recycling

# Solutions :

- Une nouvelle classe **Unique** pour représenter les **Paper** et les **Bottle** ?

```
1 Unique paper = new Unique();  
2 Unique bottle = new Unique();
```

- Problème : Comment distinguer les objets **Paper** des objets **Bottle** ?
  - gestion des méthodes spécifiques (tear() / crush()) de chacun ?
  - **Impossible**

# Solutions : l'Interface objet

- Interface : déclare la signature de méthodes publiques
  - $\emptyset$  d'attribut,  $\emptyset$  de body,  $\emptyset$  de méthodes privées
  - **Impossible** d'instancier une Interface !
  
- Une classe **implémente** une interface :
  - Définition du corps des méthodes déclarées dans l'interface
  - **Polymorphisme**

# Interface Recyclable

```
1 public interface Recyclable {
2     public void recycling();
3 }
4 public class Paper implements Recyclable {
5     public void tear() { /* ... */ }
6     public void recycling() {
7         System.out.println("Recycling Paper");
8     }
9 }
10 public class Bottle implements Recyclable {
11     public void crush() { /* ... */ }
12     public void recycling() {
13         System.out.println("Recycling Bottle");
14     }
15 }
```

■ Déclaration de l'interface **Recyclable**:

■ signature de la méthode **recycling**

■ Implémentation dans la classe Paper:

■ Implémentation de **recycling**

■ Implémentation dans la class Bottle:

■ Implémentation de recycling

# Interfaces : usage

```
1 public static void aMethod(Recyclable r) {
2     /** ... */
3 }
4 // ...
5 Recyclable r1 = new Paper();
6 aMethod(r1);
7 aMethod(new Bottle());
8 Paper p = new Paper();
9 aMethod(p);
10 // ...
```

Interface en type de paramètre

Instanciation d'un nouvel objet. La variable est du type de l'interface

Appels de la méthode qui prend le **Recyclable** en paramètre

La méthode peut accepter un objet **Paper**, car Paper est aussi Recyclable

# Interfaces : usage

```
1 public static void aMethod(Recyclable r) {
2     /** ... */
3 }
4 // ...
5 Recyclable r1 = new Paper();
6 aMethod(r1);
7 aMethod(new Bottle());
8 Paper p = new Paper();
9 aMethod(p);
10 // ...
```

Interface en type de paramètre

Instanciation d'un nouvel objet. La variable est du type de l'interface

Appels de la méthode qui prend le **Recyclable** en paramètre

La méthode peut accepter un objet **Paper**, car Paper est aussi Recyclable

Bonne pratique : toujours se baser sur les types abstraits :

~~Paper p = new Paper();~~

Recyclable p = new Paper();

# Multiple implémentation d'Interfaces

```
1 public class Paper implements Recyclable, Burnable {
2     public void tear() { /* ... */ }
3     public void recycling() {
4         System.out.println("Recycling Paper");
5     }
6     public void burning() {
7         System.out.println("Burning Paper");
8     }
9 }
```

# La force du polymorphisme

```
1 Recyclable[] trash = new Recyclable[2];
2 trash[0] = new Paper();
3 trash[1] = new Bottle();
4 for (int i = 0 ; i < trash.length ; i++) {
5     trash[i].recycling();
6 }
```

# Limite du polymorphisme

```
1 Recyclable[] trash = new Recyclable[2];
2 Paper p = new Paper();
3 trash[0] = p;
4 p.recycling();
5 p.tear();
6 trash[0].recycling();
7 trash[0].tear(); // !!!
```

■ Appel des méthodes `recycling` et `tear` depuis une variable de référence `Paper`

■ Appel des méthodes `recycling` depuis une variable de référence `Recyclable`, mais **impossible** d'appeler `tear()`

# Mais comment ça marche ?

```
1 Recyclable[] trash = new Recyclable[2];
2 trash[0] = new Paper();
3 trash[1] = new Bottle();
4 for (int i = 0 ; i < trash.length ; i++) {
5     trash[i].recycling();
6 }
```

- Output:

“recycling paper”

“recycling bottle”

# Late-Binding

- Vérification à la compilation que l'appel de la méthode est autorisé
  - ↳ Dépend uniquement du type de l'appelant
  
- Résolution à l'exécution de l'implémentation de la méthode à exécuter
  - ↳ Dépend de l'instance sur laquelle on appelle la méthode

# Late-Binding : exemple

```
1 public static void doRecycling(Recyclable r) {  
2     r.recycling();  
3 }
```

■ Appel de **recycling** sur le paramètre

■ Paramètre : **Recyclable**

# Late-Binding : exemple

```
1 public static void doRecycling(Recyclable r) {
2     r.recycling();
3 }
4 public static void main(String args[]) {
5     Recyclable recyclable;
6     if ("paper".equals(args[0])) {
7         recyclable = new Paper();
8     } else {
9         recyclable = new Bottle();
10    }
11    doRecycling(recyclable);
12 }
```

■ Test des paramètres d'entrée

■ Instanciation d'objets de différents types

■ Appel de la méthode doRecycling sur le **Recyclable** : soit **Paper** soit **Bottle**

# Late-Binding : exemple

```
1 $ java RecyclingMain paper
2 > recycling paper
3 $ java RecyclingMain bottle
4 > recycling bottle
```

# Généricité

# Motivation : objets pair

- On veut pouvoir manipuler des pairs :
  - ↳ Des objets qui ont deux types différents
  
- Exemple : des objets Pair d'un String et d'un entier

# Motivation : objets pair d'un string et d'un entier

```
1 public class PairStringInteger {
2
3     private String val1;
4     private Integer val2;
5
6     public PairStringInteger(String val1, Integer val2) {
7         this.val1 = val1;
8         this.val2 = val2;
9     }
10
11     public String getFirst() { return this.val1; }
12     public Integer getSecond() { return this.val2; }
13
14     public void setFirst(String val1) { this.val1 = val1; }
15     public void setSecond(Integer val2) { this.val2 = val2; }
16
17     public String toString() { return this.val1 + ";" + this.val2; }
18 }
```

Class PairStringInteger

Valeurs String/Integer

Constructeur

Getters

Setters

toString()

# Motivation : pair de string et integer usage

```
1 public static void main(String [] args) {
2     PairStringInteger p1 = new PairStringInteger("La réponse", 42);
3     PairStringInteger p2 = new PairStringInteger("Le nombre", 23);
4     System.out.println(p1);
5     System.out.println(p2);
6     p2.setFirst("23 à l'envers");
7     p2.setSecond(32);
8     System.out.println(p2);
9 }
```

- Output :

“La réponse;42”

“Le nombre;23”

“23 à l'envers;32”

# Motivation : objet pair string robot

- On veut maintenant pouvoir manipuler des paires de String et de Robot:
  - ↳ Il faut implémenter de nouveau une classe : **PairStringRobot**

# Motivation : objet pair string robot

- On veut maintenant pouvoir manipuler des paires de String et de Robot:
  - ↳ Il faut implémenter de nouveau une classe : **PairStringRobot**

```
1 public class PairStringRobot {
2
3     private String val1;
4     private Robot val2;
5
6     public PairStringRobot(String val1, Robot val2) {
7         this.val1 = val1;
8         this.val2 = val2;
9     }
10
11     public String getFirst() { return this.val1; }
12     public Robot getSecond() { return this.val2; }
13
14     public void setFirst(String val1) { this.val1 = val1; }
15     public void setSecond(Robot val2) { this.val2 = val2; }
16
17     public String toString() { return this.val1 + ";" + this.val2.toString(); }
18 }
```

# Motivation : pair de string et robot usage

```
1 public static void main(String [] args) {  
2     Robot r2d2 = new Robot();  
3     PairStringRobot p1 = new PairStringRobot("r2d2", r2d2);  
4     System.out.println(p1);  
5     p1.setFirst("bb8");  
6     System.out.println(p1);  
7 }
```

- Output :  
    “r2d2; (0;0) : NORTH [100]”  
    “bb8; (0;0) : NORTH [100]”

# Motivation : conclusion

- Le comportement des objets **Pair** ne dépend pas des types contenus dans la paire
- Solution :
  - Paramétrer le type **Pair** avec le type des valeurs manipulées
  - Programmation d'une seule classe **générique Pair**
  - Définition des types par l'utilisateur de la classe **Pair**
- La classe **Pair** n'est codée qu'une seule fois et fonctionne pour tous les types de paramètres possibles

# Le type Pair<T1, T2> générique

```
1 public class Pair<T1, T2> {  
2  
3     private T1 val1;  
4     private T2 val2;  
5  
6     public Pair(T1 val1, T2 val2) {  
7         this.val1 = val1;  
8         this.val2 = val2;  
9     }  
10  
11     public T1 getFirst() { return this.val1; }  
12     public T2 getSecond() { return this.val2; }  
13  
14     public void setFirst(T1 val1) { this.val1 = val1; }  
15     public void setSecond(T2 val2) { this.val2 = val2; }  
16  
17     public String toString() {  
18         return this.val1.toString() + ";" + this.val2.toString();  
19     }  
20 }
```

■ Class Pair

■ Paramétriser avec deux types T1 et T2

■ Références au types T1 et T2 :

■ Getters

■ Setters

■ toString()

# Le type `Pair<T1, T2>` générique : usage

```
1 public static void main(String [] args) {
2     Pair<String, Integer> p1 = new Pair<>("La réponse", 42);
3     Robot r2d2 = new Robot();
4     Pair<String, Robot> p2 = new Pair<>("r2d2", r2d2);
5     System.out.println(p1);
6     p2.setFirst("bb8");
7     System.out.println(p2);
8     Pair<char, int> p3 = new Pair<>('a', 0);
9 }
```

■ Instanciation d'un objet `Pair<String, Integer>`

■ Instanciation d'un objet `Pair<String, Robot>`

■ “sucre syntaxique” :

`<>`  
raccourcis pour  
`<String, Robot>`

Il est **incorrect** d'utiliser des types “**primitifs**”. Pour les génériques, il faut absolument utiliser des **types d'objet**

~~`Pair<char, int> p3 = new Pair<>('a', 0);`~~

■ Output :

“La réponse;42”

“bb8; (0;0) : NORTH [100]”

# L'interface Comparable<T>

```
1 public interface Comparable<T> {  
2     /**  
3     * > 0 si this > that  
4     * < 0 si this < that  
5     * = 0 si this == that  
6     */  
7     public int compareTo(T that);  
8 }
```

Interface **Comparable**

Paramétré par un type **T**

**compareTo** permet de définir un ordre des objets

# Implémentation de Comparable<Robot>

```
1 public class Robot implements Comparable<Robot> {  
2     // ...  
3     public int compareTo(Robot that) {  
4         return this.currentEnergy - that.currentEnergy;  
5     }  
6     // ...  
7 }
```

Interface **Comparable**

Paramétré avec le type  
**Robot**

On ordonne les robots en fonction de leur  
niveau courant d'énergie

# Structures de données

# Plan

- Tableau
- Les **Collections** standards Java
- Itérable & Iterator
- Liste et Liste chaînée
- Set
- Map
- File
- Pile
- Arbre

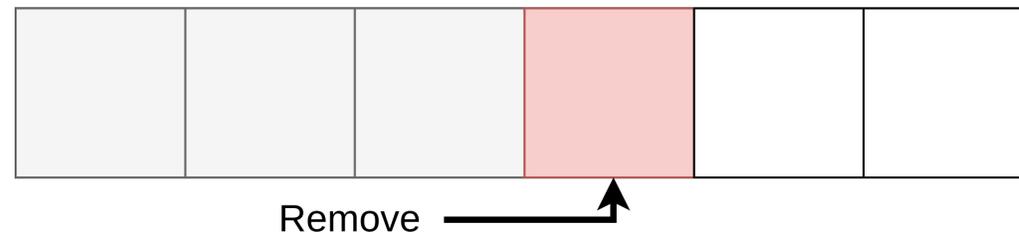
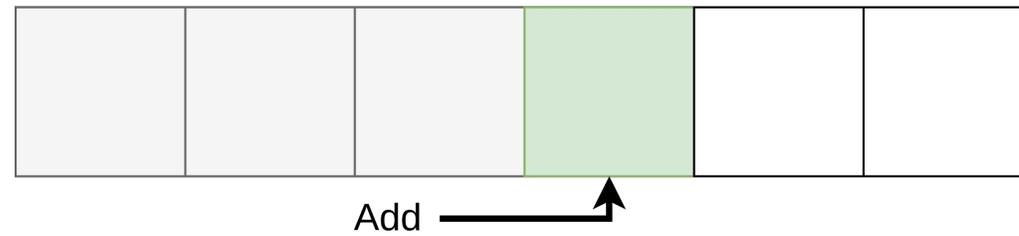
# Introduction

- Organisation des données
- Automatisation de manipulation
- Performance différente
- Impact sur le développement

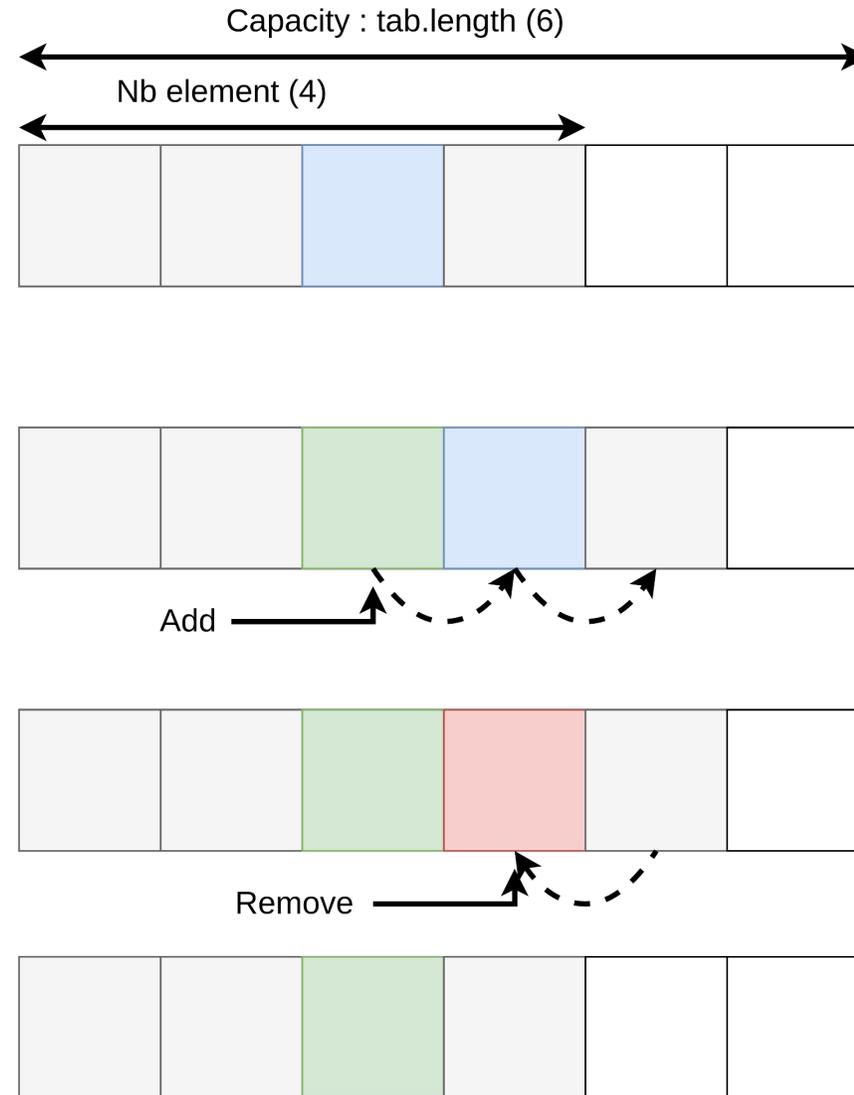
# Tableau

- Fourni par le langage
- Efficace pour consulter / modifier des éléments
- Peu de mémoire consommée

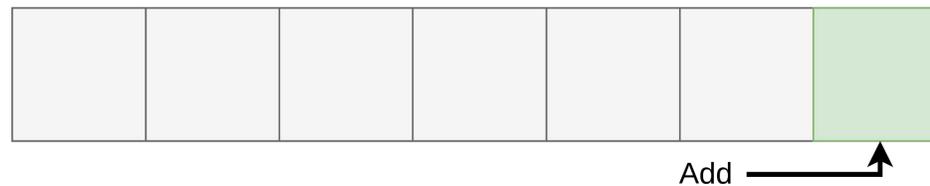
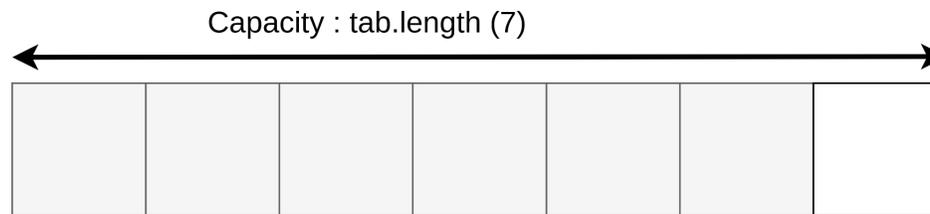
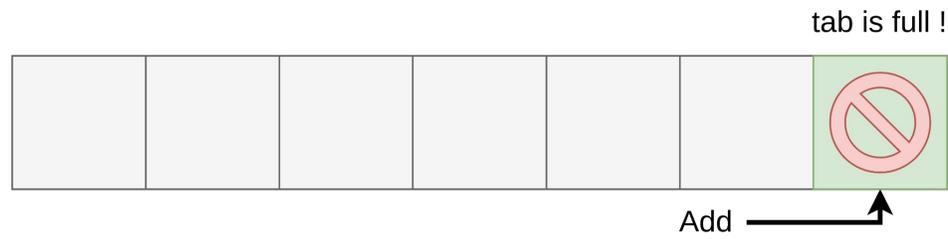
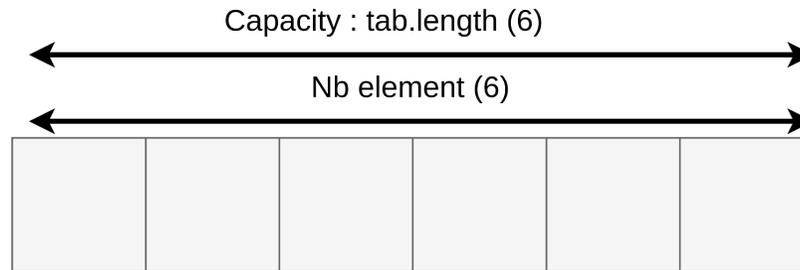
# Tableau : ajout/retrait en fin de tableau



# Tableau : ajout/retrait à un indice donné



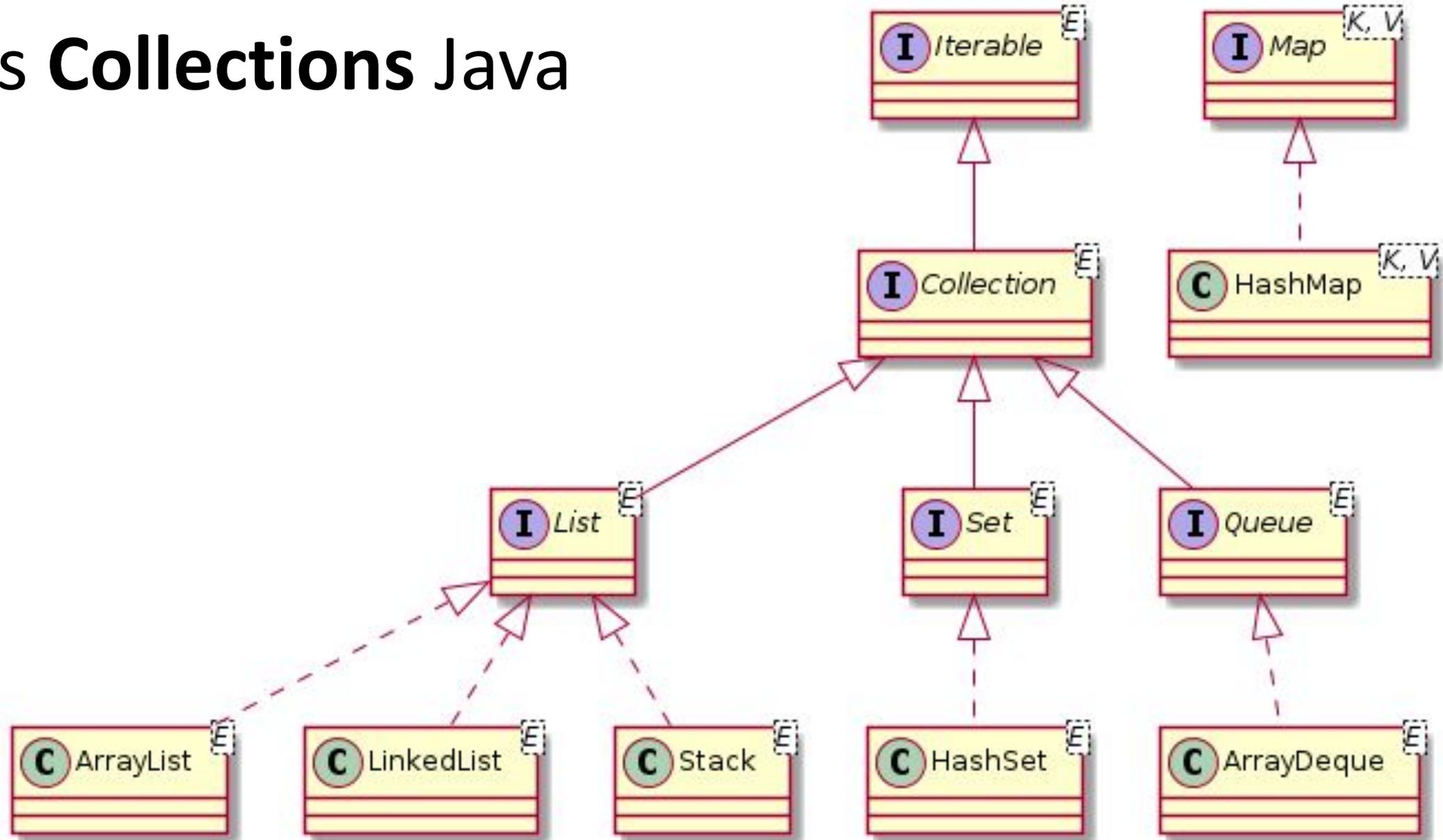
# Tableau : ajout d'élément dans tableau plein



# Tableau : conclusion

- Les plus :
  - Simple et rapide à utiliser
  - Bonne performance pour l'accès et la modification
- Les moins :
  - Mauvaise performance pour l'ajout ou la suppression
  - Taille fixe
- → Besoin d'une structure de données plus souple

# Les Collections Java



# Les Collections Java

```
1 public interface Collection<E>
2     extends Iterable<E> {
3
4     boolean add(E element);
5
6     boolean contains(E element);
7
8     boolean isEmpty();
9
10    Iterator<E> iterator();
11
12    boolean remove(E element);
13
14    int size();
15    // ...
16 }
```

Ajoute l'élément à la collection

Retourne **vrai** si l'élément a bien été ajouté, **faux** sinon

Retourne **vrai** si la Collection contient l'élément, **faux** sinon

Retourne **vrai** si la Collection ne contient aucun élément, **faux** sinon

Retourne un itérateur sur les éléments de la collection

Retire l'élément de la collection  
Retourne vrai si l'élément a bien été ajouté, faux sinon

Retourne le nombre d'éléments dans la collection

# Iterable et Iterator

```
1 public interface Iterator<E> {
2
3     boolean hasNext();
4
5     E next();
6     // ..
7 }
8 // ..
9 // a collection of Robots
10 Collection<Robot> robots = // ..
11 Iterator<Robot> it = robots.iterator()
12 while (it.hasNext()) {
13     Robot current = it.next();
14     System.out.println(current);
15 }
```

- Retourne vrai s'il y encore un élément
- Retourne le prochain élément et avance le curseur

Construction d'un iterator

- Tant qu'il y a un élément suivant
- Récupération du prochain élément

# For-Each Java

```
1 // a collection of Robots
2 Collection<Robot> robots = // ..
3 Iterator<Robot> it = robots.iterator();
4 while (it.hasNext()) {
5     Robot current = it.next();
6     System.out.println(current);
7 }
8 // equivalent code
9 Collection<Robot> robots = // ..
10 for (Robot current: robots) {
11     System.out.println(current);
12 }
13
```

■ Parcours tous les éléments de la Collection de la même façon

■ Variable contenant l'élément courant de l'itération de la boucle

■ Collection dont on veut parcourir les éléments

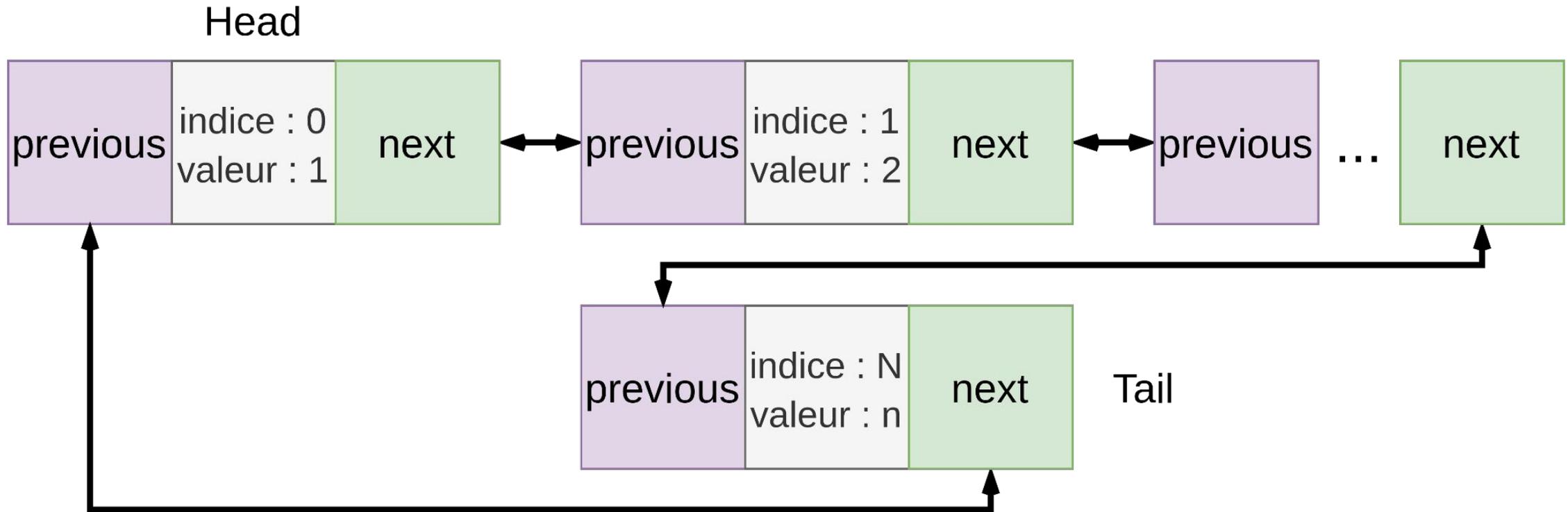
# List : ArrayList

- Basée sur un tableau
- Capacité dynamique
- Ajout/suppression en fin de List performant
- Ajout/suppression à un indice donné **PEU** performant
- Parcours possible
- Récupération d'élément à partir d'un indice

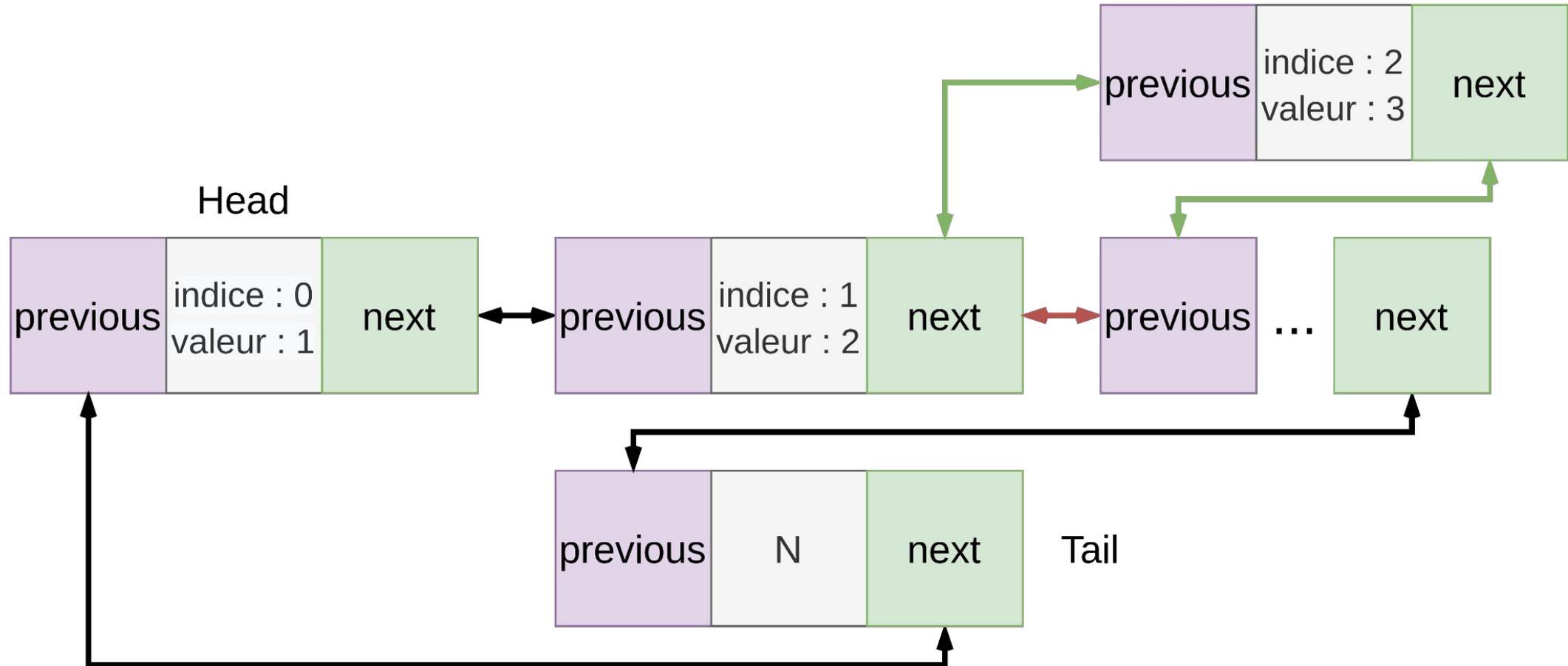
# List chaînée : LinkedList

- Basée sur des éléments liés les uns aux autres
- Chaque élément référence le précédent et le suivant
- Ajout/suppression sans décalage (mais nécessite un parcours)
- Parcours possible
- Récupération d'élément à partir d'un indice (moins performant)

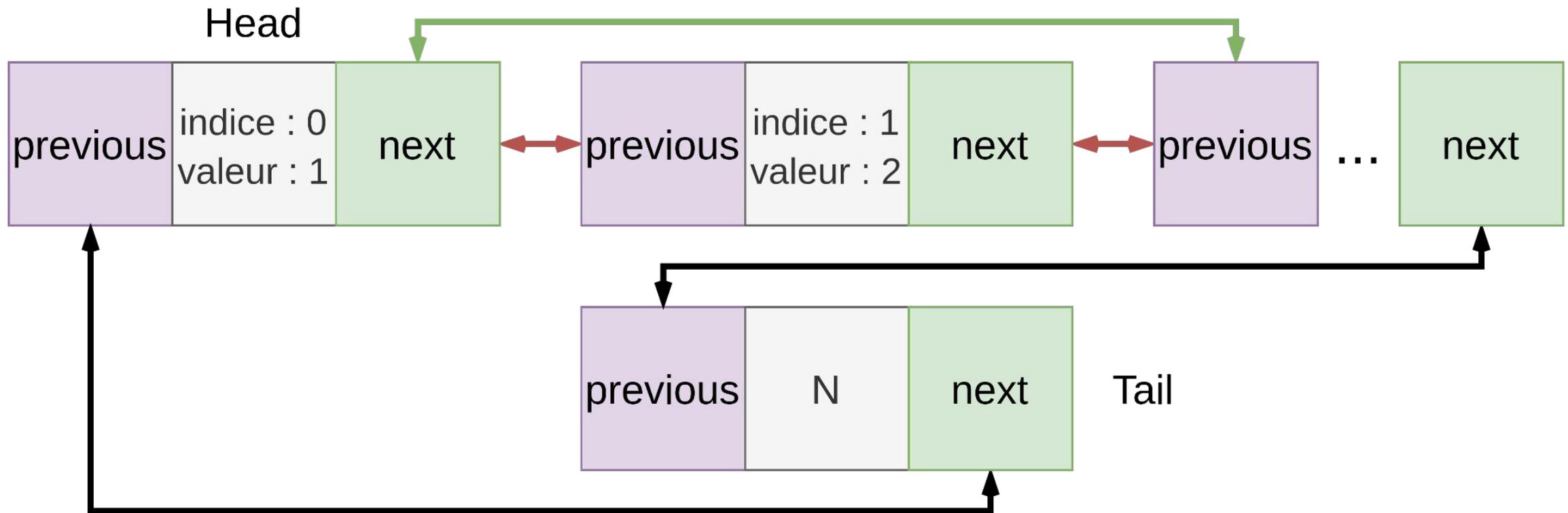
# List chaînée : LinkedList



# List chaînée : ajout



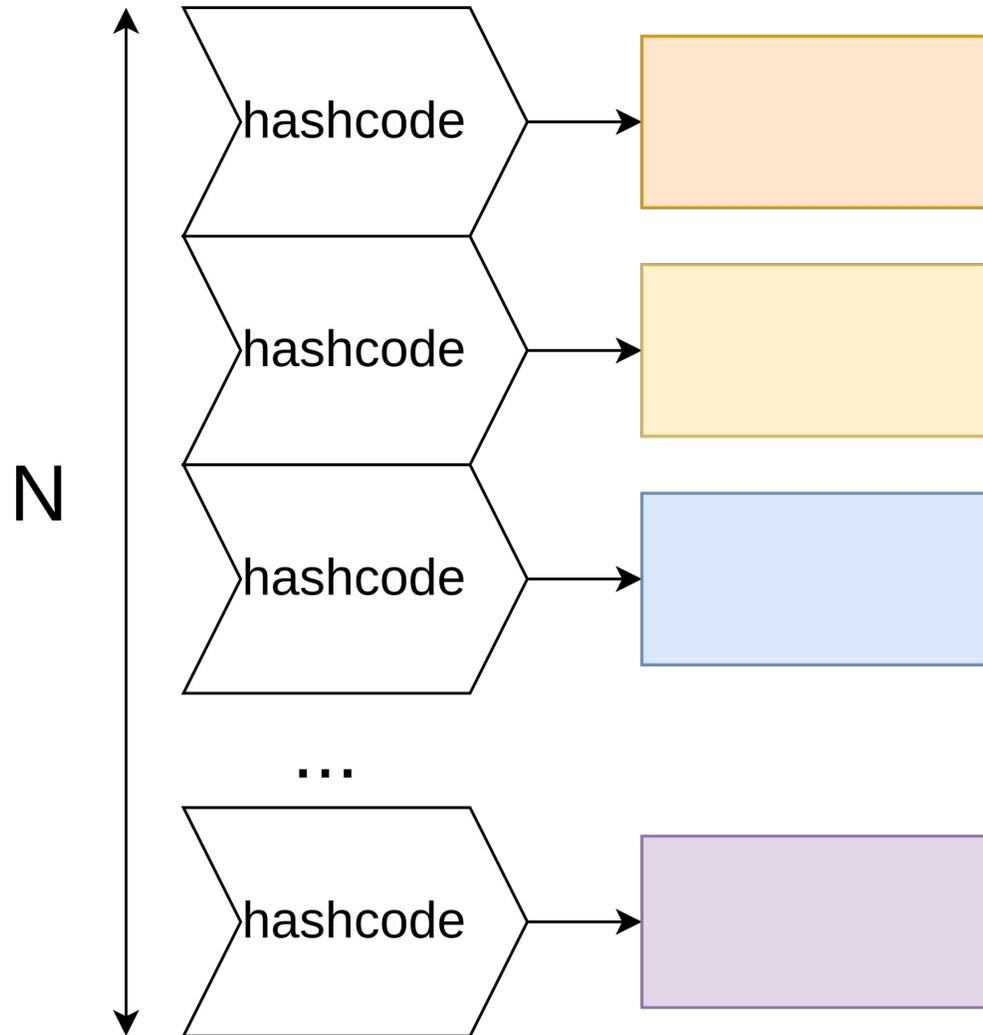
# List chaînée : suppression



# Ensemble : Set

- Ensemble au sens mathématique
- Ajout/suppression similaire au List avec `add()` et `remove()`
- Pas de notion d'indice : parcours uniquement avec un Itérateur
- Implémentation 1 : `HashSet<E>`
- Implémentation 2 : `TreeSet<E>`

# HashSet



- Basé sur une représentation numérique des objets : le hashcode
- Associe une instance à une valeur (le hashcode)
- Performant dans le cas où la fonction de hachage est bien choisie
- Pas d'ordre sur les éléments

# HashCode et fonction de hachage

- Hashcode : valeur numérique qui correspond à une instance particulière d'une classe
- Fonction de hachage :
  - rapide
  - une instance → un entier

```
1 // Robot.java
2 public int hashCode() {
3     int tmp = (this.y+((this.x+1)/2));
4     return this.x+(tmp*tmp);
5 }
```

# Bonne pratique : `hashCode()` et `equals()`

- Si on implémente **`hashCode()`** alors on implémente **`equals()`** (et vice-versa)
- **`hashCode()`** et **`equals()`** doivent reposer sur les même champs
- Si deux objets sont égaux (**`o1.equals(o2) == true`**) alors leurs hashcodes sont égaux : **`o1.hashCode() == o2.hashCode()`**

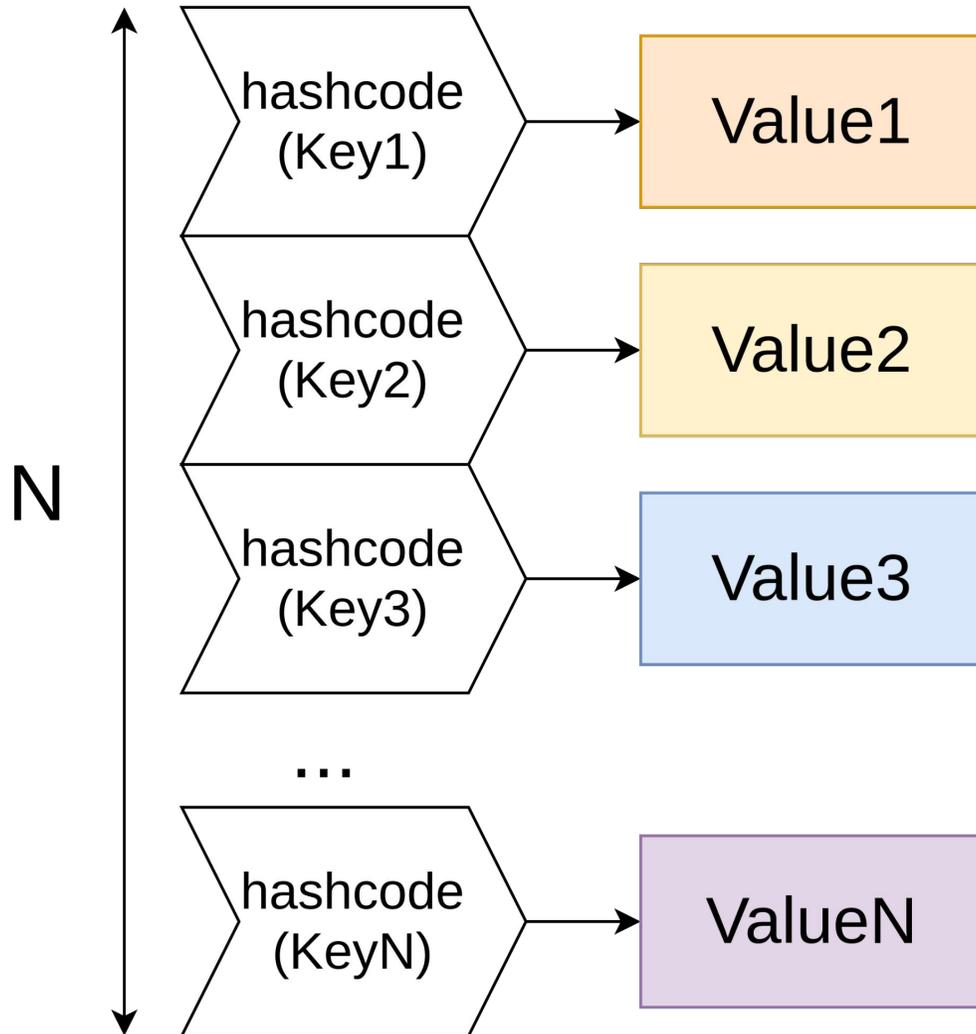
# TreeSet

- Utilise un “arbre” binaire
- La plupart des opérations ont le même coût
- Ordre imposé et nécessaire : les objets doivent implémenter l'interface **Comparable**

# Map

- Manipule des couples Clé - Valeur
- Aussi appelée : dictionnaire, table associative, index, etc.
- N'implémente pas Iterable
- Exemple d'implémentation : `HashMap<K, V>`

# Map : HashMap<K,V>



- Basée sur une représentation numérique des objets : le hashcode
- Associe le hashcode de la clé à la valeur

# Map : usage

```
1  V put(K key, V value);
2  V get(K key);
3  V remove(K key);
4  boolean containsKey(K key);
5  boolean containsValue(V value);
6  int size();
7  boolean isEmpty();
8  void clear();
9  Set<K> keySet();
10 Collection<V> values();
```

- Ajoute un couple clé - valeur  
Renvoie **null** si la clé n'est pas dans la map,  
Renvoie l'ancienne valeur sinon
- Récupère la valeur à partir d'une clé
- Retire une association à partir d'une clé
- Vérifie si la map contient la clé ou la valeur donnée
- Retourne le nombre d'associations dans la map
- Vérifie si la map est vide ou non
- Vide la map
- Renvoie l'ensemble des clés
- Renvoie toutes les valeurs

# Map : usage

```
1 Map<String, Robot> robotsPerNames = new HashMap<>();
2 Robot r2d2 = new Robot();
3 robotsPerNames.put("r2d2", r2d2);
4 robotsPerNames.get("r2d2").equals(r2d2); // true
5 robotsPerNames.containsKey("r2d2"); // true
6 robotsPerNames.containsKey("c3p0"); // false
7 Set<String> keys = robotsPerNames.keySet();
8 for (String key: keys) {
9     System.out.println(key + " " + robotsPerNames.get(key));
10 }
```

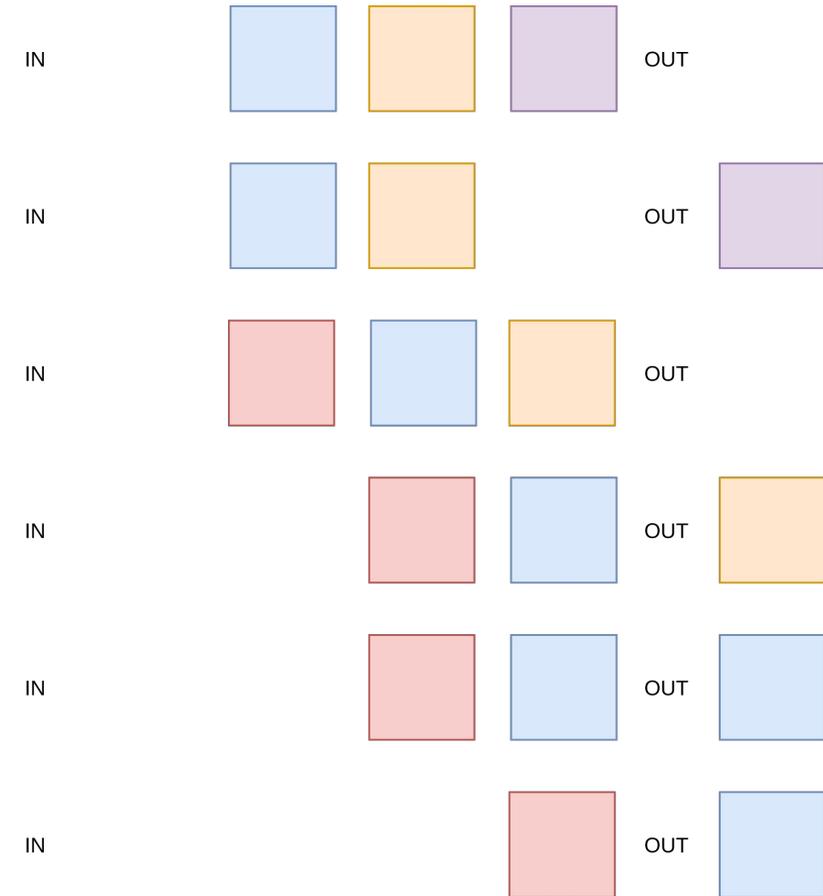
# File

- File ou Queue
- FIFO : First In First Out
- Capacité maximale
- Queue<E>

```
1 boolean add(E element);  
2 E peek();  
3 E poll();
```

# File : exemple

```
1 Queue<Color> colors = new ArrayDeque<Color>(4);
2 colors.add(Color.PURPLE);
3 colors.add(Color.ORANGE);
4 colors.add(Color.BLUE);
5 colors.poll(); // PURPLE
6 colors.add(Color.RED);
7 colors.poll(); // ORANGE
8 colors.peek(); // BLUE
9 colors.poll(); // BLUE
```



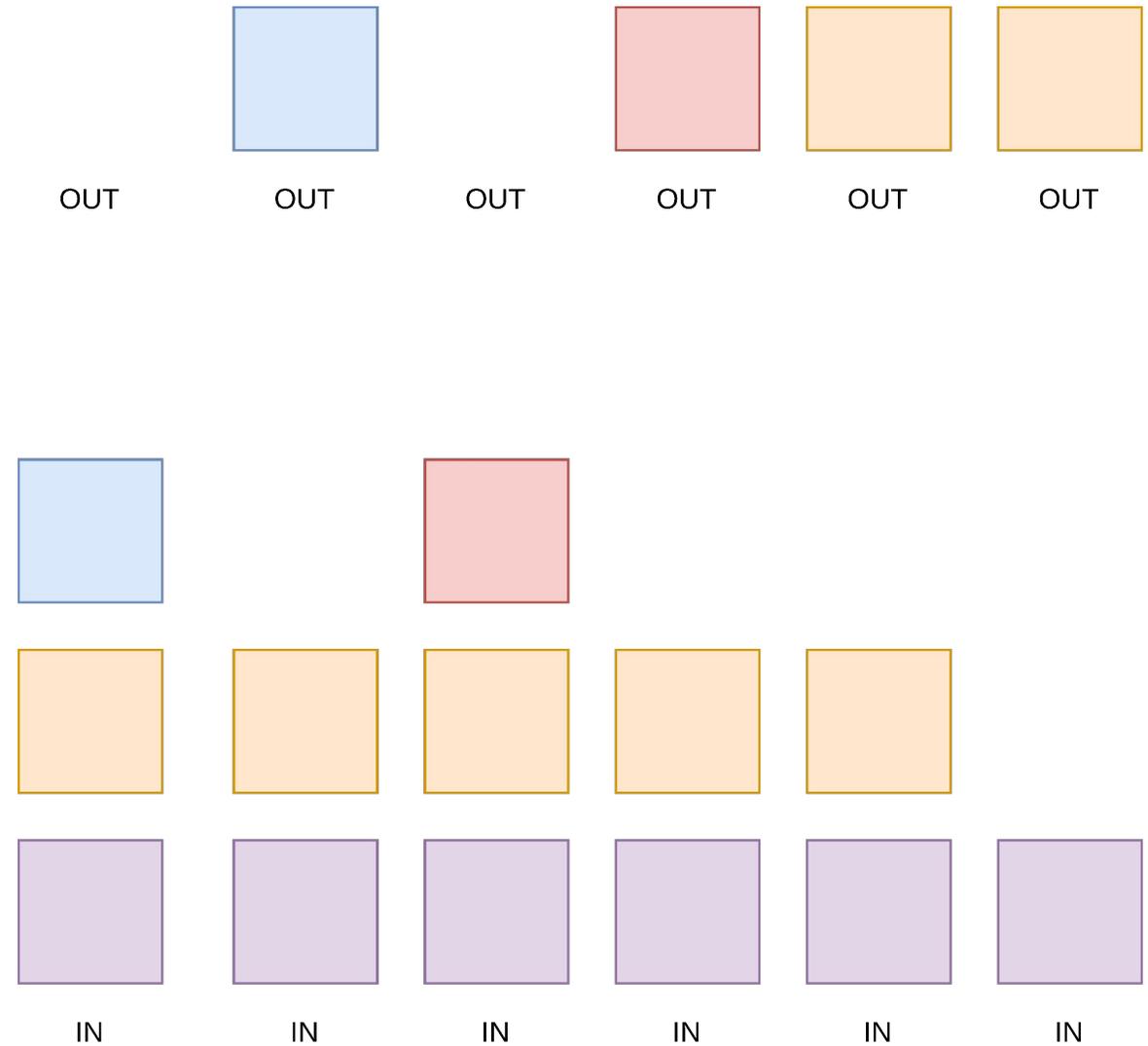
# Pile

- Pile ou Stack
- LIFO : Last In First Out
- Stack<E>

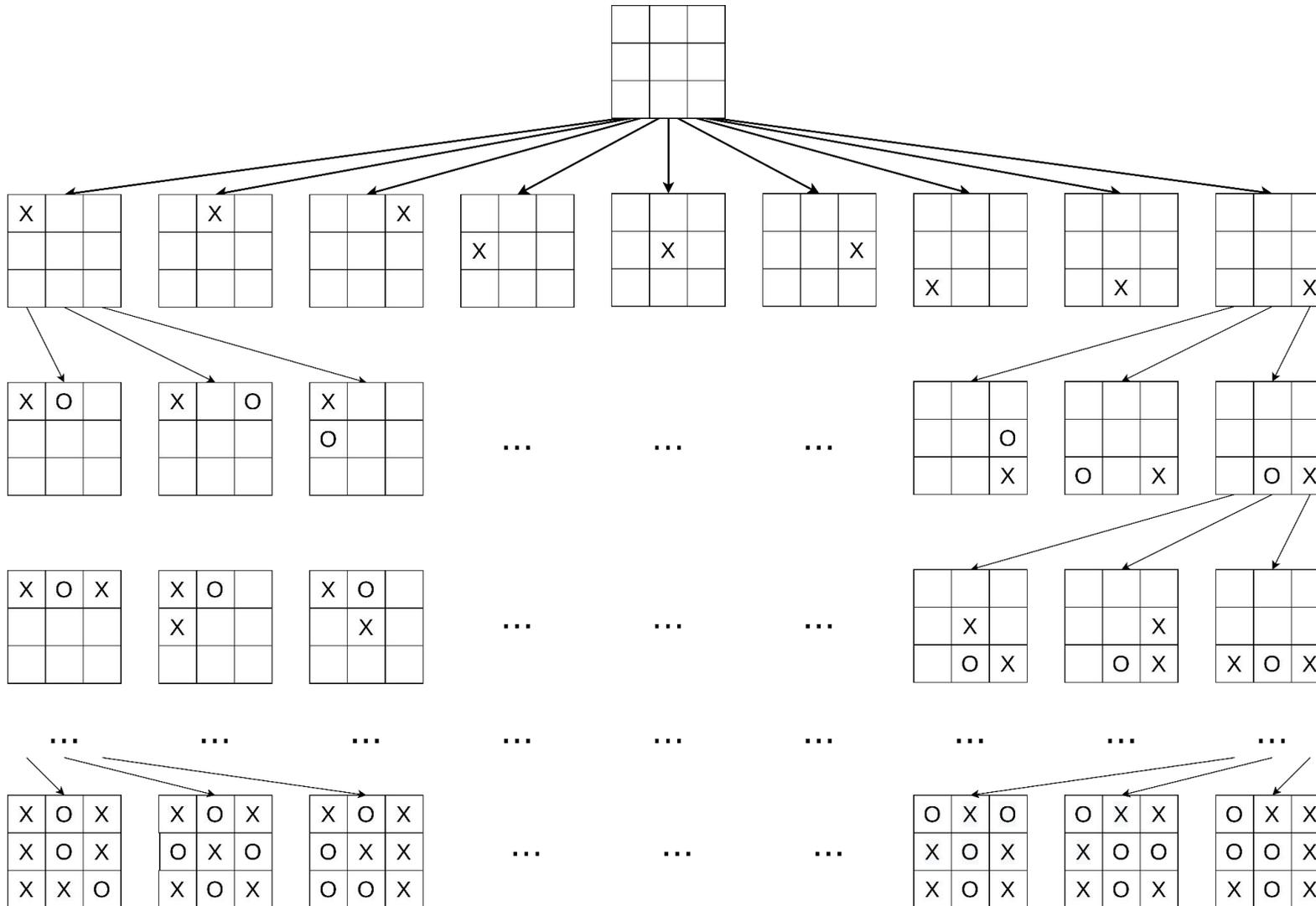
```
1 E push(E item);  
2 E peek();  
3 E pop();
```

# Pile : exemple

```
1 Stack<Color> colors =  
2   new Stack<Color>();  
3 colors.push(Color.PURPLE);  
4 colors.push(Color.ORANGE);  
5 colors.push(Color.BLUE);  
6 colors.pop(); // BLUE  
7 colors.push(Color.RED);  
8 colors.pop(); // RED  
9 colors.peek(); // ORANGE  
10 colors.pop(); // ORANGE
```



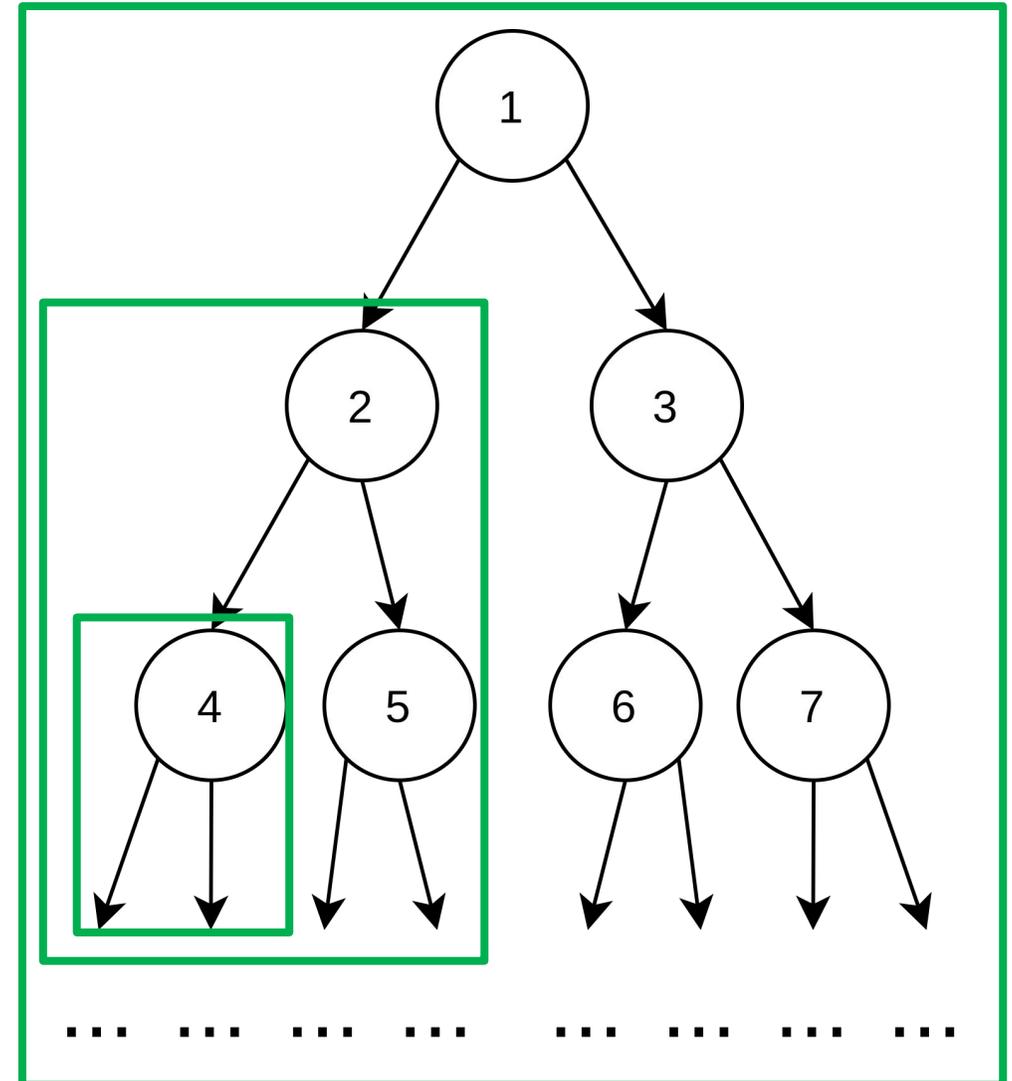
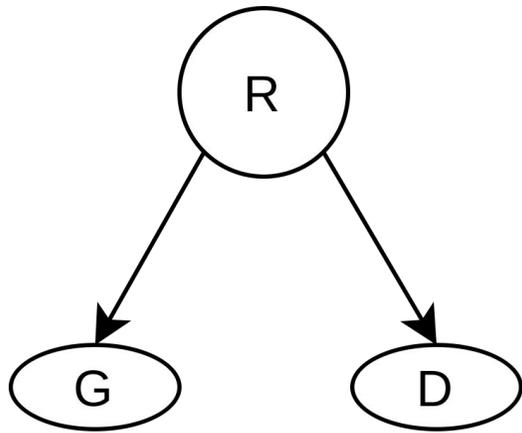
# Arbre : cas d'usage



# Arbre binaire

- Définition :

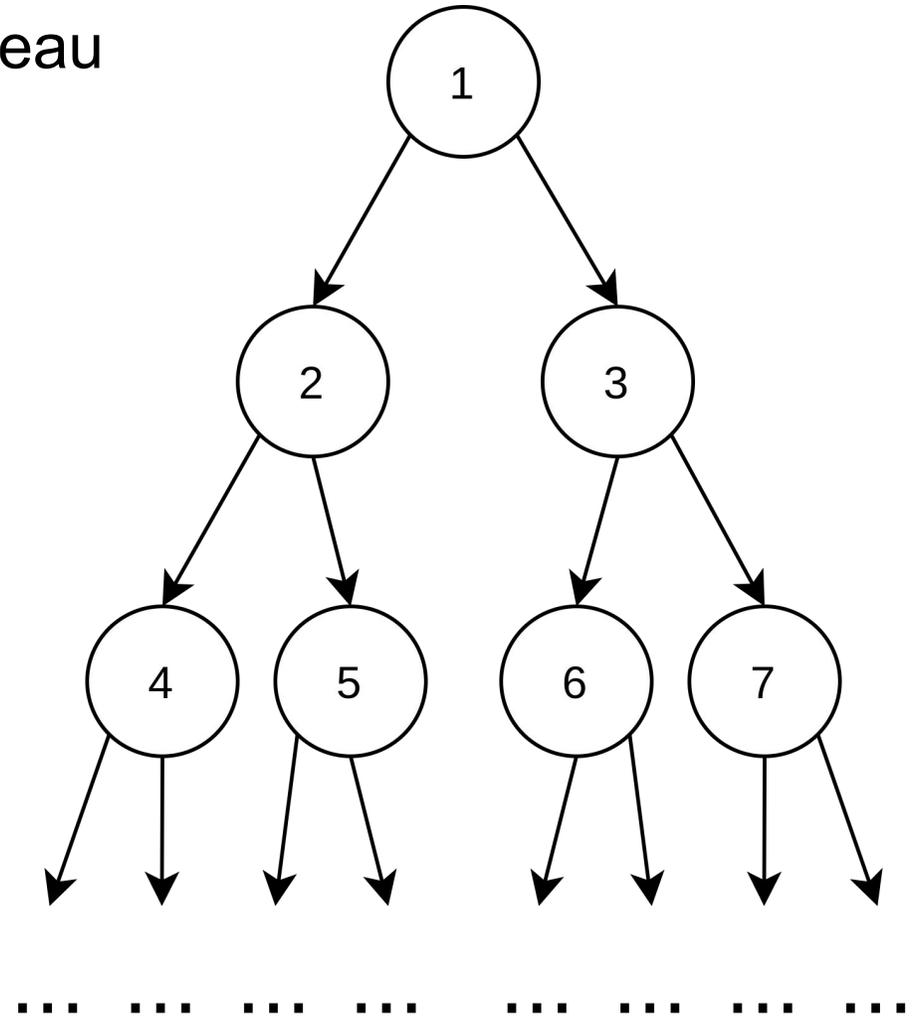
- soit vide
- soit avec une valeur **ET** 2 fils qui sont des arbres binaires



# Arbre binaire : parcours en largeur

- De la racine, descendre niveau par niveau

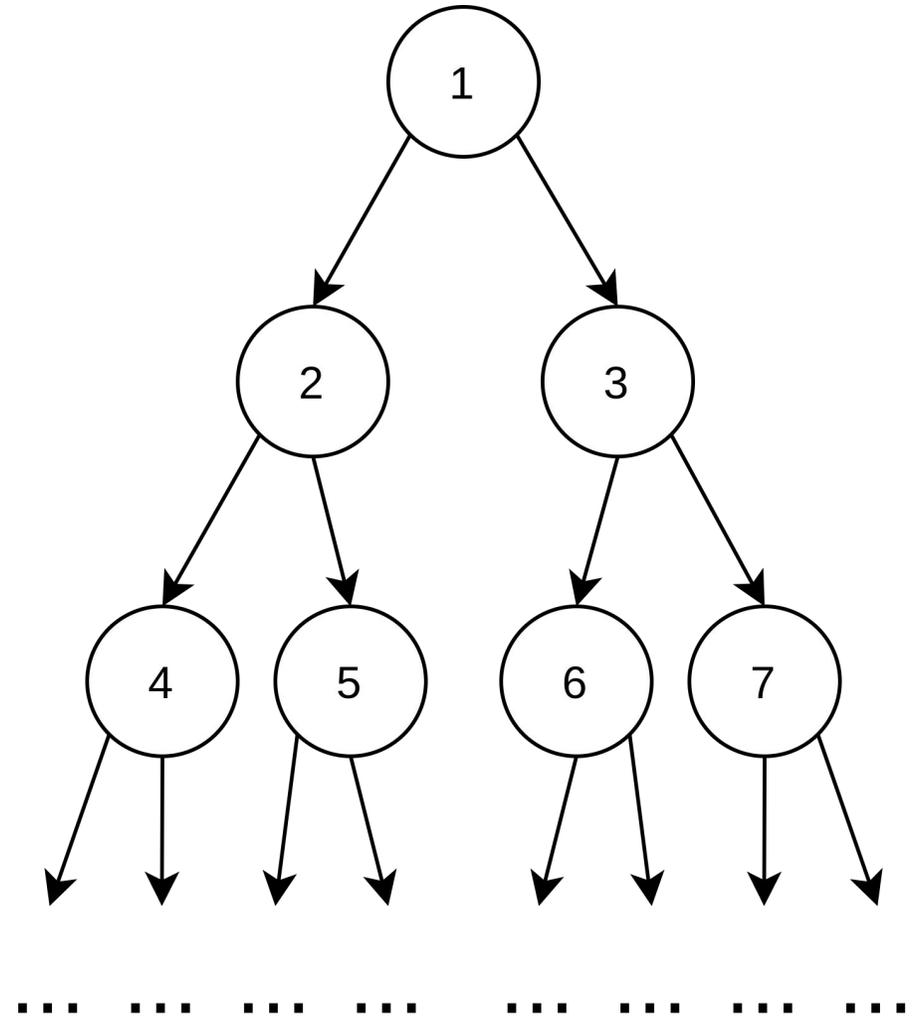
1, 2, 3, 4, 5, 6, 7 ...



# Arbre binaire : parcours en profondeur

- De la racine,  
Parcourir le sous-arbre gauche puis  
Parcourir le sous-arbre droit

1, 2, 4, 5, 3, 6, 7,



# Arbre binaire : opérations

```
1 public class BinaryTree<T> {  
2     boolean isEmpty();  
3     T getValue();  
4     boolean isLeaf();  
5     boolean hasLeft();  
6     boolean hasRight();  
7  
8     BinaryTree<T> getLeft();  
9     BinaryTree<T> getRight();  
10  
11     void setValue(T value);  
12     void setLeft(BinaryTree<T>);  
13     void setRight(BinaryTree<T>);  
14 }
```

Retourne **true** si l'arbre est vide, **false** sinon

Récupère la valeur de l'arbre

Retourne **true** si l'arbre est une feuille, il n'a pas de fils, **false** sinon

Retourne **true** si l'arbre a un fils gauche, **false**

Retourne **true** si l'arbre a un fils droit, **false** sinon

Récupère le fil gauche

Récupère le fil droit

Met à jour la valeur de l'arbre

Met à jour le sous-arbre gauche

Met à jour le sous-arbre droit

# Arbre binaire : exemple

```
1  BinaryTree<Integer> root = new BinaryTree<>();
2  root.setValue(1);
3  BinaryTree<Integer> node2 = new BinaryTree<>();
4  node2.setValue(2);
5  BinaryTree<Integer> node3 = new BinaryTree<>();
6  node3.setValue(3);
7  BinaryTree<Integer> node4 = new BinaryTree<>();
8  node4.setValue(4);
9  node2.setLeft(node3);
10 node2.setRight(node4);
11 root.setLeft(node2);
```

# Arbre binaire : récursivité

```
1 public int size() {
2     if (this.isEmpty()) {
3         return 0;
4     } else {
5         return 1 + // taille de l'arbre actuel
6                 this.getLeft().size() + // taille du sous-arbre gauche
7                 this.getRight().size(); // taille du sous-arbre droit
8     }
9 }
```

- Condition d'arrêt : l'arbre actuel est vide, sa taille est 0
- Sinon, la taille de l'arbre est 1 + la taille de ses fils
- Appel récursif à la méthode **size()** sur les deux sous-arbres

# Arbre binaire de recherche

- Définition : Arbre binaire dont
  - valeurs à gauche < racine
  - valeurs à droite > racine

$$3 < 8 < 10$$

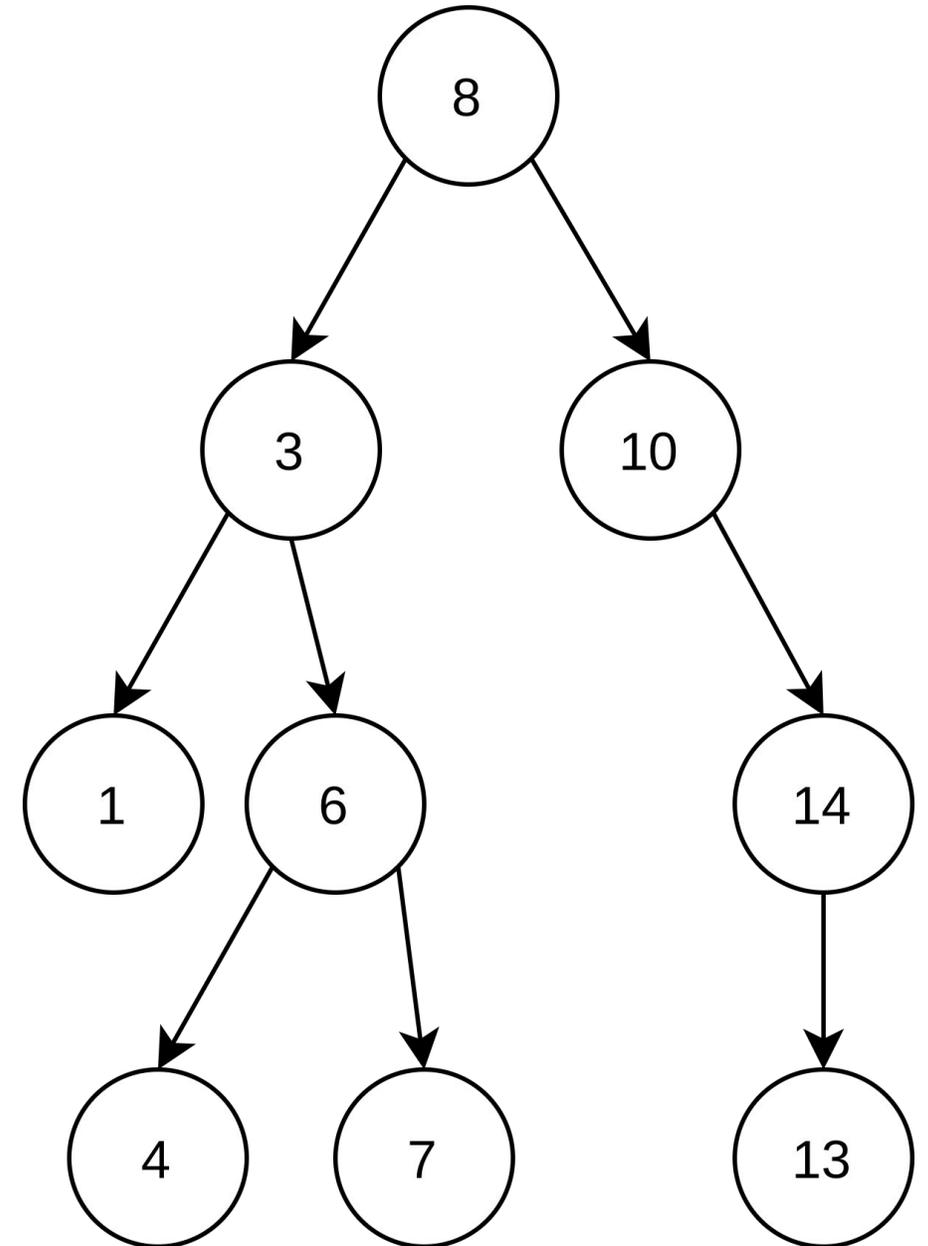
$$1 < 8 < 14$$

...

$$4 < 8 < 13$$

...

$$1 < 3 < 6$$



# Héritage

# Encapsulation

- Définition d'une classe de telle sorte à empêcher toute modification depuis l'extérieur de la classe, mais uniquement grâce à des méthodes
  - Mettre tous les attributs en **private**
  - Lecture avec les getters
  - Ecriture avec les setters
- Avantages :
  - Meilleures maintenance et évolution
  - Contrôle des modifications

# Encapsulation : Exemple

```
1 public class Person {  
2     private int age;  
3     public void setAge(int age) {  
4         if (age < 0) {  
5             System.err.println("Cannot have a negative age!");  
6             return;  
7         }  
8  
9         this.age = age;  
10    }  
11 }
```

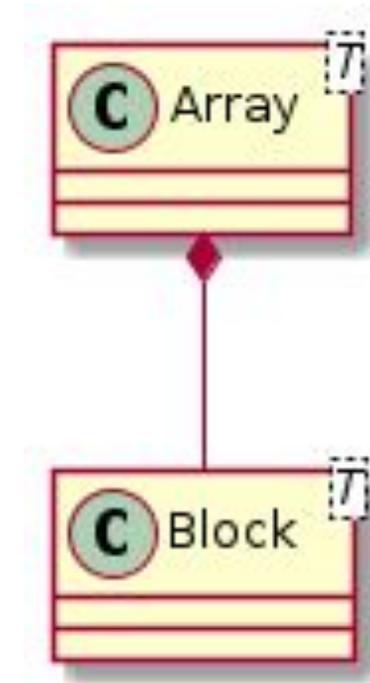
- Vérification de la valeur passée en paramètre

# Réutilisation : Composition vs Héritage

- Composition :  
Utiliser un type existant au service d'un autre
- Héritage :  
Utiliser un type existant en l'"étendant" →
  - Bénéficier des fonctionnalités inchangées
  - Modifier celles qui doivent l'être
  - Ajouter de nouvelles fonctionnalités

# Composition

- Délégation du stockage et de la manipulation de données à une (des) autres classe(s)
- Exemple
  - TP2 : Block<T>  
délègue la gestion des éléments au type Array<T>
- La notation A  B signifie que “B compose avec A”



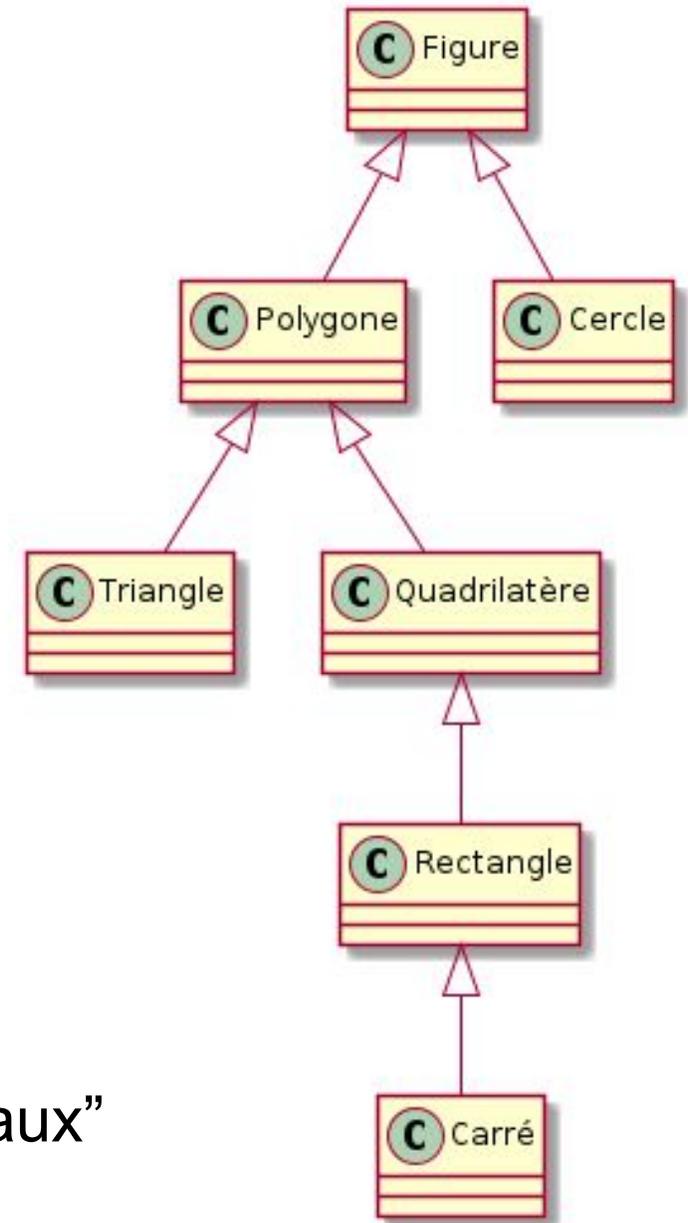
# Composition : Implémentation

- Délégation à un(des) attribut(s) interne(s) à la classe

```
1 public class Block<T> {  
2     // Composition avec Array<T>  
3     private Array<T> m_array;  
4  
5     public T get(int i) {  
6         // Délégation du stockage et de la gestion  
7         // à l'attribut de type Array<T>  
8         return this.m_array.get(i);  
9     }  
10 }
```

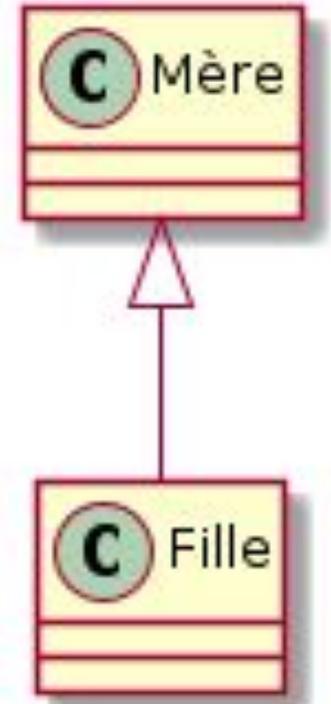
# Héritage

- Réutiliser un type existant :
  - Bénéficiaire des fonctionnalités inchangées
  - Modifier celles qui doivent l'être
  - Ajouter de nouvelles fonctionnalités
- La notation  $A \leftarrow B$  signifie B hérite (ou étend) A
- Relation hiérarchique :
  - “B est un A”, “Carré est un Rectangle”
- Notion de spécialisation :
  - “Carré est un Rectangle, avec 4 côtés égaux”



# Héritage

- B hérite de A :
  - B est la classe dite “Fille”
  - A est la classe dite “Mère”
- La class Fille peut :
  - Disposer des attributs
  - Utiliser les méthodes publiques de la classe Mère
  - Redéfinir les méthodes publiques de la classe Mère
  - Ajouter de nouvelle méthodes



# Héritage : implémentation

```
1 public class Rectangle extends Quadrilatère {
2     public int height;
3     public int width;
4     public void setHeight(int height) {
5         this.height = height;
6     }
7     public int surface() {
8         return this.height * this.width;
9     }
10 }
11 public class Carré extends Rectangle {
12     public void setHeight(int height) {
13         this.height = height;
14         this.width = height;
15     }
16     public String toString() {
17         return this.surface() + " ";
18     }
19 }
```

■ Rectangle “**extends**” Quadrilatère

■ Attributs de la class Rectangle

■ Définition de la méthode **setHeight**

■ Définition de la méthode **surface**

■ Carré “**extends**” Rectangle

■ Rédéfinition de **setHeight**

■ Utilisation des attributs de la classe mère **Rectangle**

■ Utilisation de la méthode **surface** de la classe Mère

# Rappel Visibilité

- Visibilité : définition de la permission d'accès
- **sans visibilité** : accessible par les objets du package
  - Aussi appelé "Package private"
- **public** : accessible par tous les objets
- **private** : accessible par les objets du type
- **protected** : accessible par les objets du package et "les classes filles"
- Bonne pratique : attributs toujours en privé

# Héritage : visibilité des attributs

```
1 public class Rectangle extends Quadrilatère {
2     protected int height;
3     protected int width;
4     public void setHeight(int height) {
5         this.height = height;
6     }
7     public int surface() {
8         return this.height * this.width;
9     }
10 }
11 public class Carré extends Rectangle {
12     public void setHeight(int height) {
13         this.height = height;
14         this.width = height;
15     }
16     public String toString() {
17         return this.surface() + "";
18     }
19 }
```

- Visibilité **protected** plutôt que **public** :  
Bonne pratique → Mettre les attributs **private**  
Mais pour que les classes filles accède à ces attributs, on utilise **protected**

# Héritage VS Implémentation d'interface

## ■ Héritage

- Héritage **simple** (en Java)
- **Transmission des attributs, des signatures de méthodes et du code**

## ■ Interface

- Implémentation **multiple**
- **Transmission des signatures de méthodes**

## ■ En commun

- Polymorphisme
- Late-binding

# Héritage et Classe Abstraite

- Une classe abstraite est un type non instanciable mais qui :
  - a des attributs
  - a des méthodes implémentées
  - a des signatures de méthodes déclarées ou méthodes abstraites
- Les classes qui héritent de classe abstraite **DOIVENT** implémenter les méthodes abstraites déclarées

# Héritage et Classe Abstraite : exemple

```
1 public abstract class Figure {
2     protected String color;
3     public abstract int surface();
4     public String getColor() {
5         return this.color;
6     }
7 }
8 public class Cercle extends Figure {
9     public int surface() {
10        return // implementation of surface
11    }
12    public String toString() {
13        return this.color;
14    }
15 }
```

■ Déclaration de classe abstraite

■ Déclaration de la méthode abstraite **surface**

■ Implémentation obligatoire de la méthode abstraite **surface**

■ Héritage des autres membres de la classe Figure

# Le mot-clé super

- Nouveau mot-clé **super**
  - Référence la “superclass” ou la classe mère
  - Souvent implicite, comme le **this**

# Le mot-clé super : exemple

```
1 public class Rectangle extends Quadrilatère {
2     public int surface() { return this.height * this.width; }
3     public String toString() { return "RECT " + this.surface(); }
4 }
5 public class Carré extends Rectangle {
6     public String toString() { return super.toString() + " " + this.surface(); }
7 }
```

■ Mot-clé **super** fait référence à la superclass Rectangle

■ Résolution du code : la méthode surface  $\emptyset$  dans Carré, lookup dans la classe mère  
équivalent à  
**super.surface()** ou **surface()**

# Le type racine : Object

- Racine de toutes hiérarchies de classes
- Définit les méthodes :
  - toString() : renvoie une représentation textuelle de l'objet
  - hashCode() : renvoie valeur numérique qui correspond à une instance particulière d'une classe
  - equals() : teste l'égalité de deux instances

# Quiz de révision