

## Plan

- Introduction
- Type abstrait
- Généricité
- Structures de données
- Héritage et polymorphisme
- Modélisation - Diagramme de classes

1

## Chasse au trésor (V0)

ChasseAuTrésor.java

```
package robotSimple.v0;

import java.util.Random;
import java.util.Scanner;

enum Direction { Nord, Est, Sud, Ouest }

public class ChasseAuTrésor {

    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);
        Random generateurAleatoire = new Random();
        // choisir au hasard la position du trésor
        int xTrésor = generateurAleatoire.nextInt(10);
        int yTrésor = generateurAleatoire.nextInt(10);

        // choisir au hasard la position du robot et fixer sa direction
        int xRobot = generateurAleatoire.nextInt(10);
        int yRobot = generateurAleatoire.nextInt(10);
        Direction dirRobot = Direction.Nord;
    }
```

2

## Chasse au trésor (V0)

ChasseAuTrésor.java

```
// déterminer (le carré de) la distance du robot au trésor
int distance = carreDistance(xRobot, yRobot, xTrésor, yTrésor);
int nbCoups = 0;

// répéter tant que le robot n'est pas sur le trésor
while (distance != 0) {

    //indiquer au joueur la distance au trésor
    System.out.println("direction : " + dirRobot + " distance : " + distance);

    // obtenir un ordre et l'exécuter
    System.out.print("[a]vancer, [t]ourner, [s]top ? ");
    String ordre = entree.nextLine();

    if (ordre.equals("a")) {
        // Faire avancer le robot d'une unité dans sa direction actuelle
        switch (dirRobot) {
            case Nord : --yRobot; break;
            case Est : ++xRobot; break;
            case Sud : ++yRobot; break;
            case Ouest : --xRobot; break;
        }
    }
}
```

3

## Chasse au trésor (V0)

ChasseAuTrésor.java

```
else if (ordre.equals("t")) {

    // Faire tourner le robot d'un 1/4 de tour "à droite"
    switch (dirRobot) {
        case Nord : dirRobot = Direction.Est; break;
        case Est : dirRobot = Direction.Sud; break;
        case Sud : dirRobot = Direction.Ouest; break;
        case Ouest : dirRobot = Direction.Nord; break;
    }

    else if (ordre.equals("s")) { break; } // abandon

    // recalculer la distance du robot au trésor
    distance = carreDistance(xRobot, yRobot, xTrésor, yTrésor);
    ++nbCoups;
} // fin while
```

4

## Chasse au trésor (V0)

ChasseAuTrésor.java

```
// fin itération ; 2 sorties possibles : distance = 0 ou abandon
if (distance == 0) {
    System.out.println("Gagné en " + nbCoups + " coups");
}
else {
    System.out.println("Abandon au bout de " + nbCoups + " coups");
}
entree.close();
} // fin main

// distance entre deux points
static int carreDistance(int xr, int yr, int xt, int yt) {
    return (xr - xt) * (xr - xt) + (yr - yt) * (yr - yt);
}

} // fin ChasseAuTrésor
```

5

## Type abstrait

- Encapsulation
  - regrouper et cacher les variables qui caractérisent un robot pour former une nouvelle entité
  - état interne *invisible* de l'extérieur de l'entité
  - maintien de la cohérence interne
- Abstraction
  - regrouper les opérations dans cette entité
  - garantir la cohérence de l'état interne
  - fournir des opérations abstraites

☞ raisonner sur les opérations abstraites et non sur les variables

7

## Critique

- Aucune séparation entre données et traitements
  - Aucune relation entre abscisse, ordonnée, direction :
    - éparpillement des données donc des actions,
    - maintien cohérence difficile (modification)
  - n robots  $\Rightarrow$  trois tableaux
    - abscisses, ordonnées, directions
  - Ajout de caractéristiques aux robots (vitesse, poids, taille, ...)  $\Rightarrow$  ajout de tableau pour chacune...
- ☞ Ça devient vite lourd à gérer
- éparpillement  $\Rightarrow$  évolutivité compliquée

6

## Type abstrait en java

- Type abstrait = une classe
- Encapsulation
  - variables caractéristiques = attributs privés
- Abstraction
  - opérations = méthodes
  - garantir la cohérence de l'état interne

8

## Le robot devient autonome

Robot.java

```
package robotSimple.v2;

public class Robot {
```

But : regrouper en une classe :  
- les *valeurs caractéristiques* d'un robot indépendamment de toute application en les rendant invisibles de l'extérieur  
- les *opérations* qu'on peut réaliser sur un robot, indépendamment de toute application

// attributs cachés  
private int x, y;  
private Direction direction;

**Encapsulation des données d'un Robot**

// initialiser un robot  
public Robot(int xr, int yr, Direction dir) {  
 x = xr; y = yr; direction = dir;  
}

Opération d'initialisation = **constructeur**

Appelé automatiquement lors de la création d'une variable de type Robot

9

## Créer et initialiser un robot

ChasseAuTresor.java

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

        // créer un robot
        Robot nono = new Robot(-3, 7, Direction.Sud);

        // créer un robot
        Robot astro = new Robot(10, 10, Direction.Nord);
    }
```

nono

astro

x -3  
y 7  
direction Sud

x 10  
y 10  
direction Nord

10

## Créer et initialiser un robot

ChasseAuTresor.java

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

        // créer un robot
        Robot nono = new Robot(-3, 7, Direction.Sud);

        // créer un robot
        Robot astro = new Robot(10, 10, Direction.Nord);
    }
```

nono

astro

instances de Robot  
• créées par **new**  
• désignées par **référence**

x 10  
y 10  
direction Nord

11

## Créer et initialiser un robot

ChasseAuTresor.java

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

        // créer un robot
        Robot nono = new Robot(-3, 7, Direction.Sud);

        // créer un robot
        Robot astro = new Robot(10, 10, Direction.Nord);
    }
```

nono

astro

références de Robot  
• peuvent désigner une **instance**

x -3  
y 7  
direction Sud

x 10  
y 10  
direction Nord

12

## Opérations réalisées par un robot

Robot.java

```
// Faire avancer le robot (this) d'une unité dans sa direction actuelle
public void avancer() {
    switch (this.direction) {
        case Nord : --this.y; break;
        case Est  : ++this.x; break;
        case Sud  : ++this.y; break;
        case Ouest : --this.x; break;
    }
}

// Faire tourner le robot (this) d'un 1/4 de tour "à droite"
public void tourner() {
    switch (this.direction) {
        case Nord : this.direction = Direction.Est; break;
        case Est  : this.direction = Direction.Sud; break;
        case Sud  : this.direction = Direction.Ouest; break;
        case Ouest : this.direction = Direction.Nord; break;
    }
}
```

méthode : agit sur une instance de robot désignée par la référence implicite **this**

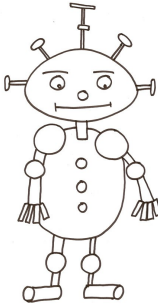
13

## Agir sur un robot

ChasseAuTresor.java

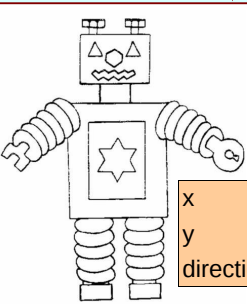
```
public class ChasseAuTresor {
    public static void main(String[] args) {
        // faire avancer le robot
        nono.avancer();
        // faire tourner le robot
        astro.tourner();
    }
}
```

nono



x	-3
y	7
direction	Sud

astro



x	10
y	10
direction	Nord

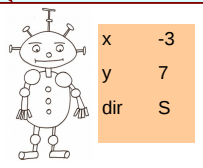
14

## Agir sur un robot

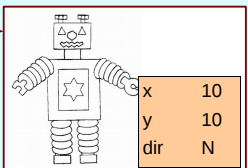
public class ChasseAuTresor {  
 public static void main(String[] args)  
 {  
 // faire avancer le robot  
 nono.avancer();  
 astro  
 }  
}

nono

astro



x	-3
y	7
dir	S



x	10
y	10
dir	N

15

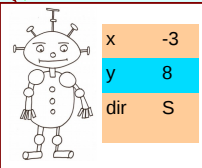
## Agir sur un robot

public class ChasseAuTresor {  
 public static void main(String[] args)  
 {  
 // faire avancer le robot  
 nono.avancer();  
 astro  
 }  
}

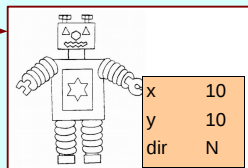
nono

astro

this



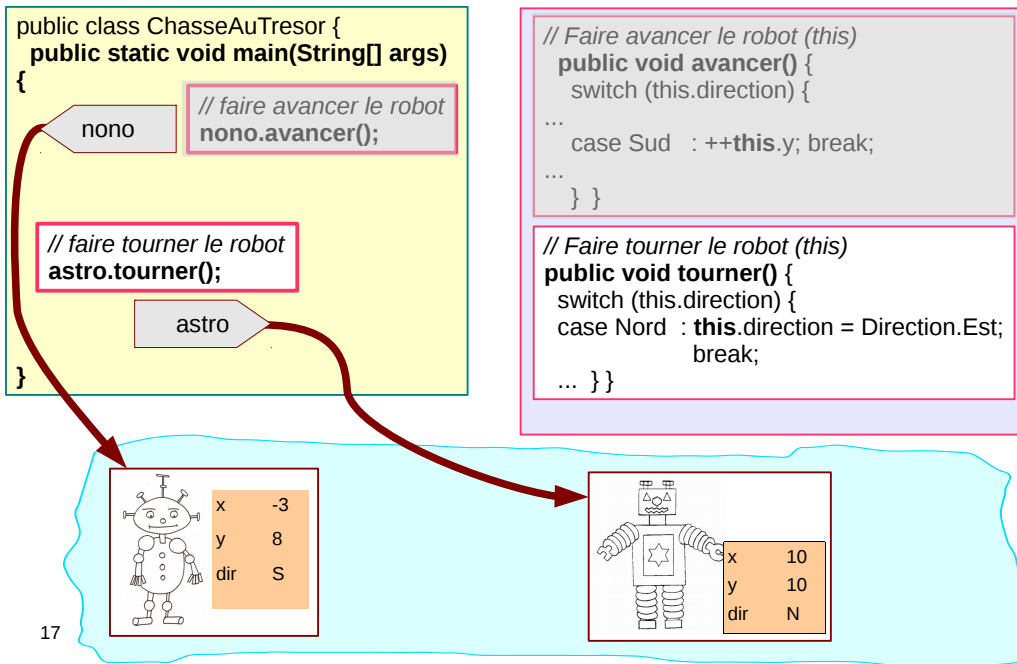
x	-3
y	8
dir	S



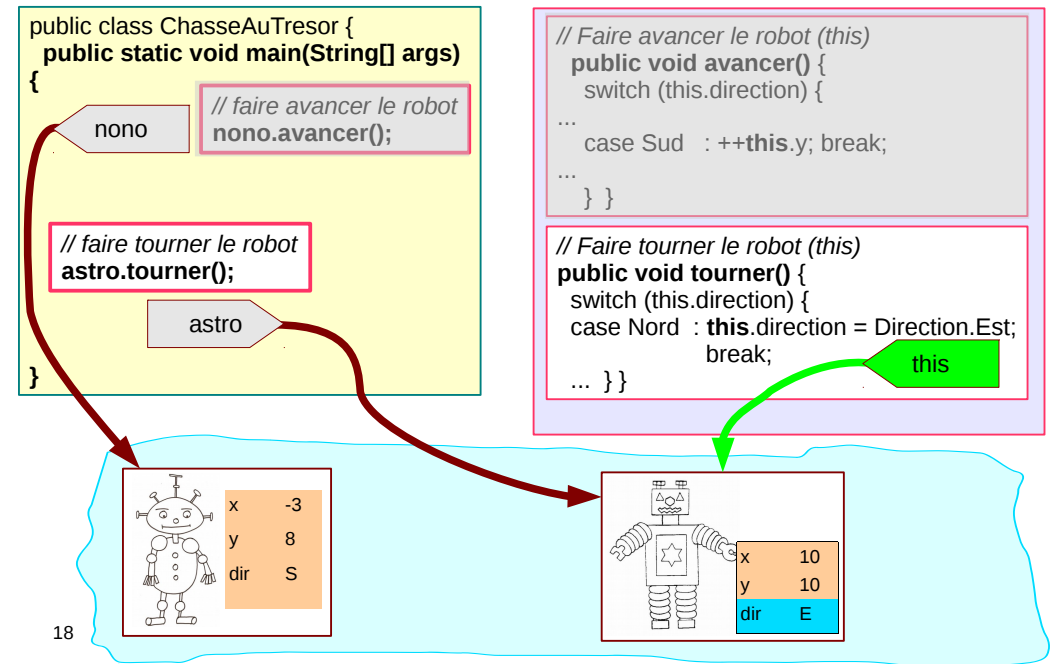
x	10
y	10
dir	N

16

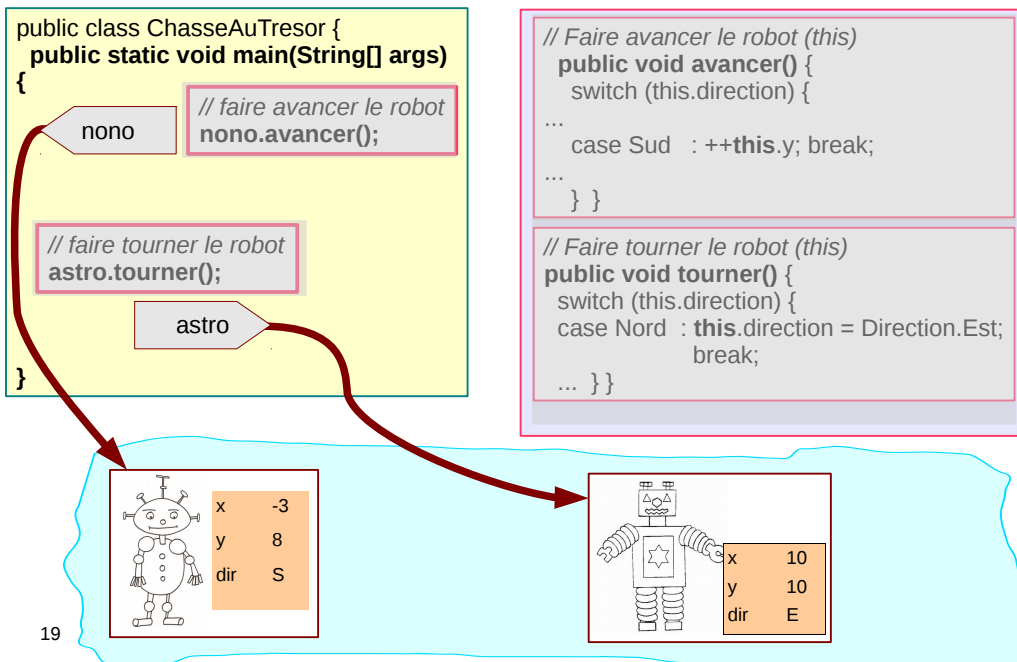
## Agir sur un robot



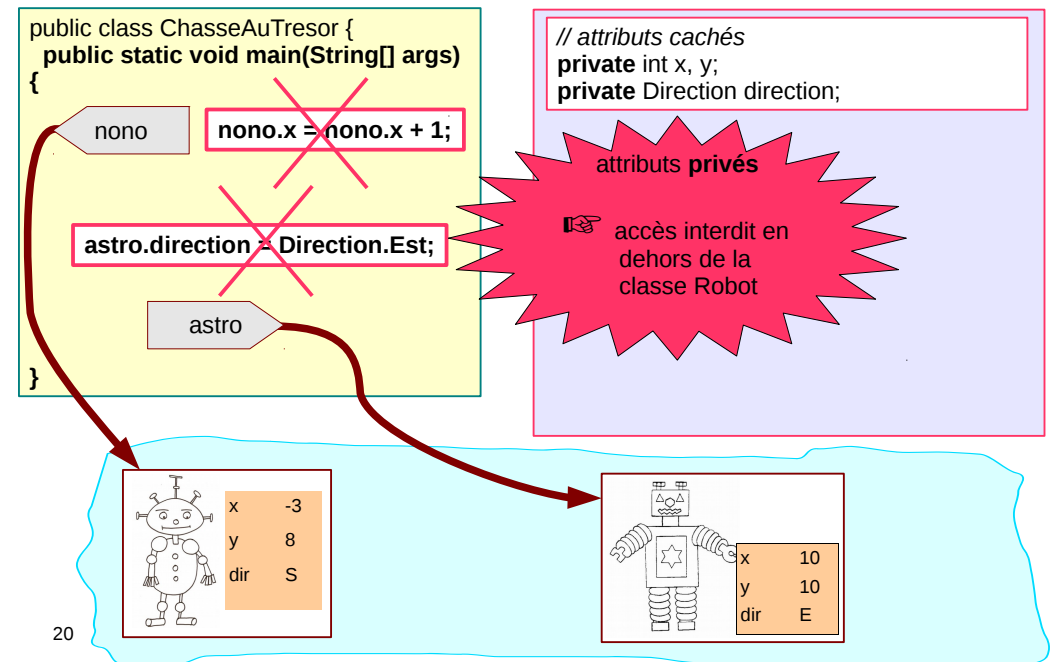
## Agir sur un robot



## Agir sur un robot



## Agir sur un robot



## Opérations réalisées par un robot

Robot.java

```
// Consulter l'abscisse du robot (this)
public int getX() { return this.x; }
```

```
// Consulter l'ordonnée du robot (this)
public int getY() { return this.y; }
```

```
// Consulter la direction du robot (this)
public Direction getDirection() { return this.direction; }
```

```
// Déterminer si le robot this et le robot autre
// sont sur la même case
public boolean memePosition(Robot autre)
{
    return this.x == autre.x &&
           this.y == autre.y;
}
```

méthodes de consultation des caractéristiques

autre(s) méthode(s)

attributs **privés**  
accès **autorisé** dans la classe Robot


21

## Agir sur plusieurs robot

```
public class ChasseAuTresor {
    public static void main(String[] args)
    {
        boolean memeCase =
            nono.memePosition(astro);
    }
```

nono

astro



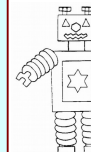
x	-3
y	8
dir	S

22

```
public boolean memePosition(Robot autre)
{
    return this.x == autre.x &&
           this.y == autre.y;
}
```

this

autre



x	10
y	10
dir	E

## Opérations réalisées par un robot

Robot.java

```
// Consulter l'abscisse du robot (this)
public int getX() { return this.x; }
```

```
// Consulter l'ordonnée du robot (this)
public int getY() { return this.y; }
```

```
// Consulter la direction du robot (this)
public Direction getDirection() { return this.direction; }
```

```
// Déterminer si le robot this et le robot autre
// sont sur la même case
public boolean memePosition(Robot autre)
{
    return this.getX() == autre.getX() &&
           this.getY() == autre.getY();
}
```

méthodes de consultation des caractéristiques

Utiliser les fonctions de consultation : plus évolutif


23

## Agir sur plusieurs robot

```
public class ChasseAuTresor {
    public static void main(String[] args)
    {
        boolean memeCase =
            nono.memePosition(astro);
    }
```

nono

astro



x	-3
y	8
dir	S

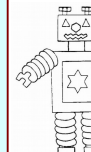
24

```
public boolean memePosition(Robot autre)
{
    return this.getX() == autre.getX() &&
           this.getY() == autre.getY();
}
```

this

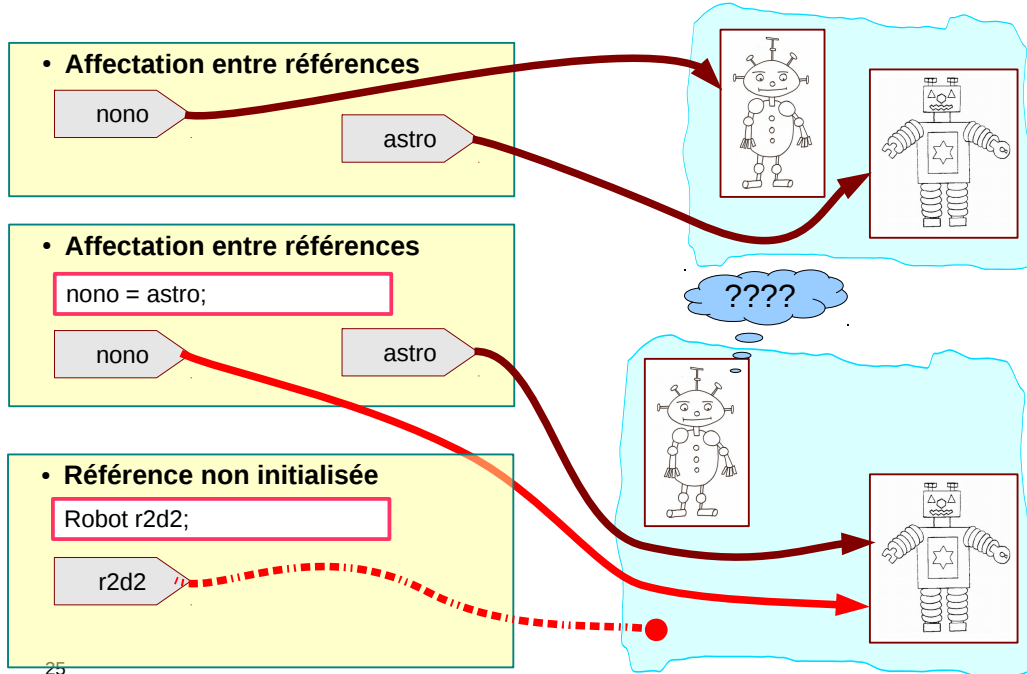
autre

```
// Consulter l'abscisse du robot (this)
public int getX() { return this.x; }
```



x	10
y	10
dir	E

## Référence, instance



## le type abstrait Robot

Robot.java

```
public class Robot {
    // attributs cachés
    private int x, y;
    private Direction direction;

    // initialiser un robot
    public Robot(int xr, int yr, Direction dir) {
        x = xr; y = yr; direction = dir;
    }

    // Faire avancer le robot (this) d'une unité dans sa direction actuelle
    public void avancer() {
        switch (this.direction) {
            case Nord : --this.y; break;
            case Est  : ++this.x; break;
            case Sud  : ++this.y; break;
            case Ouest : --this.x; break;
        }
    }
}
```

**Attributs = état interne**

**constructeur : initialiser l'état interne**

**méthode = opération**

- agit sur l'état interne
- garantit sa cohérence

26

## le type abstrait Robot

Robot.java

```
// Faire tourner le robot (this) d'un 1/4 de tour "à droite"
public void tourner() {
    switch (this.direction) {
        case Nord : this.direction = Direction.Est; break;
        case Est  : this.direction = Direction.Sud; break;
        case Sud  : this.direction = Direction.Ouest; break;
        case Ouest : this.direction = Direction.Nord; break;
    }
}

// accesseurs = méthodes de consultation
public int getX() { return this.x; }
public int getY() { return this.y; }
public Direction getDirection() { return this.direction; }

// Déterminer si le robot this et le robot autre sont sur la même case
public boolean memePosition(Robot autre) {
    return this.getX() == autre.getX() && this.getY() == autre.getY();
}

// fin classe Robot
```

**méthode = opération**

- agit sur l'état interne
- garantit sa cohérence

**selon besoin**

27

## Chasse au trésor

ChasseAuTresor.java

```
package robotSimple.v2;

import java.util.Random;
import java.util.Scanner;

public class ChasseAuTresor {

    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);
        Random generateurAleatoire = new Random();
        // choisir au hasard la position du trésor
        int xTresor = generateurAleatoire.nextInt(10);
        int yTresor = generateurAleatoire.nextInt(10);

        // choisir au hasard la position du robot et fixer sa direction
        int xr = generateurAleatoire.nextInt(10);
        int yr = generateurAleatoire.nextInt(10);

        // créer un robot
        Robot astro = new Robot(xr, yr, Direction.Nord);
    }
}
```

**Création et initialisation d'une instance**

- Désignation par référence

28



## Chasse au trésor

ChasseAuTresor.java

```
// déterminer (le carré de) la distance du robot au trésor
int distance = carreDistance(astro.getX(), astro.getY(), xTresor, yTresor);
int nbCoups = 0;

// répéter tant que le robot n'est pas sur le trésor
while (distance != 0) {

    //indiquer au joueur la distance au trésor
    System.out.println("direction : " + astro.getDirection() + " distance : " + ..

    // obtenir un ordre
    System.out.print("[a]vancer, [t]ourner, [s]top ? ");
    String ordre = entree.nextLine();

    //exécuter l'ordre
    if (ordre.equals("a")) { astro.avancer(); }
    else if (ordre.equals("t")) { astro.tourner(); }
    else if (ordre.equals("s")) { break; }

    // ...
} // fin main
// distance entre deux points ...
} // fin ChasseAuTresor
```

consulter les caractéristiques

appliquer une opération sur une instance

29

## Fonction et Robot

ChasseAuTresor.java

```
public class ChasseAuTresor {
    public static void main(String[] args) {
        // créer deux robots
        Robot nono = new Robot(-3, 7, Direction.Sud);
        Robot astro = new Robot(10, 10, Direction.Nord);
        // ...

        // déterminer quel est le robot le plus proche du trésor
        Robot leMeilleur =
            lePlusProche(nono, astro, xTresor, yTresor);

    } // fin main

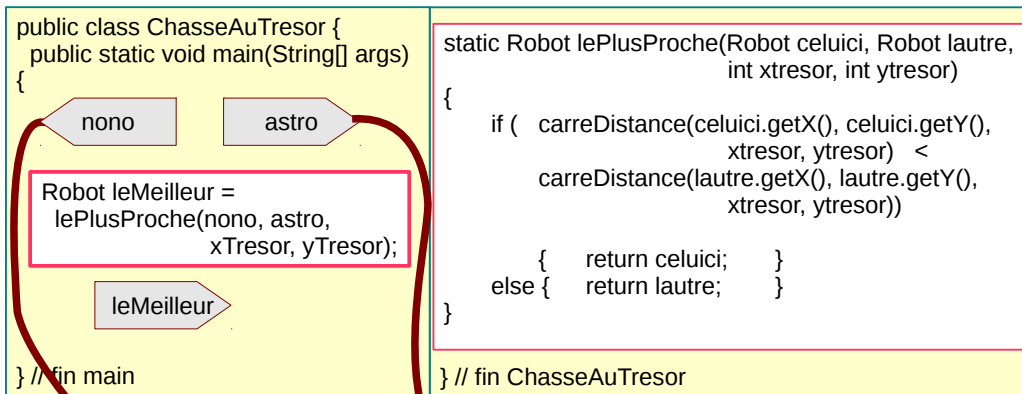
    // déterminer quel est le robot le plus proche du trésor
    static Robot lePlusProche(Robot celui1, Robot lautre, int xtresor, int ytresor)
    {
        if ( carreDistance(celui1.getX(), celui1.getY(), xtresor, ytresor) <
            carreDistance(lautre.getX(), lautre.getY(), xtresor, ytresor) )
        {
            return celui1;
        }
        else {
            return lautre;
        }
    }

} // fin ChasseAuTresor
```

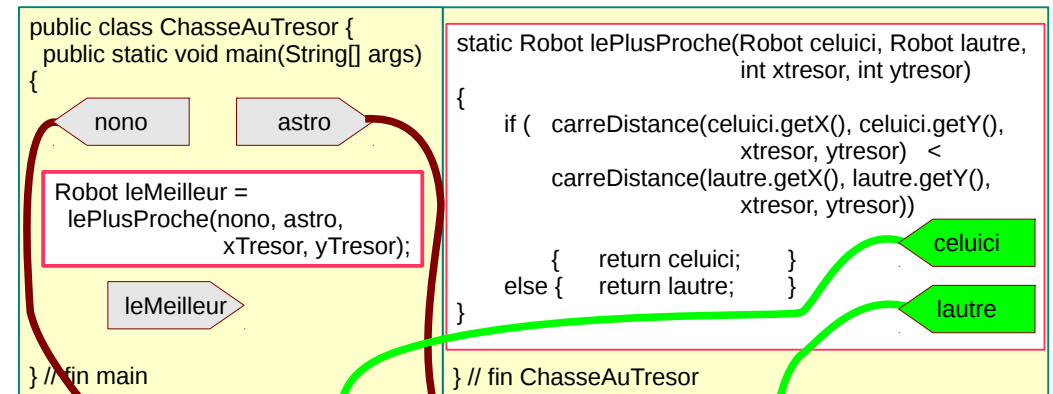
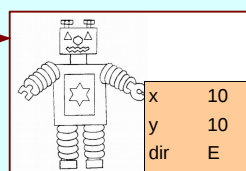
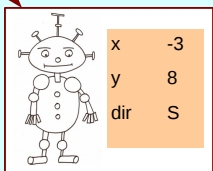
les paramètres sont des références

30

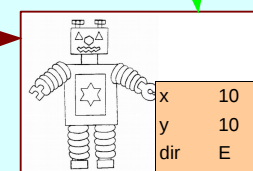
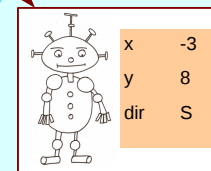
## Fonction et Robot



31

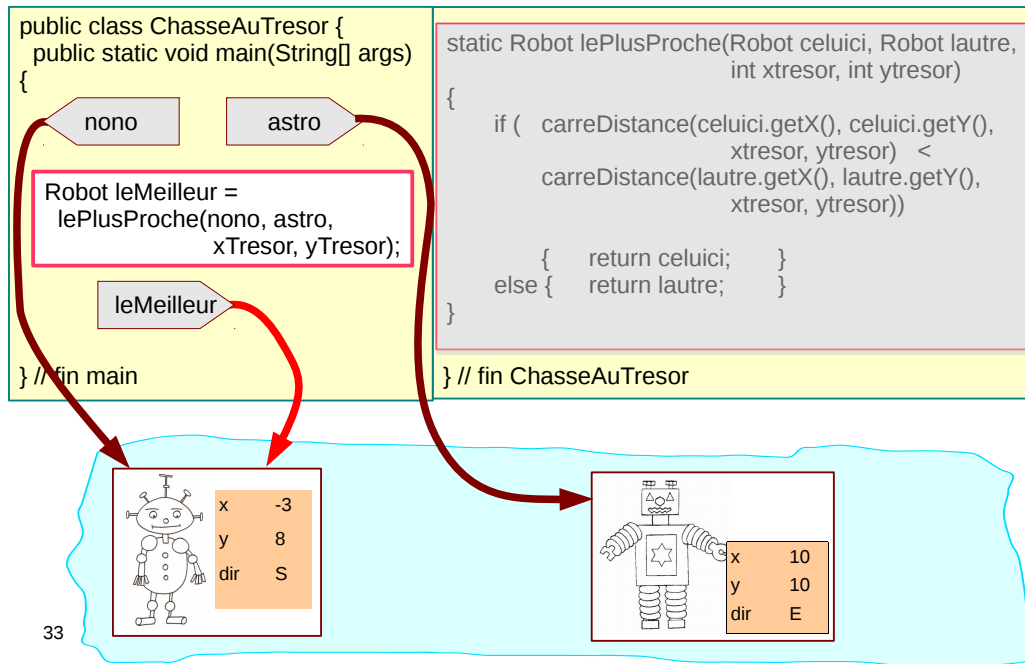


32





## Fonction et Robot



33

## Modéliser un système

### Deux étapes

#### 2) Réalisation ou mise en œuvre ou implémentation

- Choix de la représentation de l'état interne (variables = attributs)
- Programmation des opérations de la spécification

### Avantages

- travail en parallèle
- indépendance du « client » par rapport à une implémentation donnée
- modification de l'implémentation sans remise en cause des programmes « clients »

### Condition : respect strict de la spécification

35

## Modéliser un système

### Deux étapes

#### 1) Spécification

- définir les informations qui caractérisent le système
  - domaine de valeurs
  - contraintes de validité
- définir les opérations caractéristiques du système
  - pré-conditions d'utilisation (prérequis)
  - rôle de l'opération
  - effet précis sur le système (post-conditions)

34

## Application en java

#### Première possibilité

- spécification informelle (mais précise)
- implémentation avec une classe (voir classe Robot)

#### Deuxième possibilité

- spécification en java à l'aide d'une *interface*
- implémentation avec une *classe reliée à l'interface*.

#### Intérêt de la deuxième approche

- Le compilateur vérifie que l'implémentation respecte la spécification
- rendre les programmes indépendants de toute implémentation

36

## spécification du type abstrait Robot

SpecRobot.java

```
public interface SpecRobot {
    // un Robot est défini par une position sur une grille infinie et
    // une direction parmi 4 possibles

    // initialiser un robot avec deux coordonnées et une direction

    // Faire avancer le robot d'une unité dans sa direction actuelle
    // sans effet sur sa direction
    public void avancer();

    // Faire tourner le robot (this) d'un tour "à droite"
    // le robot ne change pas de position
    public void tourner();

    // Consulter la position et la direction
    public int getX();
    public int getY();
    public Direction getDirection();

    // Déterminer si 2 robots sont sur la même case
    public boolean memePosition(SpecRobot autre);
} // fin interface
```

pas d'attribut

pas de constructeur

signature des méthodes avec description des paramètres, des pré-requis et de l'effet

37

## implémentation du type abstrait Robot

Robot.java

```
public class Robot implements SpecRobot {

    // attributs cachés
    private int x, y;
    private Direction direction;

    // initialiser un robot
    public Robot(int xr, int yr, Direction dir) {

    }

    // Faire avancer le robot d'une unité dans sa direction actuelle
    public void avancer() { ... }

    // Déterminer si 2 robots sont sur la même case
    public boolean memePosition(SpecRobot autre) {

    }
} // fin classe Robot
```

attributs = représentation interne

implémentation des constructeurs

contenu identique à la classe Robot présentée plus haut

implémentation obligatoire des méthodes de l'interface

38

## Autre implémentation du type abstrait Robot

RobotMasque.java

```
public class RobotMasque implements SpecRobot {

    // attributs cachés
    private ...;
    private ...;

    // initialiser un robot
    public RobotMasque(int xr, int yr, Direction dir) {

    }

    // Faire avancer le robot d'une unité dans sa direction actuelle
    public void avancer() { ... }

    // Déterminer si 2 robots sont sur la même case
    public boolean memePosition(SpecRobot autre) {

    }
} // fin classe RobotMasque
```

représentation interne différente

implémentation des constructeurs => initialisation des attributs

fonctionnement externe identique à la première implémentation

implémentation obligatoire des méthodes de l'interface à l'aide de la nouvelle représentation

39

## Référence, instance

- On peut déclarer une référence du type d'une interface

```
SpecRobot nono;
```

nono

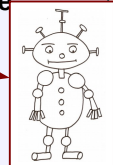
- On ne peut pas créer une instance du type d'une interface

```
new SpecRobot(10, 10, Direction.Nord)
```

- Une référence du type d'une interface peut désigner une instance du type d'une implémentation

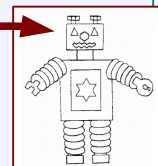
```
nono = new Robot(3, -7, Direction.Sud)
```

nono



```
SpecRobot astro = new RobotMasque(10, 10, Direction.Nord)
```

astro



40

## Référence, instance

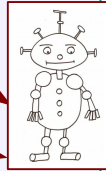
- Une référence du type d'une implémentation ne peut désigner une instance du type d'une autre implémentation

```
SpecRobot nono =
    new Robot(3, -7, Direction.Sud)
```

nono

```
RobotMasque r2d2 = nono
```

r2d2



### Recommandation

Toujours désigner les instances avec des références du type de l'interface

41

## Référence, instance

```
// déterminer quel est le robot le plus proche du trésor
static Robot lePlusProche(Robot celuiCi, Robot lautre, int xtresor, int ytresor)
{ ... }
```

on ne peut passer en paramètre que des instances de l'implémentation **Robot**

```
// déterminer quel est le robot le plus proche du trésor
static SpecRobot lePlusProche(SpecRobot celuiCi, SpecRobot lautre, ...)
{ ... }
```

on peut passer en paramètre des instances de **Robot** et de **RobotMasque**

42

## Chasse au trésor

ChasseAuTrésor.java

```
package robotSimple.v3;

import java.util.Random;
import java.util.Scanner;
```

```
public class ChasseAuTrésor {
```

```
    public static void main(String[] args) {
        Scanner entree = new Scanner(System.in);
        Random generateurAleatoire = new Random();
```

```
        // choisir au hasard la position du trésor
        int xTresor = generateurAleatoire.nextInt(10);
        int yTresor = generateurAleatoire.nextInt(10);
```

```
        // choisir au hasard la position du robot et fixer sa direction
        int xr = generateurAleatoire.nextInt(10);
        int yr = generateurAleatoire.nextInt(10);
```

```
        // créer un robot
        SpecRobot astro = new Robot(xr, yr, Direction.Nord);
```

- Référence du type de l'interface
- Instance du type d'une des implémentations

43

## Chasse au trésor

ChasseAuTrésor.java

```
// déterminer (le carré de) la distance du robot au trésor
int distance = carreDistance(astro.getX(), astro.getY(), xTresor, yTresor);
int nbCours = 0;
```

```
// répéter tant que le robot n'est pas sur le trésor
while (distance != 0) {
```

```
    //indiquer au joueur la distance au trésor
    System.out.println("direction : " + astro.getDirection() + " distance : " + ...
```

```
    // obtenir un ordre
    System.out.print("[a]vancer, [t]ourner, [s]top ? ");
    String ordre = entree.nextLine();
```

```
    //exécuter l'ordre
    if (ordre.equals("a")) { astro.avancer(); }
    else if (ordre.equals("t")) { astro.tourner(); }
    else if (ordre.equals("s")) { break; }
```

```
    // ...
} // fin main
// distance entre deux points ...
} // fin ChasseAuTrésor
```

inchangé

44

## Fonction et Robot

ChasseAuTresor.java

```
public class ChasseAuTresor {
    public static void main(String[] args) {
        // créer deux robots
        SpecRobot nono = new Robot(-3, 7, Direction.Sud);
        SpecRobot astro = new RobotMasque(10, 10, Direction.Nord);
        // ...

        // déterminer quel est le robot le plus proche du trésor
        SpecRobot leMeilleur =
            lePlusProche(nono, astro, xTresor, yTresor);

    } // fin main

    // déterminer quel est le robot le plus proche du trésor
    static SpecRobot lePlusProche(SpecRobot celuiCi, SpecRobot lautre,
        int xtresor, int ytresor)
    {
        if ( carreDistance(celuiCi.getX(), celuiCi.getY(), xtresor, ytresor) <
            carreDistance(lautre.getX(), lautre.getY(), xtresor, ytresor))
        {
            return celuiCi;
        }
        else {
            return lautre;
        }
    }
} // fin ChasseAuTresor
```

les références  
sont du type de  
l'interface

45

## Référence, instance : opérations

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot astro = new RobotMasque(10, 10,
    Direction.Nord)
```

nono

astro

### • Affectation : opérateur =

```
nono = astro;
```

nono

astro

Pas d'affectation entre instances,  
uniquement entre références

????

46

## Référence, instance : opérations

```
SpecRobot nono =
    new Robot(3, -7, Direction.Sud);
SpecRobot astro =
    new Robot(3, -7, Direction.Sud);
```

nono

astro

### • Comparaison : opérateur ==

```
nono == astro
```

vrai uniquement si nono et astro  
désignent la même instance

Pas de comparaison entre  
instances,  
uniquement entre références

47

## un Robot « java-compatible »

### • Égalité de deux instances : equals

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot astro = new RobotMasque(3, -7, Direction.Sud);
boolean egaux = nono.equals(astro);
```

SpecRobot.java

```
public interface SpecRobot {
    // ...
    // égalité de deux robots
    public boolean equals(SpecRobot autre);
}
```

Robot.java

```
public class Robot implements SpecRobot {
    // ...
    // égalité de deux robots
    public boolean equals(SpecRobot autre) {
        return
            this.getX()      == autre.getX()      &&
            this.getY()      == autre.getY()      &&
            this.getDirection() == autre.getDirection();
    }
    // ...
} // fin classe Robot
```

48

## un Robot « java-compatible »

- Égalité de deux instances : **equals**

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot astro = new RobotMasque(3, -7, Direction.Sud);
boolean egaux = nono.equals(astro);
```

```
public interface SpecRobot {
    // ...
    // égalité de deux robots
    public boolean equals(SpecRobot r);
}
```

**Attention**

- si **equals** n'est pas programmé
- pas d'erreur de compilation
- comportement identique à ==
- comparaison de références

```
public class Robot implements SpecRobot {
    // ...
    this.getX()
    this.getY()
    this.getDirection()
    // ...
} // fin classe Robot
```

49

## un Robot « java-compatible »

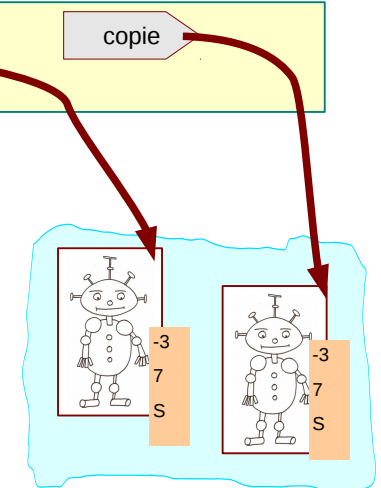
- Copie d'instance : **clone**

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot copie = nono.clone();
```

```
public interface SpecRobot {
    // ...
    // créer une copie de robot
    public SpecRobot clone();
}
```

```
public class Robot implements SpecRobot {
    // ...
    // créer une copie de robot
    public SpecRobot clone() {
        return new Robot(this.getX(), this.getY(),
                          this.getDirection());
    }
    // ...
} // fin classe Robot
```

50



## un Robot « java-compatible »

- Copie d'instance : **clone**

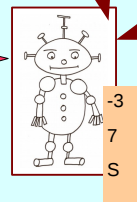
```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot copie = nono.clone();
```

**Attention**

- si **clone** n'est pas programmé
- pas d'erreur de compilation
- clone renvoie this
- copie de référence

```
public class Robot implements SpecRobot {
    // ...
    // ...
} // fin classe Robot
```

51



## un Robot « java-compatible »

- Représentation « affichable » d'une instance : **toString**

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
System.out.print("Voici nono : " + nono);
```

```
System.out.print("Voici nono : " + nono.toString());
```

```
public class Robot implements SpecRobot {
    // ...
    // représentation affichable d'un robot
    public String toString() {
        return "(" + this.getX() + "," + this.getY() + "," + this.getDirection() + ")";
    }
    // ...
} // fin classe Robot
```

52

Voici nono : (3, -7, Sud)

## un Robot « java-compatible »

- Représentation « affichable » d'une instance : **toString**

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
System.out.print("Voici nono : " + nono);
```



```
System.out.print("Voici nono : " + nono.toString());
```

Robot.java

```
public class Robot {
    // ...
    //représ
    public
    retour
    }
    // ...
} // fin classe
```

**Attention**  
si **toString** n'est pas programmé  
pas d'erreur de compilation  
affichage de la référence

Voici nono : Robot@1fc4bec

53

## Tableau, référence, instance

// Jouer avec des robots dans un tableau

// 1. Créer le tableau

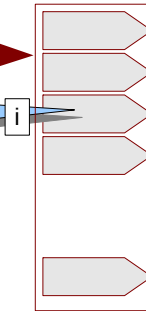
```
SpecRobot [ ] lesRobots = new SpecRobot[1000];
```

**lesRobots** : référence de tableau

ne crée pas une instance de SpecRobot, mais un tableau

lesRobots

**lesRobots[i]** : référence de type **SpecRobot** (non initialisée)  
peut désigner une instance de **Robot** ou de **RobotMasque**



54

## Tableau, référence, instance

// Jouer avec des robots dans un tableau

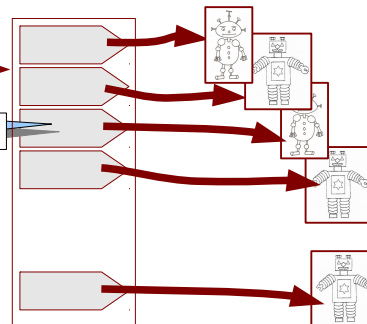
// 2. Ajouter les éléments

```
for (int i = 0 ; i < lesRobots.length; i=i+2) {
    lesRobots[i] = new Robot(..., ..., ...);
    lesRobots[i+1] = new RobotMasque(..., ..., ...);
}
```

**lesRobots** : référence de tableau

lesRobots

**lesRobots[i]** : référence de type **SpecRobot** (initialisée)  
on peut lui appliquer les opérations de la spécification



55

## Tableau, référence, instance

// Jouer avec des robots dans un tableau

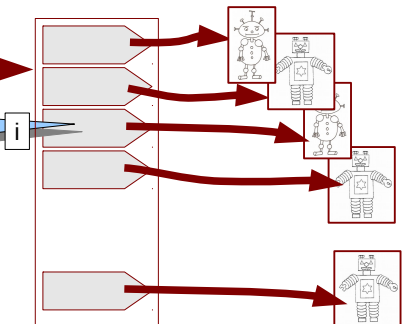
// 3. Agir sur un robot

```
for (int i = 0 ; i < lesRobots.length; i=i+1) {
    lesRobots[i].avancer();
    lesRobots[i].tourner();
}
...
```

**lesRobots** : référence de tableau

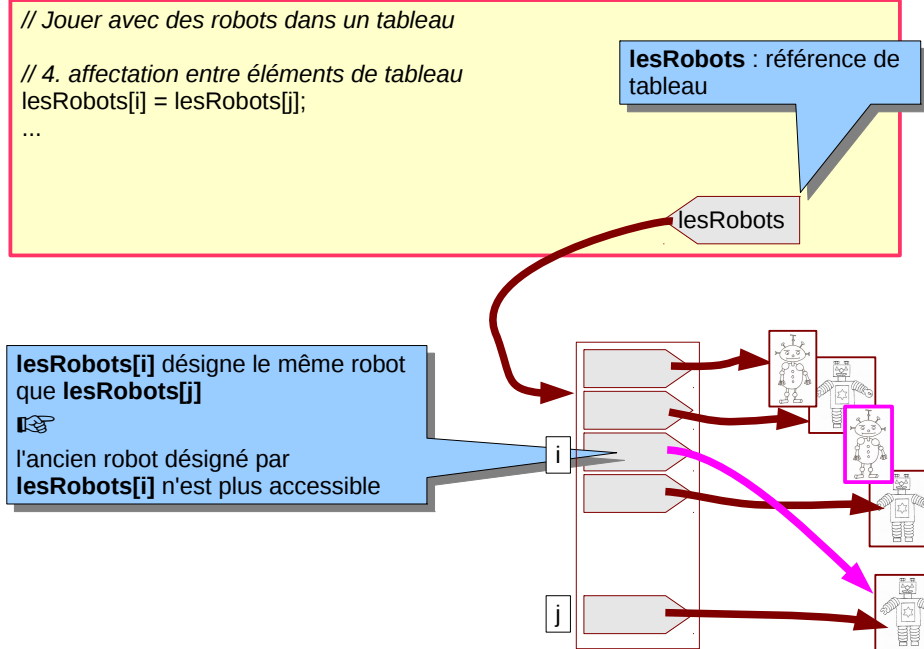
lesRobots

**lesRobots[i]** : référence de type **SpecRobot** (initialisée)  
on peut lui appliquer les opérations de la spécification



56

## Tableau, référence, instance



57

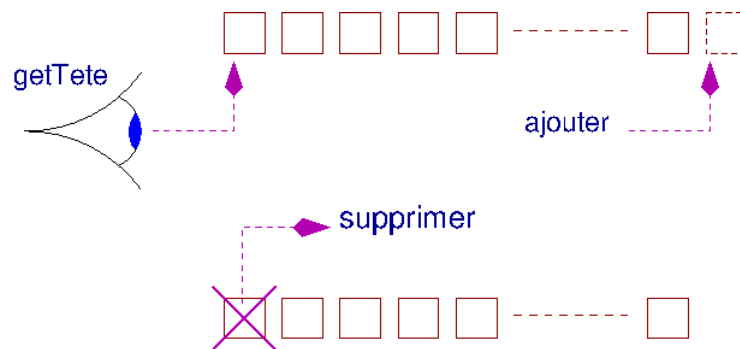
## Plan

- Introduction
- Type abstrait
- **Généricité**
- Structures de données
- Diagramme de classes
- Test
- Héritage et polymorphisme
- Modélisation

58

## le type abstrait File

- Modéliser le comportement d'une file d'attente



59

## spécification du type File

**File.java**

```
public interface File {

    // Modéliser le comportement d'une file d'attente de réels

    // constructeur : initialiser une file vide
    // public File();           // capacité non bornée
    // public File(int capacite); // capacité bornée

    // Opérations de consultation
    public boolean  estVide();    // vrai si la file est vide
    public boolean  estPleine(); // vrai si la file est pleine
    public int      getTaille(); // nombre d'éléments dans la file
                                // 0 ≤ taille ≤ capacité
    public double   getTete();   // @pre : file non vide

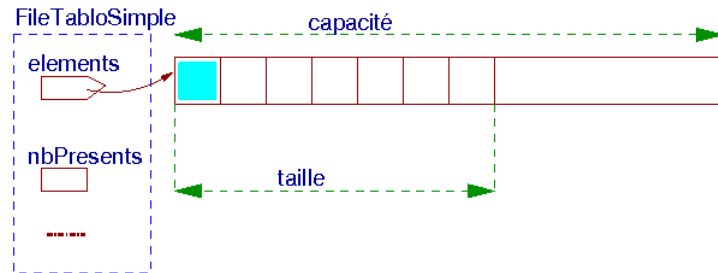
    // Opérations de modification
    public void ajouter(double x); // @pre : file non pleine
    public void supprimer();       // @pre : file non vide

} // fin interface
```

60



## implémentation du type File



61

## implémentation du type File

FileTabloSimple.java

```
public class FileTabloSimple implements File {

    // attributs : les éléments sont placés dans un tableau
    private double [ ] elements; // (référence du) tableau des éléments
    private int nbPresentes; // nombre d'éléments dans la file
    private ... // etc ...

    // constructeur : initialiser une file vide
    public FileTabloSimple(int capacite) {
        elements = new double[capacite]; // créer le tableau
        nbPresentes = 0; // file vide
        // etc ...
    }

    // Opérations de consultation
    public boolean estVide() { return nbPresentes == 0; }
    public double getTete() { return elements[0]; }
    // etc ...

    // Opérations de modification
    public void ajouter(double x) { elements[nbPresentes] = x ; ++ nbPresentes; }
    public void supprimer() { ... }
} // fin classe
```

62

## utilisation du type File

ClientFile.java

```
public class ClientFile {
    public static void main(String [ ] aaaaaargs) {

        // créer une file vide
        File maFile = new FileTabloSimple(130);

        // ajouter des éléments
        maFile.ajouter(3.98798);
        maFile.ajouter(-6.656464);
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer(); // passer au suivant
        }
    } // fin main
} // fin classe
```

63

## Généricité de type

- Et si on veut une file de chaînes (type String) ?
- ☞ il faut reprogrammer la classe File
  - en remplaçant double par String
- ☞ travail redondant
- ☞ à recommencer si on modifie l'implémentation de File

64

## spécification du type File

FileString.java

```
public interface FileString {

    // Modéliser le comportement d'une file d'attente de chaînes

    // constructeur : initialiser une file vide
    // public File();           // capacité non bornée
    // public File(int capacite); // capacité bornée

    // Opérations de consultation
    public boolean  estVide();           // vrai si la file est vide
    public boolean  estPleine();        // vrai si la file est pleine
    public int      getTaille();         // nombre d'éléments dans la file
                                           // 0 ≤ taille ≤ capacité
    public String   getTete();           // @pre : file non vide

    // Opérations de modification
    public void ajouter(String x);       // @pre : file non pleine
    public void supprimer();             // @pre : file non vide

} // fin interface
```

65

## implémentation du type File

FileTabloSimpleString.java

```
public class FileTabloSimpleString implements FileString {

    // attributs : les éléments sont placés dans un tableau
    private String [ ] elements; // (référence du) tableau des éléments
    private int nbPresentes;      // nombre d'éléments dans la file
    private ...                  // etc ...

    // constructeur : initialiser une file vide
    public FileTabloSimple(int capacite) {
        elements = new String[capacite]; // créer le tableau
        nbPresentes = 0;                 // file vide
        // etc ...
    }

    // Opérations de consultation
    public boolean  estVide() { return nbPresentes == 0; }
    public String   getTete() { return elements[0]; }
    // etc ...

    // Opérations de modification
    public void ajouter(String x) { elements[nbPresentes] = x ; ++ nbPresentes; }
    public void supprimer()      { ... }

} // fin classe
```

66

## utilisation du type File

ClientFileString.java

```
public class ClientFileString {
    public static void main(String [ ] aaaaaargs) {

        // créer une file vide
        FileString maFile = new FileTabloSimpleString(130);

        // ajouter des éléments
        maFile.ajouter("3.98798");
        maFile.ajouter("Pamplemousse");
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer();                   // passer au suivant
        }
    } // fin main
} // fin classe
```

67

## Généricité de type

- Et si on veut une file de robots ?
- ☞ il faut reprogrammer la classe File
  - en remplaçant double par SpecRobot
- ☞ travail redondant
- ☞ à recommencer si on modifie l'implémentation de File

68

## spécification du type File

FileRobot.java

```
public interface FileRobot {

    // Modéliser le comportement d'une file d'attente de chaînes

    // constructeur : initialiser une file vide
    // public File();           // capacité non bornée
    // public File(int capacite); // capacité bornée

    // Opérations de consultation
    public boolean    estVide();           // vrai si la file est vide
    public boolean    estPleine();        // vrai si la file est pleine
    public int        getTaille();        // nombre d'éléments dans la file
                                           // 0 ≤ taille ≤ capacité
    public SpecRobot  getTete();           // @pre : file non vide

    // Opérations de modification
    public void ajouter(SpecRobot x);     // @pre : file non pleine
    public void supprimer();              // @pre : file non vide

} // fin interface
```

69

## implémentation du type File

FileTabloSimpleRobot.java

```
public class FileTabloSimpleRobot implements FileRobot {

    // attributs : les éléments sont placés dans un tableau
    private SpecRobot [] elements; // (référence du) tableau des éléments
    private int nbPresentes;        // nombre d'éléments dans la file
    private ...                    // etc ...

    // constructeur : initialiser une file vide
    public FileTabloSimple(int capacite) {
        elements = new SpecRobot[capacite]; // créer le tableau
        nbPresentes = 0;                    // file vide
        // etc ...
    }

    // Opérations de consultation
    public boolean    estVide()    { return nbPresentes == 0; }
    public SpecRobot  getTete()    { return elements[0]; }
    // etc ...

    // Opérations de modification
    public void ajouter(SpecRobot x) { elements[nbPresentes] = x ; ++ nbPresentes; }
    public void supprimer()          { ... }

} // fin classe
```

70

## utilisation du type File

ClientFileRobot.java

```
public class ClientFileRobot {
    public static void main(String [] aaaaaargs) {

        // créer une file vide
        FileRobot maFile = new FileTabloSimpleRobot(130);

        // ajouter des éléments
        maFile.ajouter(new Robot(-3, 7, Direction.sud));
        maFile.ajouter(new RobotMasque(10, 10, Direction.Nord));
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer();                   // passer au suivant
        }
    } // fin main
} // fin classe
```

71

## Généricité de type

- Et si on veut une file de chaînes (type String) ?
- Et si on veut une file de robots ?
- ☞ il faut reprogrammer la classe File
  - en remplaçant double par String
  - en remplaçant double par SpecRobot
- ☞ travail inutile, redondant
- ☞ à recommencer si on modifie l'implémentation de File
- ☞ « la » solution : généricité de type
  - paramétrer la spécification et l'implémentation par le type des éléments de la file
  - définir le type effectif des éléments dans le programme « client », lors de la création de la file.

72

## spécification du type générique File

File.java

```
public interface File<T> {
    // T est le type des éléments

    // constructeur : initialiser une file vide
    // public File();           // capacité non bornée
    // public File(int capacite); // capacité bornée

    // Opérations de consultation
    public boolean  estVide();           // vrai si la file est vide
    public boolean  estPleine();        // vrai si la file est pleine
    public int      getTaille();         // nombre d'éléments dans la file
                                           // 0 ≤ taille ≤ capacité
    public T        getTete();           // @pre : file non vide

    // Opérations de modification
    public void ajouter(T x) ;           // @pre : file non pleine
    public void supprimer();             // @pre : file non vide
} // fin interface
```

73

## implémentation du type File

FileTabloSimple.java

```
public class FileTabloSimple<T> implements File<T> {
    // T est le type des éléments

    // attributs : les éléments sont placés dans un tableau
    private ...;           // à voir en TD
    private int nb Presents; // nombre d'éléments dans la file
    private ...           // etc ...

    // constructeur : initialiser une file vide
    public FileTabloSimple(int capacite) {
        elements = new ...; // créer le tableau
        nb Presents = 0;    // file vide
        // etc ...
    }

    // Opérations de consultation
    public boolean  estVide() { return nb Presents == 0; }
    public T        getTete(){ return ...; }
    // etc ...

    // Opérations de modification
    public void ajouter(T x) { ... ; ++ nb Presents; }
    public void supprimer() { ... }
} // fin classe
```

74

## utilisation du type File

ClientFile.java

```
public class ClientFile {
    public static void main(String [] aaaaaargs) {

        // créer une file vide
        File<String> maFile = new FileTabloSimple<String>(130);

        // ajouter des éléments
        maFile.ajouter("Le chat");
        maFile.ajouter("machine");
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer();                  // passer au suivant
        }
    } // fin main
} // fin classe
```

75

## utilisation du type File

ClientFile.java

```
public class ClientFile {
    public static void main(String [] aaaaaargs) {

        // créer une file vide
        File<SpecRobot> maFile = new FileTabloSimple<SpecRobot>(130);

        // ajouter des éléments
        maFile.ajouter(new Robot(10, 20, Direction.Nord));
        maFile.ajouter(new RobotMasque(-5, 13, Direction.Est));
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer();                  // passer au suivant
        }
    } // fin main
} // fin classe
```

76

## généricité et types simples

ClientFile.java

```
public class ClientFile {
    public static void main(String [] aaaaaargs) {
        // créer une file vide
        File<double> maFile = new FileTabloSimple<double>(130);
        // ...
    } // fin classe
```

Impossible en java

En java, les types paramètres d'une classe générique doivent être désignés par référence

type scalaire	type encapsulant
int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

77

## types scalaires et types « encapsulants »

// syntaxe « normale »

Integer nb1 = new Integer(23);

// **autoboxing**

Integer nb2 = 23;

// autres possibilités

int nb3 = -5;

Integer nb4 = nb3;

// **auto-deboxing**

int nb5 = nb2 + nb4;

java crée l'instance de Integer et l'initialise avec 23

java crée l'instance de Integer et l'initialise avec nb3

nb3 -5

nb5 18

Les types qui encapsulent les types scalaires possèdent :

- equals
- toString
- clone

78

## Généricité et types simples

ClientFile.java

```
public class ClientFile {
    public static void main(String [] aaaaaargs) {
        // ...
        File<Double> maFile = new FileTabloSimple<Double>(130);

        // ajouter des éléments
        maFile.ajouter(3.14159); // ⇔ maFile.ajouter(new Double(3.14159))
        maFile.ajouter(2.71828); // ⇔ maFile.ajouter(new Double(2.71828))
        // etc ...

        // traiter le contenu de la file
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer(); // passer au suivant
        }
    } // fin main
} // fin classe
```

79

## Fonction générique

ClientFile.java

```
public class ClientFile {
    // afficher en la vidant une file générique
    static <T> void afficherVider(File<T> uneFile) {
        while (! maFile.estVide()) {
            System.out.println(maFile.getTete()); // traiter l'élément de tête
            maFile.supprimer(); // passer au suivant
        }
    } // fin afficherVider
}
```

```
public static void main(String [] aaaaaargs) {
    // créer une file vide
```

```
File<Double> fileReels = new FileTabloSimple<Double>(130);
// ajouter des éléments
// ...
// afficher la file
afficherVider(fileReels);
```

```
// créer une file vide
```

```
File<SpecRobot> fileRobots = new FileTabloSimple<SpecRobot>(130);
// ajouter des éléments
// ...
// afficher la file
afficherVider(fileRobots);
// ...
```

80

## Interface générique utile

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

- comparaison ordonnée d'instances
- soient v1 et v2 des (références d')instances d'une classe qui implémente Comparable<T>
  - $v1.compareTo(v2) < 0$  si  $v1 < v2$
  - $v1.compareTo(v2) = 0$  si  $v1 = v2$
  - $v1.compareTo(v2) > 0$  si  $v1 > v2$
- ex : Integer, Double, ..., String implémentent cette interface.

```
String v1 = "bonjour", v2 = "bonsoir";  
v1.compareTo(v2) < 0  
v2.compareTo(v1) > 0  
v1.compareTo(v1) = 0
```

81

## Interface Comparable<T>

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

- utilisation
  - compléter les fonctionnalités d'une classe
- intérêt
  - avoir une méthode commune de comparaison  $\Rightarrow$  fonctions génériques
- exemple :
  - classe Robot qui implémente les fonctionnalités :
    - de l'interface SpecRobot : fonctionnalités principales
    - de l'interface Comparable<T> : fonctionnalité secondaire
  - application : classer des robots
    - tableau de robots classés par distance (dé)croissante à l'origine

82

### un Robot comparable (v1)

Robot.java

```
public class Robot implements SpecRobot, Comparable<Robot> {  
  
    // implémentation de l'interface SpecRobot  
  
    // implémentation de l'interface Comparable  
    public int compareTo(Robot autre) {  
        return Math.abs(this.getX() - autre.getX()) +  
               Math.abs(this.getY() - autre.getY());  
    }  
} // fin classe Robot
```

contenu identique à la classe Robot présentée plus haut


client.java

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);  
SpecRobot astro = new Robot(10, 10, Direction.Nord);  
int comparaison = nono.compareTo(astro); // < 0  $\Rightarrow$  nono est < astro
```

83

### un Robot comparable (v1)

- limites de cette première version :
    - ne permet pas de comparer des Robot et des RobotMasque
- client.java

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);  
SpecRobot astro = new RobotMasque(10, 10, Direction.Nord);  
  
nono.compareTo(astro);  
astro.compareTo(nono);
```
- ne garantit pas que toutes les implémentations de SpecRobot seront comparables
  - amélioration
    - imposer / garantir que toutes les implémentations de SpecRobot seront comparables
    - permettre la comparaison des instances de toutes les implémentations
  -  exiger que l'interface SpecRobot « possède » compareTo

84

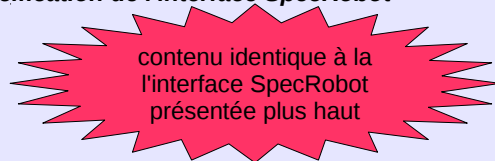
## un Robot comparable (v2)

SpecRobot.java

```
public interface SpecRobot extends Comparable<SpecRobot> {

    // spécification de l'interface SpecRobot

    // spécification de l'interface Comparable
    public int compareTo(SpecRobot autre);
} // fin interface SpecRobot
```



- extends exprime la relation entre deux interfaces
  - l'interface SpecRobot étend les fonctionnalités de Comparable<T> en y ajoutant les siennes propres

85

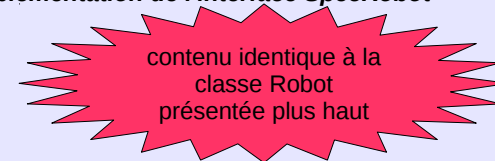
## un Robot comparable (v2)

Robot.java

```
public class Robot implements SpecRobot {

    // implémentation de l'interface SpecRobot

    // implémentation de l'interface Comparable
    public int compareTo(SpecRobot autre) {
        return Math.abs(this.getX()) - Math.abs(autre.getX()) +
               Math.abs(this.getY()) - Math.abs(autre.getY());
    }
} // fin classe Robot
```



client.java

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot astro = new Robot(10, 10, Direction.Nord);
int comparaison = nono.compareTo(astro); // < 0 ⇒ nono est < astro
```

86

## un Robot comparable (v2)

- intérêt de cette deuxième version :
  - impose / garantit que toutes les implémentations de SpecRobot seront comparables
  - permet de comparer des Robot et des RobotMasque

client.java

```
SpecRobot nono = new Robot(3, -7, Direction.Sud);
SpecRobot astro = new RobotMasque(10, 10, Direction.Nord);
```

```
nono.compareTo(astro);
```

```
astro.compareTo(nono);
```

87