Algorithmique et Programmation

Jean-Christophe Engel

École supérieure d'ingénieurs de Rennes
Université de Rennes 1

Plan

5.1 Réutilisation

Type abstrait

- utiliser un type dans différents contextes d'applications

Généricité

- généraliser un type pour ne pas le reprogrammer

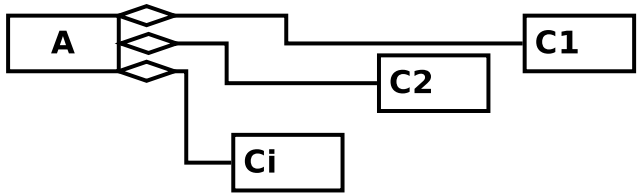
Collaboration par composition / agrégation / délégation

- utiliser un type existant au service d'un autre

Héritage

- réutiliser un type existant pour :
  - bénéficier de certaines fonctionnalités inchangées
  - modifier certaines fonctionnalités
  - ajouter des fonctionnalités

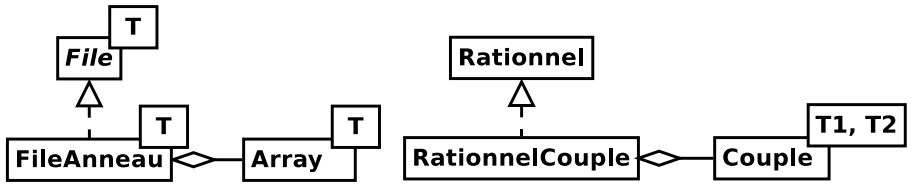
Composition / Agrégation / Délégation



Principe

- la classe A *délègue* le stockage et le traitement de certaines valeurs à une ou plusieurs classes (C1, C2, ..., Ci)
- la classe A *utilise* les services des classes Ci
- Pas de relation privilégiée entre les différentes classes
  - A est client des Ci
  - A utilise les fonctionnalités *publiques* des classes Ci

Composition / Agrégation / Délégation



Exemples

- la classe *FileAnneau* délègue à la classe *Array* le stockage des éléments et utilise ses services (méthodes)
- La classe *FileAnneau* ne connaît pas les attributs interne de la classe *Array*, mais se sert de ses méthodes publiques :
  - constructeur, get(), set(), length()
- la classe *RationnelCouple* délègue à la classe *Couple* le stockage du numérateur et du dénominateur

Composition / Agrégation / Délégation

Réalisation

- Dans A on trouve des attributs de type (référence de) C1, C2, ... Ci
- A peut créer des instances d'une ou plusieurs des classes Ci ou désigner des instances créées à l'extérieur

Exemple de la classe FileAnneau

```
public class FileAnneau<T> implements File<T>
{
  private Array<T> tElements; // tableau des éléments
  // initialiser une file
  public FileAnneau(int capacite) {
    tElements = new Array<T>(capacite); ...
  }
  // ajouter un élément en queue de file
  public void ajouter(T x) { ... tElements.set(..., x); ... }
...
}
```

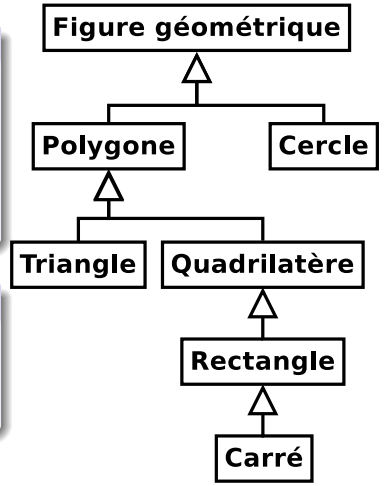

5.2 Héritage

Intérêt

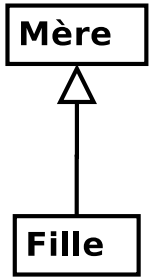
- réutiliser un type existant pour :
  - bénéficier de certaines fonctionnalités inchangées
  - modifier certaines fonctionnalités
  - ajouter des fonctionnalités

Principe

- relation hiérarchique « est-un » entre une classe fille et une classe mère



Principe de l'héritage



Principe

- relation hiérarchique « est-un » entre une classe fille et une classe mère
- la classe fille :
  - *dispose* des méthodes et attributs de classe mère
  - peut *redéfinir* certaines méthodes
  - peut *ajouter* des méthodes et des attributs

Conséquence

- toute méthode publique définie dans la classe mère peut s'appliquer sur une instance de la classe fille
- toute instance d'une classe fille peut s'utiliser à la place d'une instance d'une classe mère

Exemple du Robot

```java
package robotSimple.v3;

//Modéliser le comportement d'un Robot

public interface SpecRobot {

  public void     avancer();
  public void     tourner();

  public boolean   memePosition(SpecRobot autre);
  public boolean   equals(SpecRobot autre);
  public String    toString();
  public SpecRobot clone();

  public int       getX();
  public int       getY();
  public Direction getDirection();

} // SpecRobot
```


Exemple du Robot

```java
// Implémentation simple d'un robot
public class Robot implements SpecRobot {

  // attributs
  private int X, Y;
  private Direction direction;

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

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

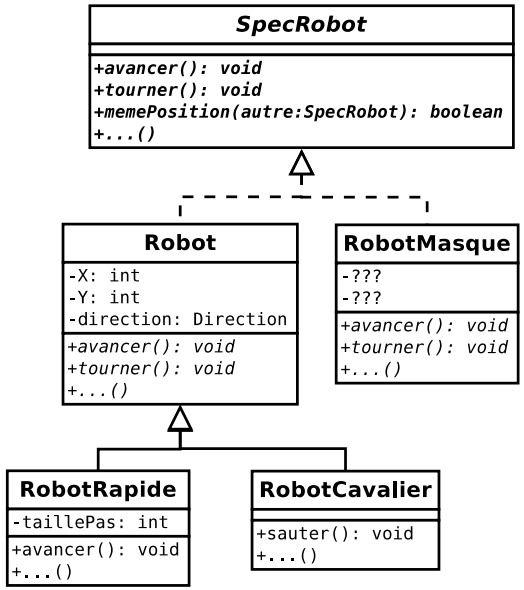

Exemple du Robot

```java
// Faire tourner le robot (this) d'un 1/4 de tour "à droite"
public void tourner() {
  switch (this.getDirection()) {
  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;
  }
}

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

// accesseurs
public int getX() { return this.X; }
public int getY() { return this.Y; }
public Direction getDirection() { return this.direction; }
```


Exemple du Robot

```java
// Déterminer si le robot this et le robot autre ont même valeur
public boolean equals(SpecRobot autre) {
  return
    this.getX()         == autre.getX() &&
    this.getY()         == autre.getY() &&
    this.getDirection() == autre.getDirection();
}

// fournir une représentation affichable d'un robot
public String toString() {
  return "(" + this.getX() + "," + this.getY() + "," + this.getDirect
}

// renvoyer une copie du robot this
public SpecRobot clone() {
  return new Robot(this.getX(), this.getY(), this.getDirection());
}
// Robot
```

De nouveaux robots



Programmer une classe fille

Que programmer dans une classe fille ?

- relation entre classe fille et classe mère : ***extends***
- *nouveaux* attributs
- constructeurs
- méthodes *redéfinies*
- *nouvelles* méthodes

Règle

- on programme *ce qui change*
- on ne programme *pas* ce qui ne change *pas*

RobotRapide

```
public class RobotRapide extends Robot {

  // attribut propre
  private int taillePas;

  // constructeur
  public RobotRapide(int x, int y, int pas) {
    // 1) initialiser attributs mère ⇒ appel constructeur mère
    super(x, y, Direction.Sud);
    // 2) initaliser attribut propre
    this.taillePas = pas;
  }

  // méthodes redéfinies
  public void avancer() {
    for (int i = 0; i < taillePas; ++i) {
      // pas d'accès aux attributs X et Y de la classe Robot
      super.avancer();
    }
  }
```


RobotRapide

```
// méthodes redéfinies
public String toString() {
  return "(" + this.getX() + "," + this.getY() + ","
    + this.taillePas + "," + this.getDirection() + ")";
}

public SpecRobot clone() {
  return
    new RobotRapide(this.getX(), this.getY(), this.taillePas);
}

// autres méthodes inchangées ⇒ inutile de les redéfinir
// public void      tourner();
// public boolean   memePosition(SpecRobot autre);
// public boolean   equals(SpecRobot autre);
// public int       getX();
// public int       getY();
// public Direction getDirection();
```

RobotCavalier

```
public class RobotCavalier extends Robot {

  public RobotCavalier(int x, int y, Direction dir) {
    // initialiser attributs mère ⇒ appel constructeur mère
    super(x, y, dir);
  }

  // nouvelle méthode : sauter comme un cavalier
  public void sauter() {
    super.avancer();
    super.avancer();
    super.tourner();
    super.avancer();
  }

  // méthode redéfinie
  public SpecRobot clone() {
    return new RobotCavalier(getX(), getY(), getDirection());
  }
}
```


Programmer une classe fille

Que programmer dans une classe fille ?

- relation entre classe fille et classe mère : ***extends***
- *nouveaux* attributs
- constructeurs
  1. initialisation attributs classe mère ⇒ appel d'un constructeur de la classe mère : **super**(paramètres)
  2. initialisation attributs propres
- méthodes *redéfinies* : même signature que la méthode mère ; appel méthode classe mère : **super**.méthode(paramètres)
- *nouvelles* méthodes

Classe mère

Empêcher la redéfinition d'une méthode : **final**

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

La méthode memePosition ne peut être redéfinie dans les classes filles

Contrôle d'accès

Accès aux attributs et méthodes d'une classe

1. public
   - accès depuis toute méthode ou fonction de toute classe
   - ⇒ méthodes de la spécification, déclarées dans une interface
2. private
   - accès réservé aux méthodes programmées dans la même classe
   - inaccessibles depuis classes filles
   - ⇒ attributs, méthodes d'implémentation
3. protected
   - accès réservé aux méthodes programmées dans la même classe ainsi qu'aux méthodes des classes filles
   - ⇒ *attributs*, méthodes d'implémentation qui peuvent servir dans les classes filles

Contrôle d'accès

Recommandation

public

- méthodes de la spécification, déclarées dans une interface
- nouvelles méthodes introduites dans une cllasse fille

private

- attributs, méthodes d'implémentation internes à la classe

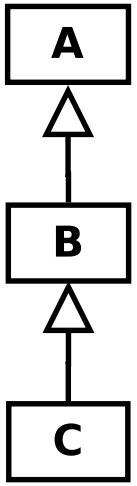
protected

- méthodes d'implémentation qui peuvent servir dans les classes filles

Hiérarchie de classes

l'héritage est transitif

- toute instance d'une classe peut s'utiliser à la place d'une instance d'une classe située **plus haut** dans la hiérarchie
- une référence d'un type peut désigner une instance du même type ou de tout type situé **plus bas** dans la hiérarchie



5.3 Hiérarchie de classes

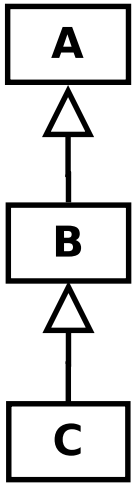
l'héritage est transitif

- un C « est-un » B qui « est-un » A
  $\Rightarrow$ un C « est-un » A
- une instance de C dispose de tout ce qui est défini dans A et B
- une instance de C peut s'utiliser à la place d'une instance de A ou de B

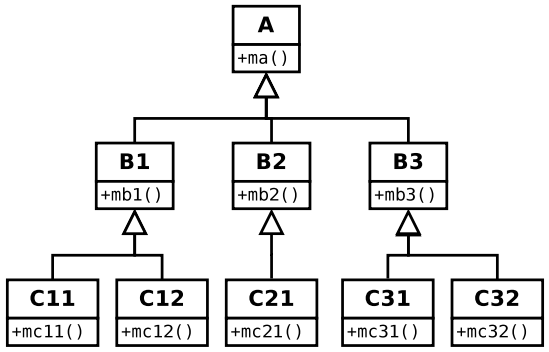
Initialisation d'une instance

une instance de C est initialisée « de haut en bas »

- d'abord les attributs de A
- puis ceux de B
- puis ceux de C



Hiérarchie de classes



- Une classe peut avoir plusieurs classes filles
- en java, une classe ne peut avoir qu'une classe mère (héritage simple)
- en C++, une classe peut avoir plusieurs classes mères (héritage multiple, complexe)

Hiérarchie de classes

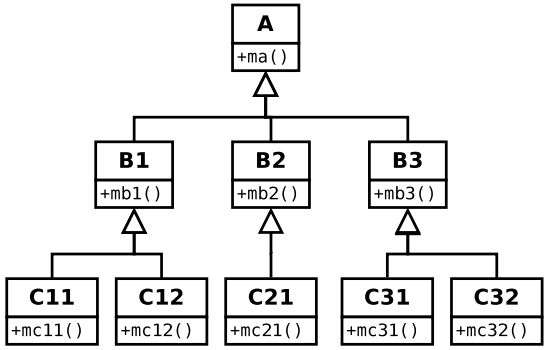
Règle de compatibilité

- Dans une hiérarchie de classes, une instance d'une classe peut :
  - être utilisée à la place de toute instance d'un type situé **plus haut** dans la hiérarchie
  - être désignée par une référence de tout type situé **plus haut** dans la hiérarchie

Corollaire

- Une référence d'un type peut être initialisée avec une référence de tout type situé **plus bas** dans la hiérarchie
- Une référence d'un type peut désigner une instance du même type ou de tout type situé **plus bas** dans la hiérarchie

Hérarchie de classe



- une référence du type de A peut désigner une instance de A, de Bi, de Cij
- une référence du type de B1 peut désigner une instance de B1, de C1j
- une référence du type de C11 peut désigner une instance de C11

Appel de méthode : référence.méthode(paramètres)

à la compilation

- la méthode doit être présente dans la classe de la **référence** directement ou par héritage
  ⇒ une méthode de la classe de l'instance qui n'existe pas dans la classe de la référence n'est pas vue par le compilateur
  ⇒ **vérification statique**

à l'exécution

- la méthode est cherchée dans la classe de **l'instance**
  ⇒ **liaison dynamique**
- si la méthode n'est pas trouvée, elle est cherchée dans la classe mère, etc... en remontant dans la hiérarchie
  ⇒ **héritage**

Cas des interfaces

si une classe d'une hiérarchie implémente une interface

- une référence du type de l'interface peut désigner une instance de toute classe qui implémente l'interface directement ou par *transitivité*

Appel de méthode

- le compilateur ne voit que les méthodes déclarées dans l'interface
  ⇒ **vérification statique**
- le choix de la méthode appelée est fait à l'exécution selon la classe de l'instance
  ⇒ **liaison dynamique**

Application : le polymorphisme

- mélanger des instances d'une hiérarchie de classes dans une structure de données ou un algorithme
- ces instances doivent être désignées avec une référence d'un type commun à toutes les instances (interface, si possible)
- seules les fonctionnalités communes des instances peuvent être utilisées : méthodes définies dans la classe ou l'interface de la référence

Polymorphisme : exemple

```
public class ChasseAuTresor {
static void deplacerRobots(Collection<SpecRobot> armee,
    int xCible, int yCible, int xBombe, int yBombe, int seuil) {
  // initialiser un parcours de la collection
  Iterator<SpecRobot> monIterateur = armee.iterator();
 // Appliquer un traitement à chaque robot
  while (monIterateur.hasNext()) {
    // parcours non terminé : traiter le robot courant
    SpecRobot courant = monIterateur.next();
    if (carreDistance(courant.getX(), courant.getY(),
          xBombe, yBombe) <= seuil) {
      monIterateur.remove(); // robot trop proche de la bombe
    } else if (courant.getDirection() != Direction.Ouest) {
        courant.tourner();
    } else if (carreDistance(courant.getX(), courant.getY(),
        xCible, yCible)) > 25) {
      courant.avancer();
    }
  } // while
} // deplacerRobots
```


Polymorphisme ou généricité

Généricité

- structure de données (List<E>) ou algorithme (tri) rendu indépendant par rapport à un ou plusieurs types
- appliquer des opérations qui ne dépendent pas du type
- *généricité contrainte* : exiger la disponibilité d'une opération (compareTo)

Polymorphisme

- utiliser dans une structure de données ou un algorithme des instances d'une hiérarchie de classes
- appliquer des opérations communes à toute la hiérarchie

deplacerRobots

- combine une structure de données générique (Collection<E>) et un algorithme polymorphe

5.4 Classe abstraite

But

- regrouper dans une classe :
  - des attributs communs à plusieurs classes d'une hiérarchie
  - des méthodes communes à plusieurs classes d'une hiérarchie
- les classes filles :
  - déclarent les attributs non communs
  - implémentent les méthodes qui diffèrent
  - peuvent redéfinir certaines méthodes si besoin

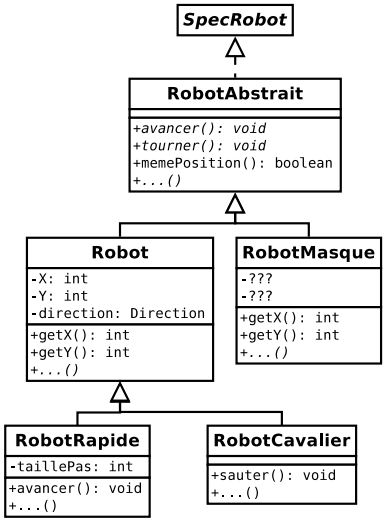
Exemple du robot

memePosition : indépendante de la représentation interne (getX, getY)

equals : on peut programmer une version générale qui peut être redéfinie si besoin

avancer, tourner : peuvent être rendues indépendantes de la représentation interne

Classe RobotAbstrait



Classe RobotAbstrait

```
public abstract class RobotAbstrait implements SpecRobot {

  // vrai si 2 robots sont sur la même case
  public final boolean memePosition(SpecRobot autre) {
    return this.getX() == autre.getX() &&
           this.getY() == autre.getY();
  }

  // vrai si 2 robots sont égaux
  public boolean equals(SpecRobot autre) {
    return
      this.getX()         == autre.getX() &&
      this.getY()         == autre.getY() &&
      this.getDirection() == autre.getDirection();
  }
```

- memePosition ne peut être redéfinie dans les classes filles
- equals peut être redéfinie dans les classes filles

Classe RobotAbstrait

Attention

- memePosition et equals dépendent de getX, getY, getDirection
- getX, getY, getDirection ne peuvent être programmées dans la classe RobotAbstrait

Conséquence

- getX, getY, getDirection sont déclarées « *abstract* » dans la classe RobotAbstrait
- elles doivent être programmées dans les classes filles
- la classe RobotAbstrait est une classe abstraite

Application

- on ne peut pas créer d'instance d'une classe abstraite

Classe RobotAbstrait

```
// Faire avancer le robot d'une unité dans sa direction
  public void avancer() {
    switch (getDirection()) {
    case Nord  : setY(getY() - 1); break;
    case Est   : setX(getX() + 1); break;
    case Sud   : setY(getY() + 1); break;
    case Ouest : setX(getX() - 1); break;
    }
  }

  // Faire tourner le robotd'un 1/4 de tour "à droite"
  public void tourner() {
    switch (getDirection()) {
    case Nord  : setDirection(Direction.Est);   break;
    case Est   : setDirection(Direction.Sud);   break;
    case Sud   : setDirection(Direction.Ouest); break;
    case Ouest : setDirection(Direction.Nord);  break;
    }
  }
```

Classe RobotAbstrait

Attention

- avancer et tourner dépendent de setX, setY, setDirection
- setX, setY, setDirection ne peuvent être programmées dans la classe RobotAbstrait
- setX, setY, setDirection ne sont pas dans la spécification (interface SpecRobot)

Conséquence

- setX, setY, setDirection sont déclarées « *abstract* » dans la classe RobotAbstrait
- elles **doivent** être programmées dans les classes filles
- elles ne peuvent être utilisées que dans la classe RobotAbstrait et ses classes filles ⇒ accessibilité **protected**

Classe RobotAbstrait

```
  // méthodes de la spécification à définir
  // dans les classes filles
  public abstract SpecRobot clone();
  public abstract int getX();
  public abstract int getY();
  public abstract Direction getDirection();

  // méthodes d'implémentation à définir
  // dans les classes filles
  protected abstract void setX(int nx);
  protected abstract void setY(int ny);
  protected abstract void setDirection(Direction nd);

} RobotAbstrait
```


Classe Robot

```
public class Robot extends RobotAbstrait {
  // attributs
  private int X, Y;
  private Direction direction;

  // initialiser un robot
  public Robot(int x, int y, Direction dir) {
    X = x; Y = y; direction = dir;
  }
  // public void avancer() : héritée
  // public void tourner() : héritée

  // public final boolean memePosition(SpecRobot autre)
  // héritée, ne peut être redéfinie

  // public boolean equals(SpecRobot autre) : héritée
  // public String toString() : héritée
```


Classe Robot

```
  // créer une copie de l'instance
  public SpecRobot clone() {
    return new Robot(this.getX(), this.getY(),
                     this.getDirection());
  }

  // autres méthodes de la spécification
  public int getX() { return this.X; }
  public int getY() { return this.Y; }
  public Direction getDirection() { return this.direction; }

  // méthodes d'implémentation
  protected void setX(int nx) { this.X = nx; }
  protected void setY(int ny) { this.Y = ny; }
  protected void setDirection(Direction nd) { direction = nd; }
} // Robot
```

Classes RobotRapide et RobotCavalier

```
public class RobotRapide extends Robot {

  // ...

} // RobotRapide
```

```
public class RobotCavalier extends Robot {

  // ...

} // RobotCavalier
```

Le contenu de ces classes ne change pas

Classe abstraite

Résumé

- classe qui regroupe
  - les attributs communs à plusieurs classes filles
  - les méthodes communes à plusieurs classes filles
- les méthodes programmées peuvent faire appel à des méthodes programmées dans les classes filles
- les méthodes qui ne peuvent être programmées sont déclarées abstraites ⇒ elles doivent être programmées dans les classes filles

Conséquence

- une classe qui contient une méthode abstraite est abstraite
- on ne peut pas en créer d'instance
- on peut déclarer des références du type d'une classe abstraite

interface ou classe abstraite ?

interface

- pas d'attribut
- toutes les méthodes sont abstraites (signature)
- pas de constructeur
- création d'instance impossible
- déclaration de référence possible

classe abstraite

- attributs
- constructeurs
- méthodes concrètes (corps)
- méthodes abstraites
- création d'instance impossible
- déclaration de référence possible

implémentation progressive

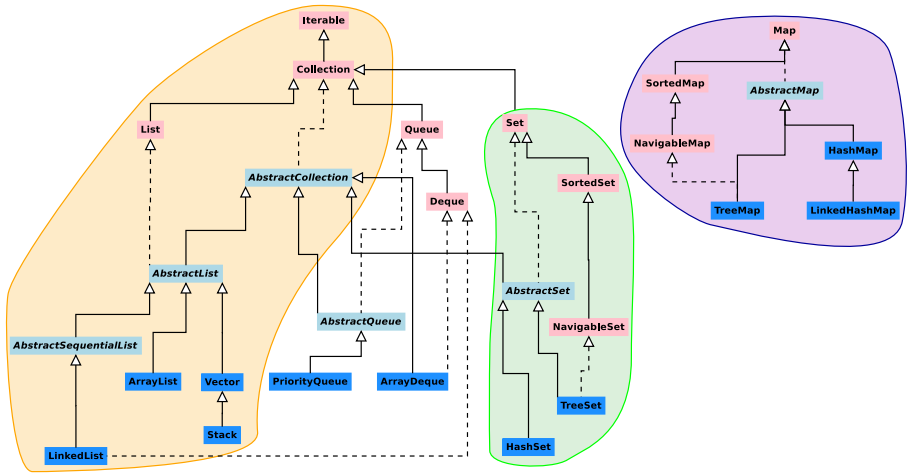
- Une classe qui hérite d'une classe abstraite peut ne pas implémenter toutes les méthodes abstraites
- elle reste abstraite

But

Programmer des implémentations progressives en déléguant certains choix d'implémentation aux classes filles

Exemple : bibliothèque de structures de données java

Bibliothèque java



Interfaces (le retour)

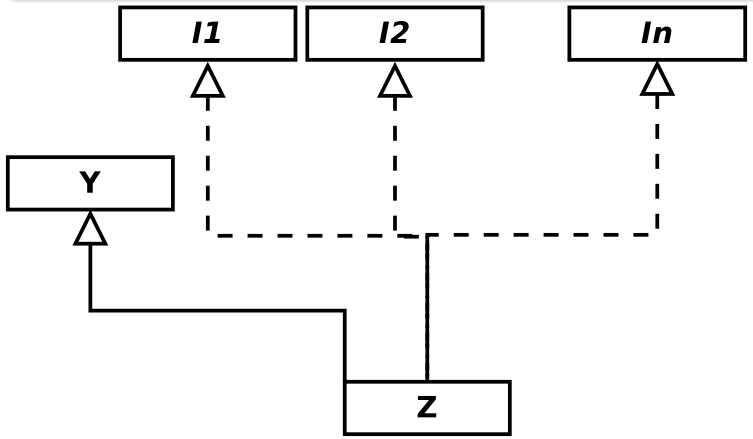
- Une classe (abstraite ou concrète) peut implémenter plusieurs interfaces
- Elle fournit donc les services de toutes les interfaces implémentées

```
public class RobotComparable implements SpecRobot,
                                        Comparable<SpecRobot>
{
  // représentation interne
  ..

  // méthodes de l'interface SpecRobot
  public void tourner() { ...}
  public void avancer() { ...}

  // méthode de l'interface Comparable<SpecRobot>
  public boolean compareTo(SpecRobot autre)
  {     ...     }
}
```


Interfaces multiples

Syntaxe

- public class X implements I1, I2, ..., In { ...}
- public class Z extends Y implements I1, I2, ..., In { ...}



Héritage d'interface

Syntaxe

public interface Y extends X { ... }

- l'interface Y étend les fonctionnalités de l'interface X
- On trouve dans Y toutes les fonctionnalités de X plus des opérations nouvelles (propres à Y)
- Toute classe qui implémente Y doit programmer
  - les opérations de X
  - les opérations propres de Y

Héritage d'interface : exemple

```
public interface SpecRobot extends Comparable<SpecRobot>
{
  // signatures des méthodes de l'interface SpecRobot
  public void tourner();
  // etc...
  // interface Comparable<SpecRobot>
  public boolean compareTo(SpecRobot autre);
}
```

```
public class RobotComparable implements SpecRobot
{
  // représentation interne
  ..

  // méthodes de l'interface SpecRobot
  public void tourner() { ... }
  // etc...
  // méthode de l'interface Comparable<SpecRobot>
  public boolean compareTo(SpecRobot autre) { ... }
}
```