ÉTS
Le génie pour l'industrie

LOG100 / GTI100
Programmation en génie logiciel et des TI
Automne 2024

Cours 4 :
Interfaces & Abstraction

Chargé de cours: Anes Abdennebi

Crédits à: Ali Ouni, PhD

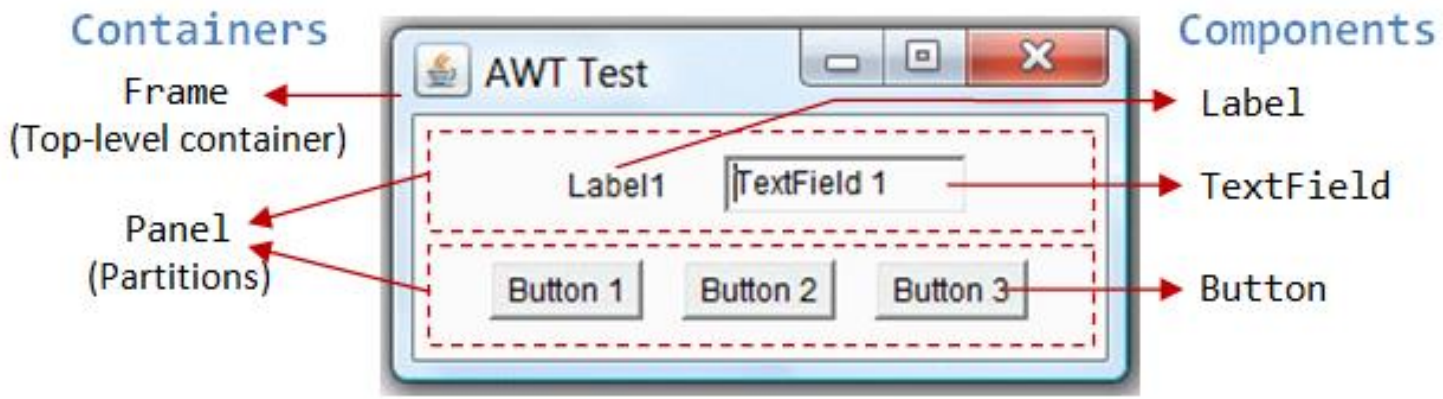
Plan

- Interfaces
- Polymorphisme
- Les interfaces Comparable et Comparator
- Classes anonymes
- Classe abstraite

Interfaces : exemple

- La classe *JOptionPane* facilite l'affichage de boîtes de dialogue de format standard pour :
  - informer ou avertir les utilisateurs
  - les inviter à entrer des valeurs
  - afficher des messages d'erreurs

Interfaces : exemple



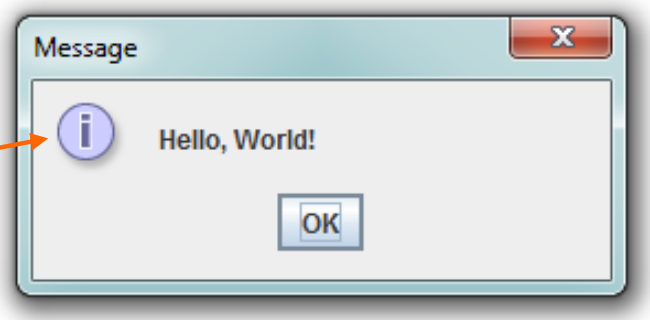
Source: https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html

Interfaces : exemple

- Version courte de la méthode *showMessageDialog*

```
JOptionPane.showMessageDialog(
   null,
   "Hello, World!"
);
```

L'icône d'information



Exemple : interface Icon

- Version plus complexe de la méthode *showMessageDialog*
  - Permet de personnaliser l'icône affichée :

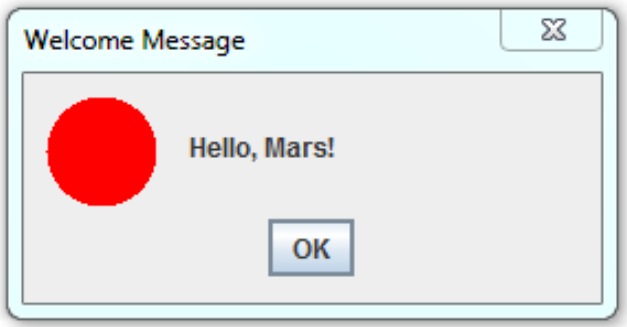
```
JOptionPane.showMessageDialog(
  null,                                 // Fenêtre parente
  "Hello, World!",                      // Message
  "Welcome Message",                    // Titre de la fenêtre
JOptionPane.INFORMATION_MESSAGE, // Type de message
  new ImageIcon("earth-icon.png")
);
```

- La classe *ImageIcon* dessine des icônes à partir d'images

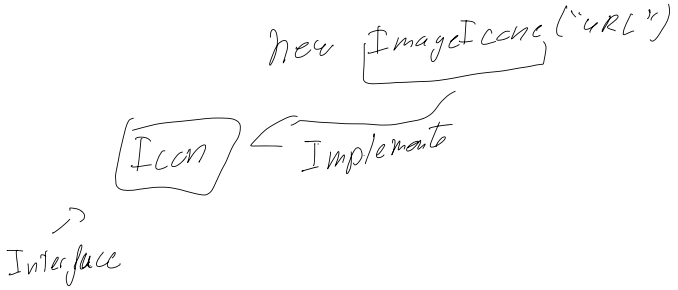
Exemple : interface Icon

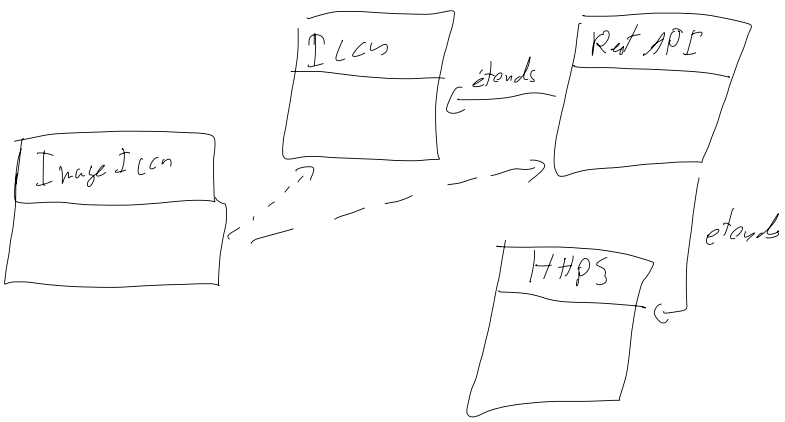
- Signature de la version longue de la méthode *showMessageDialog* :

```
public static void showMessageDialog(
   Component parentComponent,
   Object message,
   String title,
   int messageType,
   Icon icon)
```

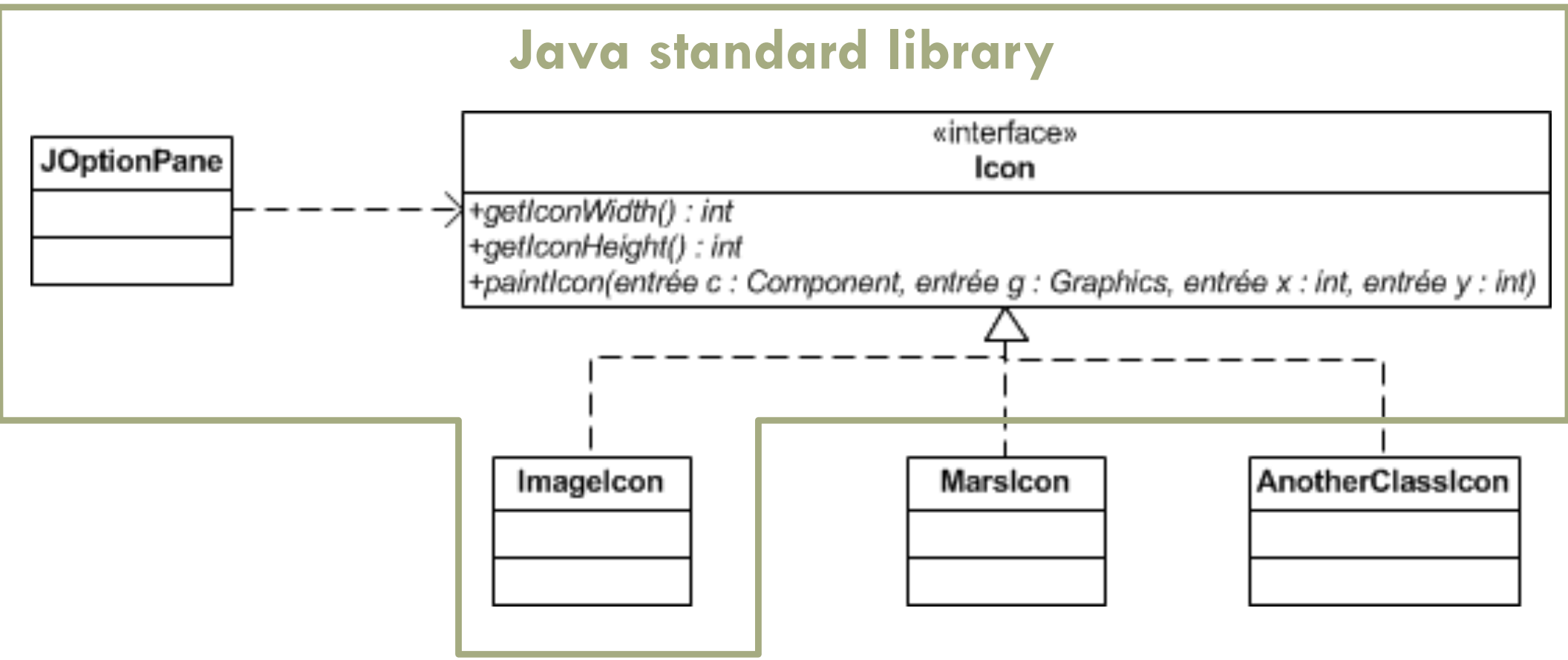


- On peut afficher n'importe quelle forme dont la classe implémente le type-interface *Icon*
- Cette classe n'est pas connue à priori


new ImageIcon("URL")
Icon
Implements
Interface


ICon
étends
Rest API
ImageICon
étends
HHPS

Exemple : interface Icon



Le type « interface »

- Une **interface** spécifie un ensemble de méthodes sans les implémenter
- La classe implémentant une interface doit implémenter toutes les méthodes définies par cette interface
- Une interface est un *contrat* entre une classe qui l'implémente et le monde extérieur
  - L'interface spécifie ce qui doit être fait par une classe sans spécifier comment

Le type « interface »

- Une interface peut définir des :
  - Constantes : attributs qui sont publiques, statiques, et finaux
  - Méthodes : une méthode est définie par sa signature (son nom, ses paramètres et leurs types)
    - Les méthodes de l'interface sont publiques
  - Types : classes ou interfaces

Le type « interface »

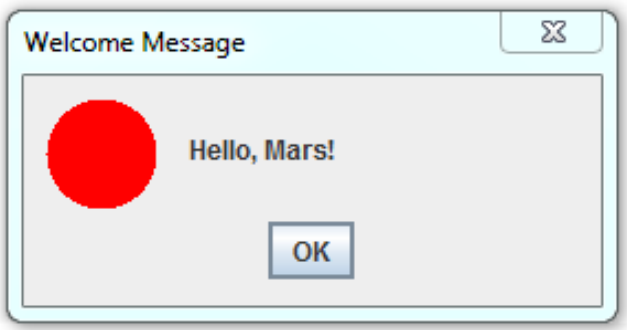
- Une interface ne **peut pas** être instanciée
  - Elle peut être implémentée par des classes
  - Elle peut être étendue par d'autres interfaces
- Une classe peut implémenter **plusieurs** interfaces
  - Elle doit implémenter les méthodes de **toutes** ces interfaces

Plan

- Interfaces
- Polymorphisme
- Les interfaces Comparable et Comparator
- Classes anonymes
- Classe abstraite

Le polymorphisme : Exemple 1

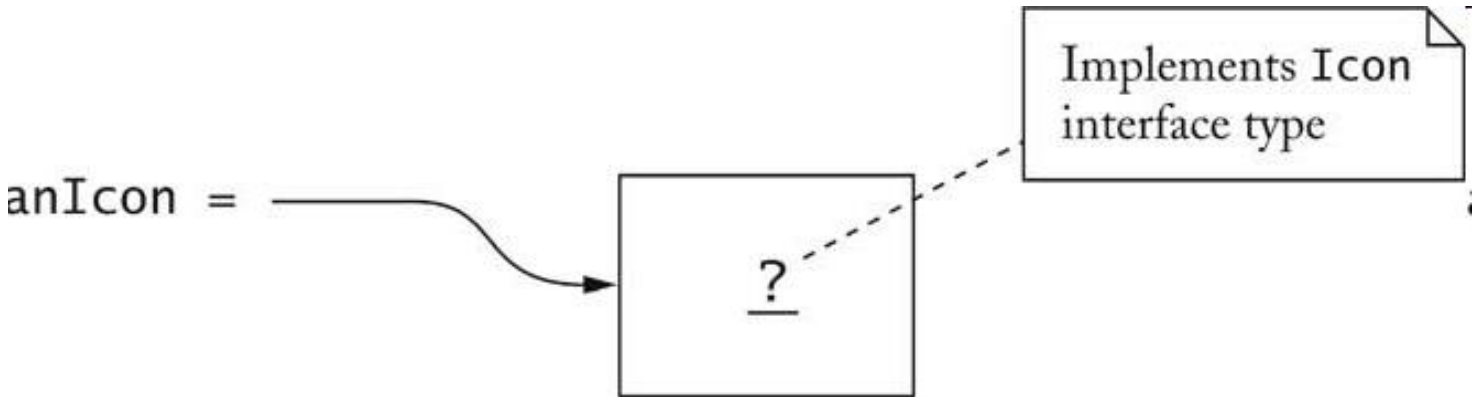
- *showMessageDialog*(…, Icon anIcon)
  - Il n'y a pas d'objet de type *Icon*
  - Le développeur de *showMessageDialog* ne sait pas quelle icône est passée en paramètre



Le polymorphisme : Exemple 1

- Comment le développeur peut-il manipuler l'icône sans savoir de laquelle il s'agit?
  - Exemple :
    - Taille de la boite de dialogue = Largeur de l'icône + largeur du message + les espaces blancs pour séparer les éléments
    - Comment connaître la largeur de l'icône?

Le polymorphisme : Exemple 1



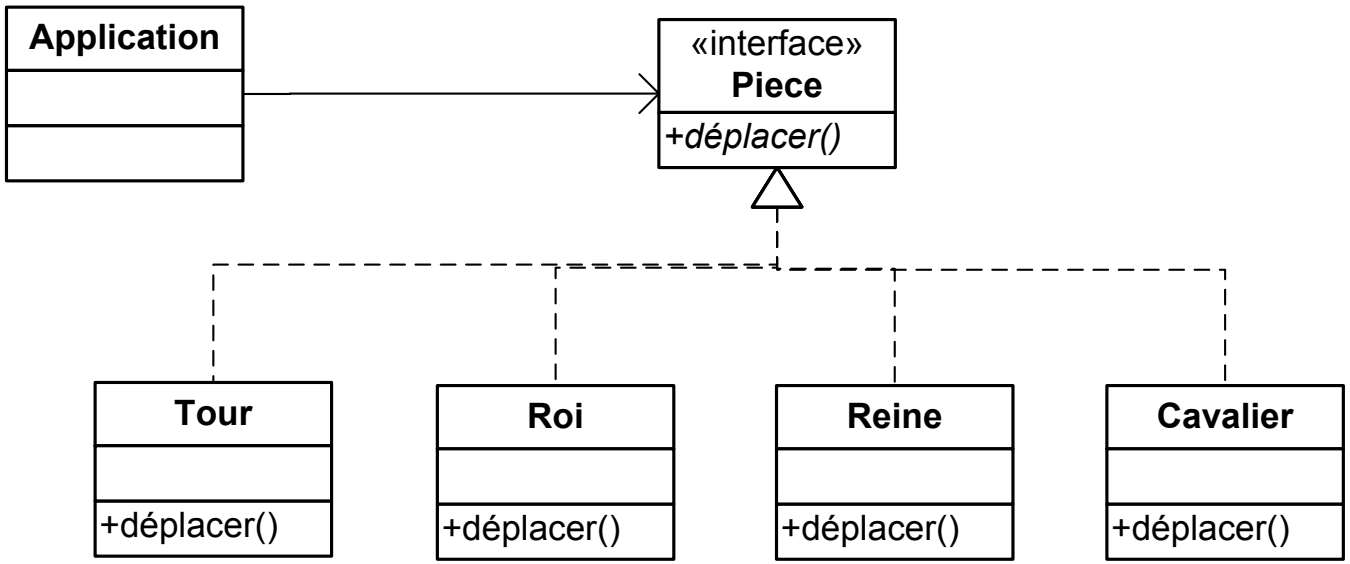
- *anIcon* est une **variable** dont le type est *Icon*
- Cette variable contient une **référence** à un **objet** qui est une **instance** d'une classe qui **implémente** *Icon*
- L'utilisateur peut donc appeler les méthodes définies par l'interface *Icon* :

```
int iconWidth = anIcon.getIconWidth();
```

Le polymorphisme : Exemple 1

- Laquelle des méthodes getIconWidth est appelée?
  `MarsIcon.getIconWidth`, `ImageIcon.getIconWidth`, ...?
- Cela dépend de l'objet sur lequel **pointe** la référence *anIcon* au moment de l'exécution
  - Dans le cas de `showMessageDialog(..., new MarsIcon(50))`, c'est la méthode *getIconWidth* de la classe *MarsIcon* qui est appelée

Le polymorphisme : Exemple 2



- Chacune des pièces du jeu d'échec se déplace de façon différente, mais l'application ne s'en préoccupe pas
  - Elle définit une variable de type *Piece*

Le polymorphisme

- Avantages?
  - Réduction du ***couplage***
    - La classe *JOptionPane* est découplée de la classe *ImageIcon*
    - Elle n'a pas besoin de connaître les détails reliés au traitement des images
    - Elle connaît uniquement les aspects exposés via l'interface *Icon*
  - Extensibilité
    - On peut facilement définir de nouveaux types d'icône

Pré-conditions et post-conditions des méthodes

- Une méthode peut être vue comme un **contrat**
- **Précondition** d'une méthode = une condition qui doit être **satisfaite avant** que la méthode ne soit appelée

```
public class FileMessages {
  // ...
  Message supprime() {
    return elements.remove(0); // La file d'attente ne doit pas être vide
  }
}
```

- **Post-condition** d'une méthode = une condition qui doit être **satisfaite après** que la méthode soit exécutée

Pré-conditions et post-conditions des méthodes

- Example (Pré-conditions):

```
/**
 * Définit le taux de rafraîchissement.
 *
 * @param  rate taux de rafraîchissement, en images par seconde.
 * @throws IllegalArgumentException si rate <= 0 ou bien
 *         rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
   // Appliquer la condition préalable spécifiée dans la méthode publique
   if (rate <= 0 || rate > MAX_REFRESH_RATE)
      throw new IllegalArgumentException("Illegal rate: " + rate);

   int intervalRefRate = setRefreshInterval(1000/rate);
}
```

Préconditions des méthodes redéfinies

- Une précondition d'une méthode redéfinie **ne peut être plus forte** que celle de la méthode de la super-classe

```
public class Employee {
  /**
   * Sets the employee salary to a given value above 10,000.
   * @param aSalary the new salary, which should be greater than 10,000
   * @throws IllegalArgumentException if aSalary <= 10,000
   */
  public void setSalary(double aSalary) { /* Test, exception, etc. */ }
}
```

  - Peut-on redéfinir `Manager.setSalary` avec une pré-condition salary > 100,000?
  - Non – cette condition pourrait être violée :

```
Manager m = new Manager();
Employee e = m;
e.setSalary(50000);
```

Post-conditions, visibilité et exceptions des méthodes redéfinies

- Si *Employee.setSalary* a pour **post-condition** de ne pas diminuer le salaire, alors *Manager.setSalary* doit respecter cette post-condition.
- Une méthode redéfinie ne peut **pas être moins accessible**
  - La méthode redéfinie ne peut pas être « *private* » quand celle de la super-classe est « *public* »
- Une méthode redéfinie ne peut **pas lever plus d'exceptions** que la méthode de la super-classe

Post-conditions, visibilité et exceptions des méthodes redéfinies

- Une méthode redéfinie ne peut **pas lever plus d'exceptions** que la méthode de la super-classe.

```
abstract class A {
   public abstract void foo() throws IOException ;
}

class B extends A {
   @Override
   public void foo() throws SocketException {
      System.out.println(« Foo de Socket");
   } // autorisé

   @Override
   public void foo() throws SQLException {
      System.out.println(« Foo de SQL");
   } // non autorisé
}
```

Plan

- Interfaces
- Polymorphisme
- Les interfaces Comparable et Comparator
- Classes anonymes
- Classe abstraite

Collections

- La classe « **Collections** » de Java fait partie du framework Java Collections.
- Elle appartient au package ***java.util*** et fournit des méthodes utilitaires statiques pour opérer sur des collections, telles que des *listes*, des *sets* et des *maps*.
- Ces méthodes incluent des fonctionnalités telles que le tri, la recherche, l'inversion, le remplissage, le mélange (shuffle) et les opérations thread-safe.

L'interface Comparable

- La classe « *Collections* » de Java a une méthode statique « *sort* » pour trier des listes

```
ArrayList<E> myList = ...
Collections.sort(myList);
```

- Les objets de *myList* doivent appartenir à une classe qui implémente le type interface *Comparable*

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

  - `object1.compareTo(object2)` retourne :
    - Une valeur négative si *object1* est plus petit que *object2*
    - 0 s'ils sont identiques
    - Une valeur positive si *object1* est plus grand que *object2*

L'interface `Comparable`

- Pourquoi la méthode « *sort* » exige que les éléments (qu'elle trie) implémentent l'interface *Comparable*?
  - Elle a besoin de comparer les éléments pour pouvoir les classer

```
if (object1.compareTo(object2) > 0) . . .
```

- Exemple : La classe String implémente le type interface *`Comparable<String>`* pour permettre le tri par ordre alphabétique
- Exemple : une classe *`Country`* implémente *`Comparable<Country>`* pour permettre de comparer les pays selon leur superficie

L'interface `Comparator`

- Comment ordonner les pays par leur nom au lieu de la superficie?
  - On ne peut pas implémenter *`Comparable`* deux fois!
- Il existe une autre méthode « *sort* » dans `Collections` qui permet de spécifier l'ordre de tri de façon plus flexible
- Pour utiliser cette méthode il est nécessaire de fournir un objet qui implémente le type interface `Comparator`

L'interface Comparator


```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

- Définir une classe qui implémente *Comparator*
- Créer un objet *myComp*, instance de cette classe
- Passer l'objet à « *sort* »
  - `Collections.sort(myList, myComp);`
  - Cette méthode trie la collection selon l'ordre défini dans le comparateur *myComp*
- Les éléments de la liste à comparer n'ont plus besoin d'implémenter *Comparable*

Plan

- Interfaces
- Polymorphisme
- Les interfaces Comparable et Comparator
- Classes anonymes
- Classe abstraite

Objets anonymes

- Retour sur la méthode *sort* avec un comparateur

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

- Pas besoin de stocker l'objet comparateur dans la variable *comp* : il n'est utilisé qu'une seule fois

- On utilise un objet **anonyme**

```
Collections.sort(countries, new CountryComparatorByName());
```

Classes anonymes

- La même règle s'applique aux classes
  - Pas besoin de nommer les classes utilisées une seule fois
  - On utilise une classe interne **anonyme** (sans nom)

Classes anonymes

- L'expression anonyme

```
Comparator<Country> comp = new Comparator<Country>() {
   public int compare(Country country1, Country country2) {
    return country1.getName().compareTo(country2.getName());
   }
};
```

  - Définit une classe anonyme qui implémente *Comparator*
  - Implémente la méthode « *compare* » pour cette classe
  - Construit un objet de cette classe

Plan

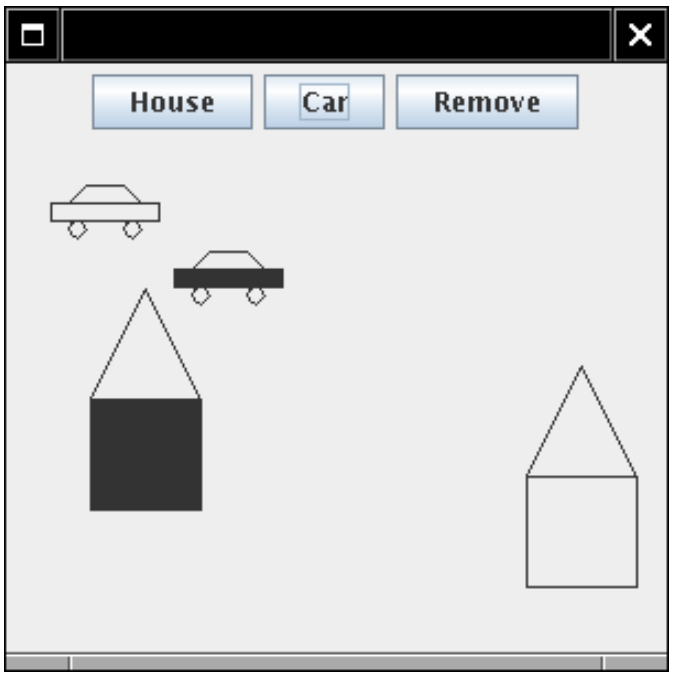
- Interfaces
- Polymorphisme
- Les interfaces Comparable et Comparator
- Classes anonymes
- Classe abstraite

Classes abstraites

- Une classe abstraite est une classe déclarée abstraite : elle peut inclure ou non des méthodes abstraites. Les classes abstraites ne peuvent pas être instanciées, mais elles peuvent être sous-classées.

Classes abstraites

- Exemple de conception orientée objet : éditeur de scènes qui dessine des formes variées
  - L'usager peut ajouter, supprimer et déplacer les formes
  - L'usager sélectionne et désélectionne une forme en cliquant dessus avec la souris et une forme sélectionnée est mise en surbrillance



Classes abstraites

- Il existe un ensemble d'opérations communes aux formes que l'on peut spécifier dans une interface *SceneShape* :

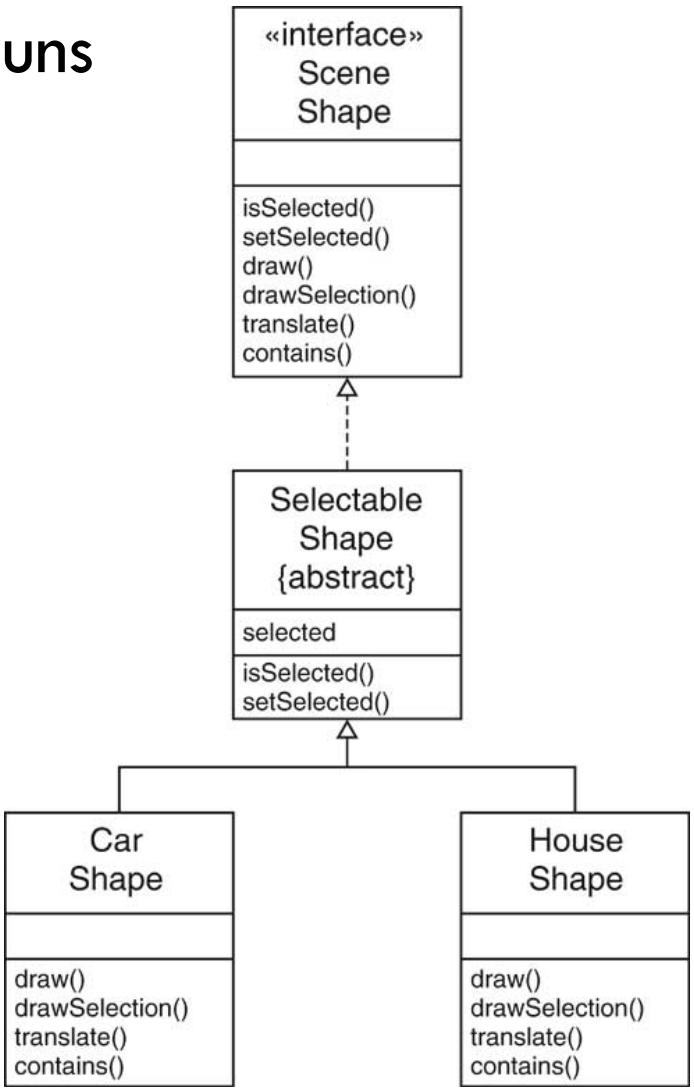
```
public interface SceneShape {
    void setSelected(boolean b);
    void draw(Graphics2D g2);
    void drawSelection(Graphics2D g2);
    boolean isSelected();
    void translate(int dx, int dy);
    boolean contains(Point2D aPoint);
}
```

- Les classes des différentes formes (*CarShape*, *HouseShape*) vont implémenter *SceneShape*
- Ces classes doivent définir un attribut « *selected* »
  - Chacune des classes va implémenter *setSelected* et *isSelected* de façon identique!

Classes abstraites

- On peut factoriser les comportements communs dans une super-classe *SelectableShape* :

```
public abstract class SelectableShape
  implements SceneShape {
  private boolean selected;
  public void setSelected(boolean b) {
    selected = b;
  }
  public boolean isSelected() {
    return selected;
  }
}
```



Les classes des différentes formes (*CarShape*, *HouseShape*) vont étendre *SelectableShape*

Classes abstraites

- *SelectableShape* ne définit pas toutes les méthodes de *SceneShape*
  - Les méthodes non définies sont dites *abstraites*
    - *draw*, *drawSelection*, *translate*, *contains*
- SelectableShape est dite **abstraite**

```
public abstract class SelectableShape implements SceneShape
```

- On ne **peut pas instancier** une classe abstraite
- *HouseShape* et *CarShape* sont des classes **concrètes**

- On peut avoir une variable dont le type est une classe abstraite

```
SelectableShape s = new HouseShape();
```

Classes abstraites (Règles)

- Une méthode abstraite doit être déclarée comme "public", ce qui est logique car elle est destinée à être redéfinie dans une classe dérivée.
- Lors de la déclaration d'une méthode abstraite, des noms d'arguments fictifs doivent être inclus dans l'en-tête de la méthode.
- Une classe dérivée d'une classe abstraite n'est pas tenue de redéfinir toutes les méthodes abstraites de sa classe de base. Dans ce cas, elle demeure abstraite, mais il est nécessaire d'utiliser le mot-clé "abstract" dans sa déclaration.
- Une classe abstraite peut avoir un ou plusieurs constructeurs, mais ces constructeurs ne peuvent pas être abstraits.

Source: Delannoy C. *Programmer En Java*. Vol 10e édition. Eyrolles; 2018. Accessed September 25, 2023. https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1662025&site=ehost-live

Classes abstraites (Règles)

- Exemple:

```
abstract class Mere {
        public abstract void m1();
        public abstract void m2(int c);
        // ...
}

abstract class Enfant extends Mere // "abstract" est
obligatoire ici
{
        public void m1() { ... }

        // Pas de définition de m2
}
```

Classes abstraites vs. interfaces

- Les classes abstraites **peuvent avoir** des attributs
- Les classes abstraites **peuvent implémenter** des méthodes
- Dans Java, une classe peut étendre **une seule** autre classe

- Les interfaces peuvent avoir seulement des **constantes** (`public static final`)
- Les interfaces peuvent seulement **déclarer** des méthodes *
- Une classe implémente un **nombre quelconque** d'interfaces

* *Java 8 : implémentation par défaut*