Hiérarchie de classes

Relation de généralisation (bas vers le haut)

Un chien est un animal. Tous les chiens sont des animaux.

Un chat est un animal. Tous les chats sont des animaux.

Tous les animaux ne sont pas des chiens (idem pour chats).

Héritage de comportement:

tout ce que sait faire un objet de type général un objet de type particulier sait aussi le faire soit de la même manière soit de manière propre

- Même manière: pas de rédéfinition de méthode
- Manière propre ou spécifique: rédéfinition de méthode

Hiérarchie de classes

- **Geler l'héritage et la redéfinition de méthode**

aucune sous classe de Carre ne pourra être définie

```
final class  Carre extends Rectangle
{/* … */}
```

```
class Couronne extends PlaqueCirculaire
{
  // ...
```

aucune sous classe de Couronne ne pourra redéfinir la méthode surfaceDuTrou()

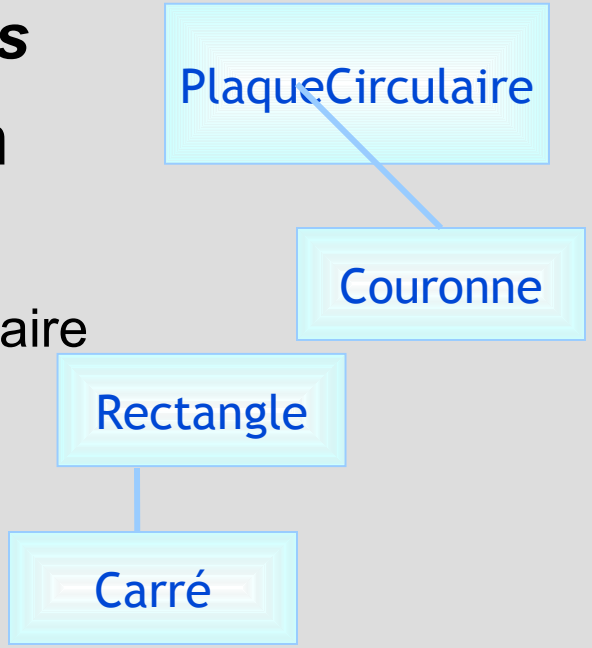
```
  final double surfaceDuTrou()
  {/* … */}

  // …
}
```

Polymorphisme

- **Polymorphisme**
  - Le polymorphisme repose sur la règle de **compatibilité de type** qui consiste à remarquer ***qu'une sous-classe possède toutes les caractéristiques de ses ancêtres***
- **Relation d'héritage** : traduction

Sorte de …

- Une couronne est une sorte de plaque circulaire
- Un carré est une sorte de rectangle



Polymorphisme

- **Exemple**

```
class  Fontes
{
public static void main (String[] args)
    {       // déclarations possibles
       PlaqueCirculaire unePlaque;
       unePlaque = new Couronne(100, 80, 20);

        Rectangle egout = new Carre(80, 120, 20);
       unePlaque = egout;
    }
}
```

- Pour que le polymorphisme fonctionne, il est nécessaire que toutes les classes dérivent de la même classe
  - Les méthodes sont communes à tous les objets

Objet et classe

- Il est possible de tester l'appartenance d'un objet à une classe:

  - On utilise l'opérateur **instanceof** qui retourne

VRAI si l'instance (l'objet) appartient à la classe

FAUX dans le cas contraire

  - Rappel:
    - VRAI **boolean true**
    - FAUX **boolean false**

Objet et classe

- Exemple:

```
class Fontes
{
    public static void main (String[] args)
    {
            PlaqueCirculaire unePlaque = new Couronne(100, 80, 20);

            if (unePlaque instanceof Couronne)
                double s = ((Couronne) unePlaque).surfaceDuTrou();
      //…
    }
}
```

Observer bien le transtypage d'une instance de PlaqueCirculaire dans l'expression (Couronne) unePlaque afin de pouvoir appliquer la méthode surfaceDuTrou de la classe Couronne.

Interface

Définition

- Une interface est une sorte de classe abstraite définie par le mot clé **interface** en lieu et place du mot clé **class**.
- Elle définie des **méthodes abstraites** destinées à être redéfinies et donc implémentées dans les classes qui en hériteront.
- Une interface ne définit pas de code pour les méthodes.
- Elle s'intéresse à la problématique du **QUOI** sans se préoccuper du **COMMENT**
- **QUOI**: les méthodes de l'interface, les services destinés à être implémentés, ce que l'on saura faire...
- **COMMENT**: l'implémentation des services, ce que l'on sait faire concrètement

Interface

- Comment faire pour que des objets sans rapport les uns avec les autres possèdent des caractéristiques communes?
- Exemple
  - *En considérant la hiérarchie des plaques de fontes, des factures, et des clients, il peut être utile que des objets issus de toutes ces classes possèdent des méthodes communes qui permettent l'enregistrement de données dans un fichier de la classe Fichier*

Interface

- **Mise en œuvre**
  - Une classe d'Interface ne peut être mise en œuvre qu'à la suite d'un héritage. Le mot clé à utiliser est **implements** en lieu et place de **extends**.
  - Les méthodes de l'interface doivent être définies avec le mot-clé **abstract** (facultatif) et la visibilité **public**

Interface

```
interface Enregistrable
{
        abstract public void EnregistrerDans(Fichier f);

        abstract public void LireDepuis (Fichier f);
}


class Facture implements Enregistrable
{
        // …
        public void EnregistrerDans (Fichier f)
        {
        // Instruction d'enregistrement de facture
        }
        public void LireDepuis (Fichier f)
        {
        // instructions de lecture de la facture
        }
}
```

Interface

```
Class Avoir extends Facture
{
  // …
public void EnregistrerDans (Fichier f)
 {
     // ...
  }

public void LireDepuis (Fichier f)
  {
     // ...
   }
}
```

```
Class Appli {
public static void main (String[] args}
  {
   Facture aRelancer = new Facture (1000,
                                    19.6);

   Enregistrable unAvoir = new Avoir (2000,
                                    19.6);
   }
}
```

Les types Avoir et Facture des classes sont compatibles avec le type Enregistrable de l'interface.

Classe abstraite

- **Définition**

Une classe abstraite est une classe ayant au moins une méthode abstraite.

Une méthode abstraite ne possède pas de définition.

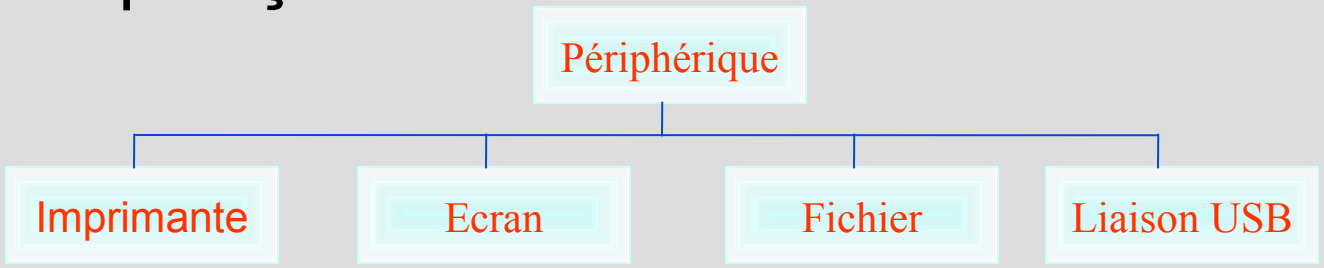
Une classe abstraite ne peut pas être instanciée (`new`).

Une classe abstraite peut toutefois posséder des constructeurs (visibilité protected pour les sous-classes)

Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.

Classe abstraite

- A quoi ça sert?



- Problème pour la construction
  - Qui hérite de qui?
  - On définit lire() et écrire() comme des méthodes abstraites dans la classe Peripherique
  - Il est nécessaire d'écrire pour toutes les sous-classes les méthodes lire() et écrire()

Classes abstraites

- Exemple

```
abstract class Peripherique
{
abstract public void ecrire (char Lettre);
abstract public void lire();
}
class Ecran extends Peripherique
{
public void ecrire(char Lettre) { ... }
public char lire() {  … }
}
class Imprimante extends Peripherique
{
 public void ecrire(char Lettre) { ... }
public char lire() {  … }
}
```

```
class Appli
{
        public static void main(String[] args)
        {
                // …
        Peripherique ecran = new Ecran()
        // …
       Peripherique imprimante = new Imprimante();
        // …
        ecran.ecrire ('Z');
        imprimante.ecrire('Y');
        ecran.lire ();
        imprimante.lire();
        }
}
```

Méthodes et classes abstraites se déclarent avec le mot clé *abstract*

Tableaux et polymorphisme

- Exemple
  - Voici un exemple de tableaux avec la classe **PlaqueCirculaire** et la sous-classe **Couronne**

```
Class Appli
{
public static void main (String args[])
        {
        System.out.println (Plaques en Fontes");
        PlaqueCirculaire catalogue [] = {       new PlaqueCirculaire(100,20);
                                                new PlaqueCirculaire(120,25);
                                                new Couronne (100, 60, 25);
                                                new Couronne (120, 80, 30);
                                        };
        // EDITION DE LIEN DIFFEREE (DYNAMIQUE)
        for (int i = 0; catalogue.length; i++)
                catalogue [i].afficher();
        }
}
```

Tableaux et polymorphisme

```
public class abstract Shape {
    public abstract double perimeter();
}
public class Circle extends Shape {
    ...
    public double perimeter() { return 2 * Math.PI * r ; }
}
public class Rectangle extends Shape {
    ...
    public double perimeter() { return 2 * (height + width); }
}
...
Shape[] shapes = {new Circle(2),
                  new Rectangle(2,3), new Circle(5)};
double sum_of_perimeters = 0;
for(int i=0; i<shapes.length; i++)
    sum_of_perimeters = shapes[i].perimeter();
```

Tableaux et polymorphisme

```
abstract class Shape { public abstract double perimeter(); }

interface Drawable { public void draw(); }

class Circle extends Shape implements Drawable, Serializable {
    public double perimeter() { return 2 * Math.PI * r ; }
    public void draw() {...}
}

class Rectangle extends Shape implements Drawable, Serializable {
    ...
    public double perimeter() { return 2 * (height + width); }
    public void draw() {...}
}

...
Drawable[] drawables = {new Circle(2), new Rectangle(2,3),
new Circle(5)};
for(int i=0; i<drawables.length; i++)
    drawables[i].draw();
```

Vecteur

- Définition
  - Les vecteurs sont des tableaux d'objets qui contiennent un nombre quelconque d'éléments
- Choix
  - Il est préférable d'utiliser un vecteur dans les cas suivants
    - Gestion d'une liste d'objets
    - Nécessité de tester la présence d'un objet dans la liste
- Utilisation
  - L'utilisation de la classe Vector implique d'importer le package Vector: ***import.java.util.Vector***
  - Les méthodes de la classe Vector sont énoncées dans les transparents qui suivent

Vecteurs fonctionnalités

capacityIncrement	Nombre d'éléments à ajouter au tableau interne à chaque fois qu'on essaie de stocker plus d'un élément que peut en contenir le vecteur
elementCount	Nombre d'éléments
elementData	Tableau dans lequel sont stockés les objets
Vector(int, int)	Création d'un vecteur avec un nombre fini d'éléments et un incrément donné
Vector (int)	Création d'un vecteur avec un nombre fini d'éléments
Vector()	Création d'un vecteur vide
addElement(Object)	Ajoute un objet à la fin de la liste
capacity()	Retourne la capacité actuelle du vecteur
clone()	Fait une copie du vecteur
contains (object)	Le vecteur contient-il l'objet ?
copyInto (Object[])	Copie l'ensemble des éléments du vecteur dans un tableau d'objets
elementAt (int n)	Retourne l'objet d'indice n dans le vecteur
elements()	Retourne tous les objets les uns après les autres
ensureCapacity (int)	Vérifie que le vecteur possède la capacité de stockage donnée
firstElement ()	Retourne le premier élément du vecteur

Vecteur: fonctionnalités

indexOf (Obect)	Cherche l'objet précisé et retourne son indice, -1 si non
indexOf (Object, int)	Idem, commence la recherche à partir d'une position donnée
insertElementAt(Object, int)	Insère l'objet à une position donnée
isEmpty()	Le vecteur contient-il un élément?
lastElement()	Renvoi le dernier élément du vecteur
lastIndexOf(Object)	Retourne l'indice du dernier indice d'un objet donné
removeAllElements()	Vide le vecteur
removeElement (Object)	Retire un objet ciblé dans le vecteur
setElementAt (Object, int n)	Remplace un élément d'indice n dans le vecteur
setSize(int)	Modifie la taille du vecteur
size()	Retorne le nombre d'élément du vecteur
toString()	Convertit un vecteur en une chaîne de caractères
trimToSize()	Elimine les espaces non utilisés en fin de vecteur

Vecteur exemple

```
import.java.util.vector;

class PlaqueCirculaire
{
        // …
        void afficher()
        { … }
        // …
}

class Couronne extends PlaqueCirculaire
{
        // …
        void afficher()
        { … }
        // …
}

class Utilisation
{
        public static void main (String args[])
        {
                System.out.println ("Application avec un vecteur");
                Vector catalogue = new Vector();

                PlaqueCirculaire plaque = new PlaqueCirculaire (100, 10);

                catalogue.addElement (plaque);
                catalogue.addElement(new PlaqueCirculaire (110, 10);
                catalogue.addElement(new Couronne (110, 80, 10);
                catalogue.addElement(new Couronne (110, 80, 20);
   }
```

…./…

Vecteur exemple

```
        for (int i =0; i < catalogue.size(); i++)
        {
                // transtypage car la méthode elementAt a pour type de retour Object
                PlaqueCirculaire plaque =  (PlaqueCirculaire) catalogue.elementAt(i);

                if (plaque  !=  null)
                {
                        if (plaque instanceof PlaqueCirculaire)
                                plaque.afficher ();
                }
        }

        // Trouve une plaque
        int i = catalogue.indexOf (plaque);
        System.out.println ("La plaque est localisée en "+ i);

        // Supprime une plaque
        catalogue.removeElementAt (i);
}
```