جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

# Chapitre 6
## Classes abstraites et Interfaces

**S. BOUKHEDOUMA**

**USTHB – FEI – département d'Informatique**
**Laboratoire des Systèmes Informatiques -LSI**
sboukhedouma@usthb.dz

# Classe abstraite ?

Une classe abstraite est une classe qui **ne peut pas être instanciée -** on ne peut pas écrire ***new*** nomclasse (…)

Une classe abstraite Peut contenir des **méthodes abstraites** (zéro, une  ou plus)

Une méthode **abstraite** est une méthode qui **n'est pas définie (on écrit seulement sa signature)**

# Classe abstraite

Une classe (ou méthode) abstraite est déclarée avec le mot réservé **abstract**

**Syntaxe**: **abstract** class nomClasse
{ //attributs
//constructeur
// méthodes **(abstraites ou non)** }

**abstract** type nom-méthode (types des paramètres);
***// pas de corps de la méthode***

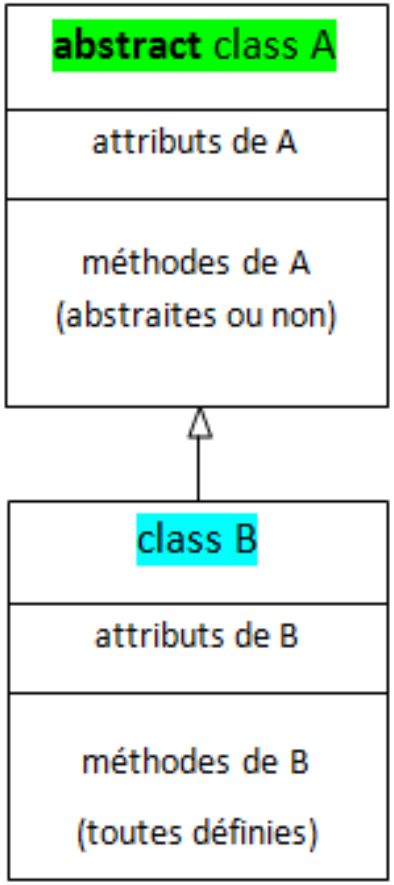
# Classe abstraite

Une classe qui contient au moins une méthode abstraite doit être **déclarée abstraite (abstract class…)**

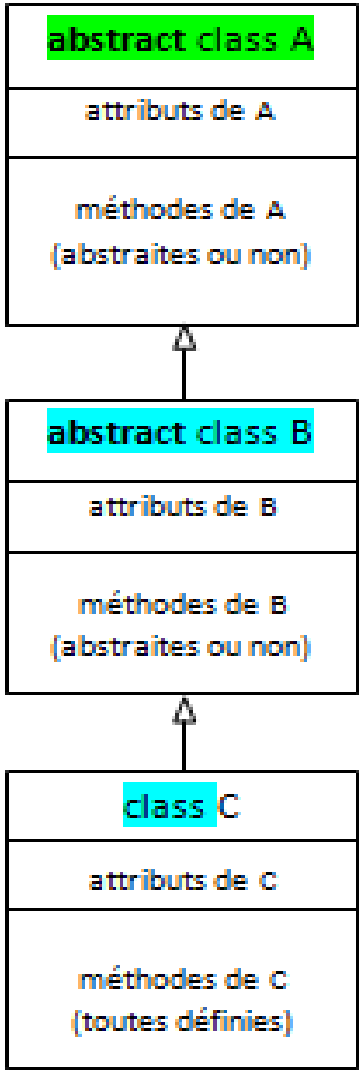
Une classe abstraite est définie pour être **étendue (dérivée)**

**Elle sert à créer d'autres classes qui peuvent être abstraites ou non** (concrètes, instanciables)

# Classe abstraite




class B **extends** A




**abstract** class B **extends** A

class C **extends** B

# Classe abstraite

## Exemple

```
public abstract class Forme {
// attributs
Point centre;  String couleur;
                                        // constructeur
public Forme ( Point C, String coul)
      { centre = new Point (C.x, C.y); couleur = coul; }

// la classe Forme contient deux méthodes  abstraites
public abstract double Perimetre();
public abstract double Surface ();      // signatures des méthodes
                                        abstraites
```

# Classe abstraite

## Exemple

```
// suite de la classe
// la définition d'une méthode peut faire appel à des méthodes abstraites

public boolean plusGrand (Forme F)
{ if this.surface() > F.surface()) return true; else return false;}

public void afficher ()
{ System.out.println ( this + Surface() + " et " + Perimetre()); }
                        // this correspond à l'appel de la méthode toString
}
} // fin de la classe abstraite
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
public class Rectangle extends Forme

{float longueur, largeur ;
// constructeur
public Rectangle (Point P,  String  C, float lg, float larg)
        { super (P, C);
           longueur = lg ; largeur = larg ;}

//définition des méthodes abstraites

public float Perimetre ()  { return ( 2*(longueur + largeur)) ;}

public float Surface ()  {return (longueur*largeur)} ;
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
// suite de la classe Rectangle

/* redéfinition de la méthode toString */

public String toString ()
{ return (" Le rectangle de longueur" + longueur + " et de largeur " +
largeur + " a pour surface et pour périmètre : ") ;}

} // fin de la classe Rectangle
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
public class Trapèze extends Forme

{float pBase gBase, hauteur ;

// constructeur
public Trapèze (Point P,  String  C, float p, float g, float h)
        { super (P, C);
           pBase = lg ; largeur = larg ;}

//définition des méthodes abstraites

public float Perimetre ()  { // somme des 4 côtés
                                  return (pBase+gBase+…);}
public float Surface ()  {return (pBase+ gBase)*hauteur/2; }
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
// suite de la classe Trapèze

/* redéfinition de la méthode toString */

public String toString ()
{ return (" Le trapèze de  petite base"+ pBase + " et de grande base " +
gBase  + " et de hauteur" + hauteur + " a pour surface et pour
périmètre : ") ;}

} // fin de la classe Trapèze
```

# Classe abstraite

## Test des sous-classes (suite Exemple)

```
public class TestFormes ()
{ public static void main (String[ ] arg)
   {  // créer les points P1 et P2

    Rectangle   R1 = new Rectangle ( P1,  "vert", 8.5, 5.0);
    Trapèze   T1  =  new Trapeze (P2,  "gris", 4.0, 6.5, 3.5)
     R1.affiche();
     T1.affiche();

      Rectangle  R2  = new  Rectangle (P2,  "orange", 10.5 , 3.5);

      if  (R1.plusGrand(R2)) System.out.println ("R1 est plus grand que R2");
              else System.out.println ("R1 n'est pas plus grand que R2");}
```

# Classe abstraite

## Test des sous-classes (suite Exemple)

On peut écrire :

Forme R1 = new Rectangle (…);
Forme T1 = new Trapèze (….);

*Mais on ne peut pas écrire :*

Forme F = new Forme (…) *car la classe Forme est abstraite*

# Classes abstraites – Avantages

Une sous-classe d'une classe abstraite peut :

- implémenter toutes les méthodes abstraites. Elle pourra alors être déclarée comme concrète et donc instanciée ;

- ne pas implémenter toutes ces méthodes abstraites. Elle reste alors nécessairement **abstraite** et ne pourra être instanciée ;

- ajouter d'autre(s) méthode(s) **abstraite(s)**. Elle reste alors nécessairement abstraite et ne pourra être instanciée.

# Utilité des classes abstraites

- Les classes abstraites permettent de définir des méthodes dépendant d'autres méthodes qui ne sont pas définies (sauf leurs signatures sont définies)

- Les méthodes qui ne sont pas abstraites et qui ne changent pas de comportement dans les classes dérivées peuvent être écrites une seule fois dans la superclasse même si elles font appel aux méthodes abstraites définies ultérieurement de manière spécifique dans les sous-classes.

# Classe abstraite

## Autre Exemple

```
public abstract class Personne {
// attributs
String nom, prénom;  int age;
                                    // constructeur
public Personne (String  nom,  String prénom, int age)
      { this.nom = nom; this.prénom = prénom; this.age = age;}

Public  String toString ()
                  { return (nom+ "\t"+ prénom + + "\t"+ age);}
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
public class Etudiant extends Personne

{long matricule; String filière; int anEtude;
// constructeur
public Etudiant (String  nom,  String prénom, int age,  long mat, String fil, int an)
        { super (nom, prénom, age);
          matricule =  mat; filière = fil; anEtude= an;}


Public  String toString ()
        { return (" Etudiant: " matricule +"\t"+ super.toString()+ "\t"+
                  filière +"\t"+  anEtude);}
}
```

# Classe abstraite

## Création de sous-classes (suite Exemple)

```
public class Enseignant extends Personne

{String NumEns; int expérience;  String grade, spécialité;
// constructeur
public Enseignant  (String  nom,  String prénom, int age,  String num,  int
                          exper, String  grade, String spec)
        { super (nom, prénom, age);
          NumEns =  num; experience = exper; this.grade = grade;
          spécialité = spec;}

Public  String toString ()
         { return (" Enseignant: " + "\t"+ NumEns +"\t"+ super.toString()+
                  "\t"+ expérience + "\t"+  grade + "\t"+ spécialité);}
```

# Utilité des classes abstraites

➢ On peut déclarer une **classe abstraite** sans qu'elle contienne **aucune méthode abstraite**

**Intérêt : on n'autorise pas la création d'objet de type superclasse (déclarée abstraite) qui réellement n'a pas d'existence dans le domaine d'étude.**

**Par exemple dans notre application, une personne est soit un « Etudiant », soit un « Enseignant » donc « Personne » dans l'absolu reste « abstrait ».**

# Utilité des classes abstraites

- Une classe abstraite peut avoir un constructeur (ou non)

- Le constructeur sera appelé par les constructeurs des classes dérivées

- On peut **définir toutes les méthodes** dans une classe et la déclarer **abstraite** car on n'autorise pas la création d'objets de la superclasse qui réellement n'a pas d'existence.

Méthode abstraite ⇒ classe abstraite

Classe abstraite ⇏ méthode abstraite (pas forcément)

# Les Interfaces

# Interface?

Une interface est une classe où **aucune méthode n'est implémentée**

Une interface est définie par:

- un ensemble de **signatures de méthodes**
- et éventuellement des **constantes de classe**

Une interface est une classe **complètement abstraite**

# Interfaces

**Syntaxe de définition**

```
interface Nom_interface {

    liste de constantes ; //optionnelle
    liste de signatures de méthodes ; }
```

Les mots **abstract** et **public** sont implicites

Une interface **n'est pas instanciable**

Une interface n'a **pas d'attributs d'instance**

**Pas de constructeur** dans une interface

# Interfaces

## Implémentation d'une classe à partir d'une interface

Une classe peut implémenter une interface en utilisant le mot « **implements** »

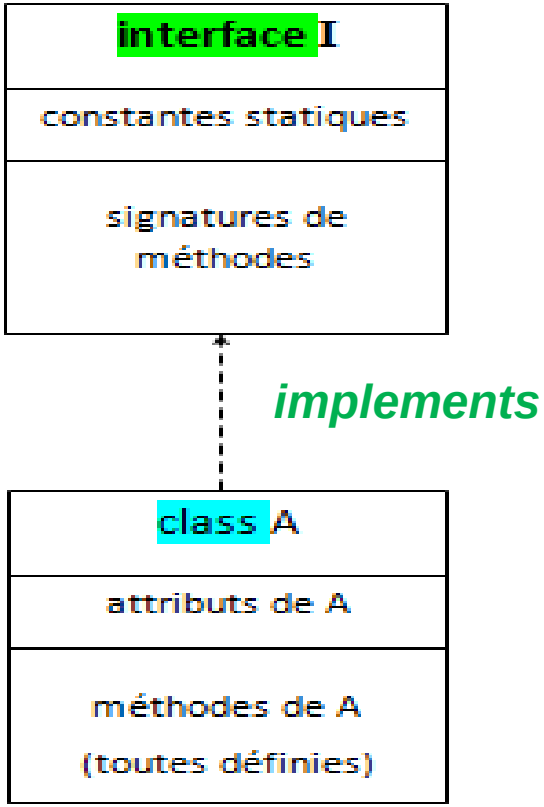
```
class Nom_classe implements Nom_interface {

    // implémenter toutes les méthodes de l'interface
}
```

# Interfaces

## Implémentation d'une classe à partir d'une interface

La classe doit implémenter toutes les méthodes de l'interface **sinon** elle va être une **classe abstraite**



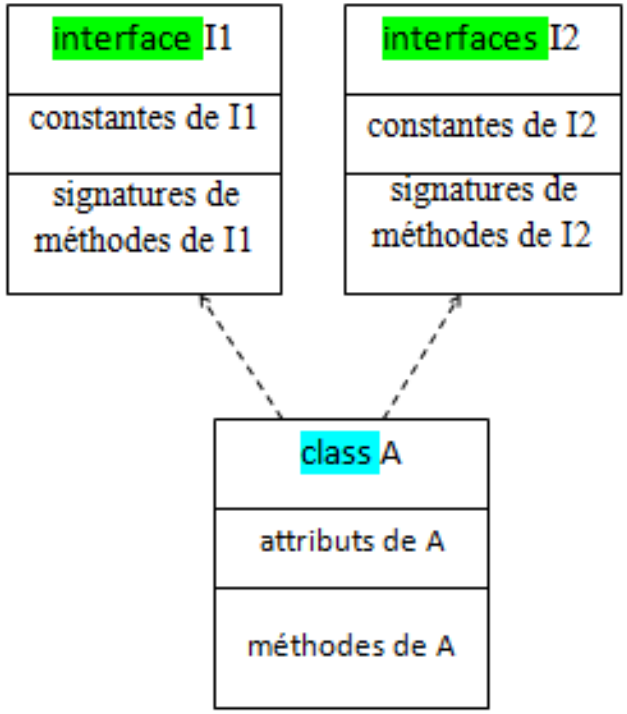
# Interfaces

## Implémentation d'une classe à partir d'une interface

Une classe peut implémenter **plusieurs interfaces**

```
class Nom_classe implements  I1, I2

   // implémenter toutes les méthodes
des interfaces I1 et I2
 }
```



# Interfaces

## Extension d'une interface

Une interface peut étendre une autre interface

```
interface Nom_interface extends I

  // ajouter d'autres signatures de méthodes
}
```

Nom_interface **hérite de toutes** les signatures de méthodes et constantes définies dans l'interface I

# Interfaces

## Extension d'une interface (héritage multiple)

Une interface peut étendre plusieurs autres interafces

```
interface Nom_interface extends  I1, I2

  // ajouter d'autres signatures de méthodes
}
```

Nom_interface **hérite de toutes** les signatures de méthodes et constantes définies dans I1 et dans I2.

# Interfaces

## Exemple d'une interface

Plusieurs classes peuvent implémenter la même interface

```
interface Liste {
        // signatures de méthodes
 public void ajouter (int);
 public void supprimer (int);
 public boolean rechercher (int); }
```

A partir de cette interface, on peut implémenter une listeFIFO, une listeLIFO, une liste triée, une liste sans répétition, …etc. même avec différentes structures sous forme de vecteur ou de liste chainée.

# Interfaces

## Exemple d'une interface

Chaque classe doit implémenter toutes les méthodes

de l'interface avec un code spécifique

```
public class ListeLIFO implements Liste {
        // attributs
//constructeur
 public void ajouter (int x) {… corps de la méthode}
 public void supprimer (int x) {… corps de la méthode}
 public boolean rechercher (int x ) {... corps de la méthode }
 }
```

# Interface Comparable

En java, une interface appelée **Comparable** est prédéfinie dans java.lang.

```
public abstract interface Comparable
                { public int compareTo (Object obj); }
```

Cette interface est souvent implémentée par des classes où on veut avoir **un ordre des éléments (un tri dans une structure).**

# Interface Comparable

**Exemple d'une classe qui implémente l'interface Comparable**

La classe **Résultat** contient les résultats de candidats à un examen

(nom, prénom, note)

```
public class Resultat implements Comparable
{ String nom, prénom;
  float note;
                                  // Constructeur
Public Resultat (String nom, String prénom; float note)
{ this.nom = nom;  this.prénom = prénom; this.note = note; }

Public  String toString ()
        { return (nom +"\t"+ prénom +"\t"+ note);}
```

# Interface Comparable

## Exemple d'une classe qui implémente Comparable

```
                    // suite de la classe Resultat

// Implémentation de la méthode compareTo

public int compareTo (Object obj)
{ Resultat R = (Resultat)obj      // cast explicite

  if (this.note < R.note ) return -1;
              else if (this.note = = R.note) return 0;
                          else return 1; }

} // fin de la classe Resultat
```

# Interface Comparable

## Exemple d'une classe qui implémente Comparable

```
Import java.util.Scanner;
Import java.util.Arrays;    //contient la méthode de tri

public class ProgTriTableau
{ public static void main (String [ ] arg)
   { int nb = 100;      // nbre de résultats à comparer
      Resultat [ ] V = new Resultat [nb];
                                  // création d'un vecteur V d'objets Resultat
int i; String nom; float note;
Scanner e = new Scanner (System.in);
```

# Interface Comparable

## Exemple d'une classe qui implémente Comparable

```
for (i = 0 ; i<nb ; i++)
{ // lire les données pour créer les objets
 System.out.println ( " donnez le nom, prénom  et la note du candidat: ");
   nom = e.nextLine();  prénom = e.nextLine(); note = e.nextFloat();
   V [i] = new Resultat (nom, prénom, note) ;
}
// appel de la méthode sort pour trier le vecteur
 Arrays.sort(V);                    // public static void sort (Object [ ] )

System.out.println (" voici les résultats des candidats ordonnés") ;
for (i = 0 ; i<nb ; i++)    System.out.println ( V[i]);
} }
```

# Interface Comparable

**Remarque**

La classe Arrays contient aussi d'autres méthodes comme:

***binarySearch***: recherche dichotomique dans un vecteur

***equals*** : vérifie l'égalité entre deux vecteurs

***fill***: remplit le vecteur avec une valeur spécifique

# interfaces

**Définition d'une classe à partir une interface et d'une autre classe**

En java, il est possible de créer une classe qui:

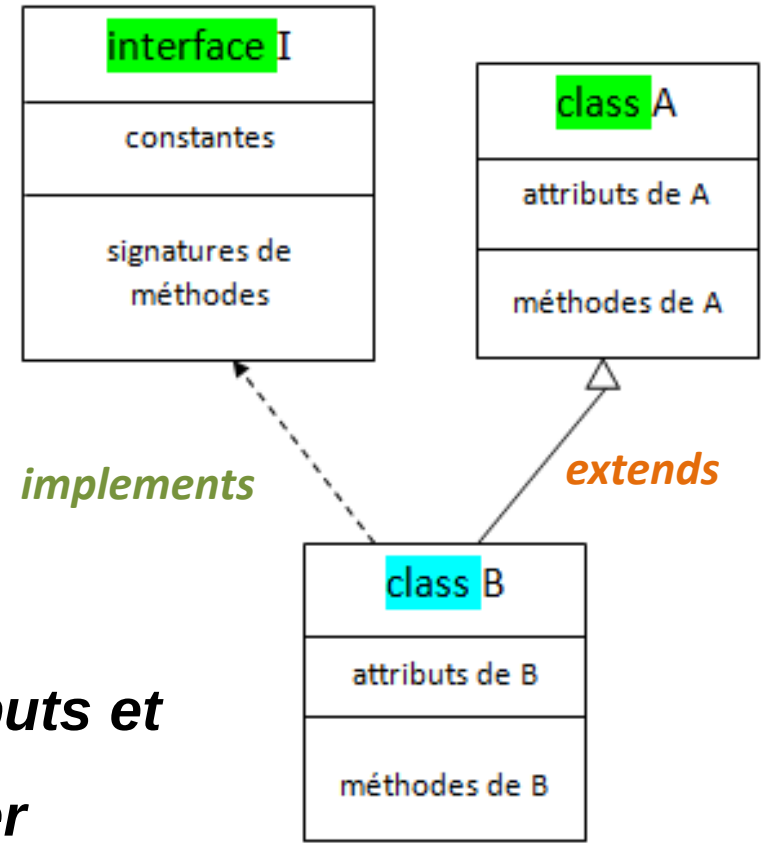
- **implémente une interface** et
- **étend (héritage) une autre classe**

Une manière de ***simuler l'héritage multiple***

# interfaces

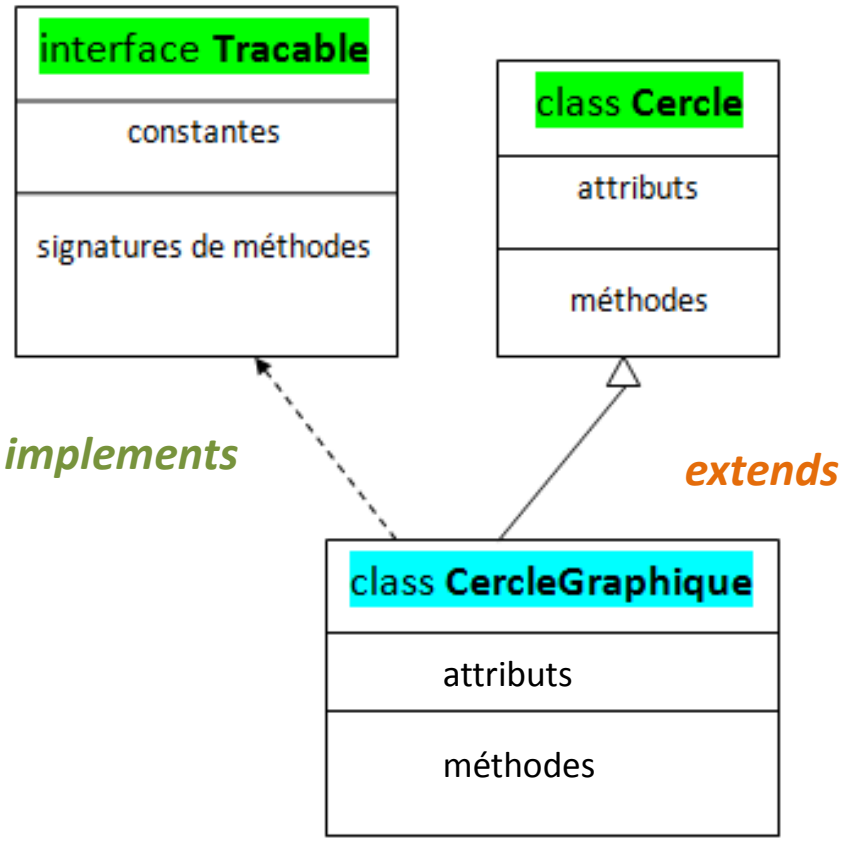
**Définition d'une classe à partir une interface et d'une autre classe**

```
Class B  extends A implements I
{   // attributs de B
   //  méthodes de B
}
```




La classe B ***hérite de tous les attributs et méthodes de A*** et ***doit implémenter toutes les méthodes annoncées dans*** $I$.

**Exemple: Définition d'une classe à partir une interface et d'une autre classe**




On veut implémenter une classe **CercleGraphique** à partir d'une classe **Cercle** et d'une interface **Traçable**

# interfaces

**Exemple (suite) :** Définition d'une classe à partir une interface et d'une autre classe

```
public class Cercle
{ Point centre;     // la classe Point est supposée définie
   double rayon;
                                    // constructeur
public Cercle ( Point C, double r)
        { centre = new Point (C.x, C.y);  rayon = r; }

... // méthodes
...
} // fin de la classe cercle
```

# interfaces

**Exemple (suite) :** Définition d'une classe à partir une interface et d'une autre classe

```
Import java.awt.Graphics;
public interface Tracable
{final static DIM_MAX = 100; // constante de classe qui définit la dimension max du tracé

Public void dessiner (Graphics g); } // signature d'une méthode dessiner
                    // g est un objet de la classe Graphics du package java.awt
}
```

# interfaces

**Exemple (suite) :** Définition d'une classe à partir une interface et d'une autre classe

```
import java.awt.Color;
public class CercleGraphique extends Cercle implements Tracable
{ Color couleur; // attribut supplémentaire

                                        // constructeur
  public  CercleGraphique ( Point C, double r, Color coul)
      { super ( C, r);
        couleur= coul;}

// implémentation de la méthode dessiner de l'interface
```

**Exemple (suite) :** Définition d'une classe à partir une interface et d'une autre classe

```
// implémentation de la méthode dessiner de l'interface
public void dessiner (Graphics g)
{ if (rayon <= DIM_MAX)
      { g.setColor (couleur) ;
        g.drawOval (Centre.getX()-rayon, Centre.getY()-rayon,
                              2*rayon, 2*rayon) ; }
    else System.out.println (" Rayon trop grand ");}
 } // fin de la classe CercleGraphique

// setColor et drawOval sont des méthodes de la classe Graphics
```

**Exemple (suite) :** Définition d'une classe à partir une interface et d'une autre classe

```
import java.awt.Graphics;

class ProgDessin
{ public static void main (String arg[ ] )
   {  Point p = new  Point (3, 2);
      CercleGraphique C = new CercleGraphique (p, 3.5, Color.red) ;

// Création d'un objet de la classe Graphics
   Graphics g = new Graphics ();

    C.dessiner (g) ;  // On dessine le cercle C à l'écran

 } } // fin de la classe ProgDessin
```

**Remarque**

De la même manière, on pourrait utiliser la classe **Carré** et l'interface **Traçable** pour implémenter un **CarréGraphique**, il suffit d'implémenter la méthode **dessiner** pour dessiner un carré.

# Interfaces

**Exemples d'interfaces prédéfinies en java**

- Interface ***Comparable***
- Interface ***Cloneable***
- Interface ***Serialisable***
- Interface ***Collection***
- Interface ***Set***
- Interface ***List***
- Interface ***Map***

# Utilité des interfaces

- Les interfaces permettent d'implémenter un ensemble de classes qui offrent un service minimum commun (**signatures de méthodes identiques**).
- Les interfaces permettent de faire du **polymorphisme** avec les objets des classes qui les implémentent.
- Les interfaces permettent **d'utiliser des objets sans connaitre leur type réel** (comment ils sont implémentés).
- Les interfaces permettent de **simuler l'héritage multiple** (non permis en java)

# Interfaces

**Interfaces à partir de java 8**

A partir de la version 8 de java, les interfaces permettent de définir:

***-des méthodes statiques (avec le code dans l'interface)***

***public static*** *void method (…)*

*{ …}*

***-des méthodes par défaut (avec le code dans l'interface)***

***public default*** *void method (…)*

*{…}*