# LES OBJETS EN JAVA

# DÉFINIR UN OBJET

# EXEMPLE DE LA CLASSE POINT

```
public class Point {
  double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double distance() {
    return Math.sqrt(x*x + y*y);
  }
}
```

# LES ATTRIBUTS

Ici **x** et **y**. Ce sont les variables internes associées à la classe.

# LES MÉTHODES “CLASSIQUES”

Ici **distance** est une méthode qui ne prend pas d’argument et qui permet de calculer la distance à l’origine du point.

- **double** réfère ici au type de sortie.
- Si une fonction ne retourne rien, le type de sortie est **void**
- () signifie que la fonction ne prend pas d’arguments en entrée.

# CONSTRUCTEUR

Une classe doit définir un constructeur dont le but est de définir les attributs (l'état) de l'instance. Si elle n'en définit pas, le compilateur définira un **constructeur par défaut** qui initialisera tous les attributs de classe aux valeurs par défaut.

C'est un choix qui peut prêter à confusion.

Le constructeur porte toujours le même nom que la classe. Ce cas particulier de méthode **n'a pas de type de sortie** (même pas *void*). Ici, son prototype est :

```
Point(double x, double y)
```

- Le constructeur utilise deux arguments nommés x et y dans le corps de la fonction
- Il est possible de définir plusieurs constructeurs avec des prototypes différents.

# NOM DES VARIABLES

Au sein d'une classe, les attributs peuvent être utilisés directement et font référence à l'instance courante (en général)

Si au sein d'une méthode, une variable est définie avec un nom identique, c'est la nouvelle variable qui sera associée à ce nom localement.

Pour référencer à nouveau l'attribut : le mot clé **this**

# Voir la différence entre

```
public double distance2() {
    int x = 0 ;
    int y = x+1 ;
    return Math.sqrt(x*x + y*y);
}
```

# et

```
public double distance3(){
    int x = 0 ;
    int y = x+1 ;
    return Math.sqrt(this.x*this.x + this.y*this.y);
}
```

# ACCÉDER AUX ATTRIBUTS ET MÉTHODES D'UNE CLASSE

Avec l'opérateur "." (lorsque cela est possible)

```
// Si p est de classe Point
p.x  ; retourne x
p.distance() ; // retourne la distance à l'origine
```

# INSTANCIER UNE CLASSE

Avec le mot clé **New** et l'appel au constructeur

```
Point myPoint = new Point() ;
```

ou

```
Point myPoint,myPoint2 ;
myPoint = new Points(1,0) ;
myPoint2 = myPoint ; // Attention, même référence
```

# RÈGLES DE NOMMAGE

- **variables** : commencent par une minuscule puis une majuscule sur les mots suivants accolés
  - ex. : x, isFed,…
- **constantes** : en capitales avec underscore si différents mots
  - ex. : PI, ARRAY_SIZE,…

- **methodes** : commencent par une minuscule puis une majuscule sur les mots suivants accolés
  - ex.: getArea(), isHappy(),…
- **classes** : commencent par une majuscules puis une majuscule sur les mots suivants accolés
  - ex. : Cat, Point, MainClass,…

# EXERCICE FIL ROUGE

Dans un nouveau projet, définir les classes **Square**, **Circle** avec des attributs pertinents.

Y implémenter des méthodes **moveTo**, **translate**, **scale**

S’inspirer de la doc SVG pour déduire une façon adéquate de représentation.

Ajouter des méthodes de calcul de **bounding box** (retourne un tableau de 4 **double**)

# LA SURCHARGE DE MÉTHODES

# DÉFINITION

Il s’agit de définir au sein d’une même classe (ou ses dérivées) une méthode dont le nom existe mais avec une signature différente.

```
public double distance(Point p){
    double dx = this.x - p.x ;
    double dy = this.y - p.y ;
    return Math.sqrt(dx*dx + dy*dy);
}
```

# On peut aussi surcharger le constructeur.

```
Point(){
    x = 0 ;
    y = 0 ;
}
```

# HÉRITAGE

# UNE CLASSE DISQUE1 QUI RESSEMBLE À POINT...

```
public class Disque1 {
    double x, y , r;
    public Disque1(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double distance() {
        return Math.sqrt(x*x + y*y);
    }
    public double distance(Disque1 d) {
        double dx = x - d.x ;
        double dy = y - d.y ;
        return Math.sqrt(x*x + y*y) ;
    }
    public double area(){
```

# PRINCIPE DE L'HÉRITAGE

Trouver un lien naturel entre deux classes :

- Permet du factoriser le code
- On n'implémente que les nouvelles fonctionnalités

# IMPLÉMENTATION

## Grâce au mot clé **extends**

```
public class Disque2 extends Point{
    // Attributs nouveaux

    public Disque2(double x,double y)
    {
        // Définir un constructeur
    }

    // Méthodes nouvelles

}
```

Montrer qu’on peut créer un tableau avec un **Point** et un **Disque2**.

# **super** permet l'appel du constructeur de la classe parente

```
public class Disque2 extends Point{
    double r ;
    public Disque2(double x,double y)
    {
        super(x,y);
        r = 2 ;
    }
    public double area(){
        return r*r*Math.PI ;
    }
    public double perimeter(){
        return 2*r*Math.PI ;
    }

}
```

# CONSTRUCTEUR DE CLASSE FILLE

La première instruction du constructeur doit être un appel à un autre constructeur de la classe ou de la classe parente. Sinon, le compilateur appelle le constructeur par défaut de la classe parente.

Cela peut provoquer une erreur de compilation si celui-ci n’existe pas.

# AUTRE UTILITÉ DE SUPER

**super** permet également l'appel à une méthode de la classe mère.

```
super.longeur() ;
```

# MOT CLÉ INSTANCEOF

Permet de tester si une instance est une certaine classe. Utile pour utiliser des méthodes de la classe initiale lorsqu'un objet est utilisé sous le type parent.

```
class SuperClass{...}
class SubClass extends SuperClass{...}

SubClass a = new SubClass();
SuperClass b = new SubClass();

SuperClass[] tab = new SuperClass[2] ;
tab[0] = a ; tab[1] = b ;

for (int i=0 ; i<2 ;i++){
  if (tab[i] instanceOf SubClass){
     ((SubClass)tab[i]).subClassMethod();
  }
}
```

# REDÉFINITION DE MÉTHODE

Une classe fille peut éventuellement **redéfinir une méthode**.

Dans ce cas, elle a **le même prototype** que la méthode de la classe parente.

# VISIBILITÉ DES ATTRIBUTS ET MÉTHODES

Devant un attribut ou une méthode :

- **private** : accessible uniquement par la classe
- **protected** : accessible par tout descendant de la classe et les classes appartenant au même *package*
- **sans mot clé** : accessible par la classe et les classes appartenant au même *package*
- **public** : accessible par toute les classes

# RÈGLES USUELLES:

Éviter autant que possible l'utilisation de **public**, en particulier sur les variables internes.

Utiliser des fonctions **getter** et **setter** pour accéder et éventuellement modifier les attributs appropriés.

# VARIABLES ET MÉTHODES DE CLASSE

Il s’agit de variables et méthodes qui sont partagées par toutes les instances de la classe.

Utilisation du mot clé **static**

On peut y faire appel directement en accolant le nom à la classe :

```
static int variableStatique = 3 ;
static void methodeStatique();

NomClasse.variableStatique ;
NomClasse.methodeStatique() ;
```

# Exemple : compteur de nombre d’instances.

```
public class TestStatic
{
    public static int i= 0 ;

    TestStatic(){
      i++;
    }
}
```

# Exemple : méthode main d’une classe

```
public static void main()
```

**Une méthode statique ne peut faire appel à des variables d’instances (non statiques)**

# RETOUR SUR LE FIL ROUGE

- Si ce n’est déjà fait, mettre à jour de façon cohérente la visibilité des méthodes et des attributs.
- Faire en sorte que chaque objet instancié ait un identifiant de type **int** différent (numéro d’instance).
- Définir la fonction **descr** pour qu’elle retourne une chaine de caractères avec le nom de l’objet ainsi que le numéro d’instance.

# LA CLASSE OBJECT

En Java, tous les objets **dérivent de la classe Object**

Ils ont héritent donc d'un certain nombre de méthodes dont certaines sont intéressantes à surcharger ou redéfinir.

# TOSTRING()

Il s’agit de la méthode retournant une représentation de l’instance sous forme d’une chaîne de caractère.

Ainsi, pour tout objet **obj**,

```
System.out.prinln(obj) ;
```

renvoie

```
System.out.println(obj.toString());
```

# EQUALS(OBJECT OBJ)

Permet de tester l'égalité entre deux objets.

Différent de l'opérateur == qui vaut **true** lorsque deux objets pointent vers la même référence

Exemple typique : les chaînes de caractère.

*Attention : **redéfinir** equals() pour des classes personnalisées implique de redéfinir la méthode **hashCode()***

# CLONE()

Méthode **protected** permet de cloner un objet.

La classe doit **implémenter l'interface Cloneable**

```
public class Test implements Cloneable{
  double x,y,z
  // Constructeur(s) + autres méthodes
  public Test clone() throws
  CloneNotSupportedException{
      return (Test)(super.clone());
  }
}
```

```
public class Test implements Cloneable{
  double x,y,z
  // Constructeur(s) + autres méthodes
  public Test clone(){
    try{
      return (Test)(super.clone());
    }catch(Exception CloneNotSupportedException){
      System.out.println("Warning");
      return new Test() ;
    }
  }
}
```

# RETOUR SUR LE FIL ROUGE

- Faire en sorte que la description soit donnée automatiquement avec “System.out.prinln()”
- Définir la fonction **toXml** qui retourne la chaîne de caractères XML permettant d’encoder le SVG.

# CLASSES ABSTRAITES ET INTERFACES

# CLASSES ABSTRAITE

Une classe abstraite est une classe dont on **interdit la création d'une instance**

Ses classes dérivées peuvent en revanche créer des instances

Elle est déclarée grâce au mot-clé **abstract**

Une méthode abstraite est une méthode dont on ne donne pas d'implémentation. Seul son **prototype est fourni** :

```
abstract protected String methodeAbstraite(int i) ;
```

La méthode peut ou non être implémentée dans une des classes filles.

**Une classe qui contient au moins une méthode abstraite est forcément abstraite.**

# EXEMPLE :

- Imaginons une classe **dog** qui serait très similaire à la classe **cat** mais dont quelques méthodes diffèrent (un chat ne ronronne pas).
- On peut imaginer une superclass **animal** qui regroupe des attributs communs et des méthodes communes(**name**, **fed**,...)
- La méthode **listen()** va exister pour **Cat** et **Dog**, mais leur implémentation différera.

# INTERFACE

Une interface est **similaire à une classe abstraite** à l'exception qu'elle ne **contient que des méthodes abstraites**

```
public Interface MonInterface{
  abstract void methode1() ;
  abstract String methode2(int i) ;
  //....
}
```

# On dit d’une classe qu’elle **implémente une interface**

```
public class A extends SuperClass implements MonInterface{
    // La classe doit fournir le code de methode1 et methode2
    void methode1(){
        // Quelques opérations
    }
    String methode2(int i){
      return Integer(i).toString();
    }
}
```

# UTILISATION

- Une interface est vue comme un type.
- Un objet implémentant une interface donnée peut être utilisé dans n'importe quel contexte où le type de l'interface est demandé
- Compatible avec le mot clé **instanceOf**
- **Une classe peut implémenter plusieurs interfaces** (permet en quelque sorte l'héritage multiple)

# RETOUR SUR LE FIL ROUGE

S’inspirer de ces exemples pour complexifier le design :

- Se donner la possibilité d’avoir plusieurs formes
- Plusieurs solutions possibles… Classes abstraites, interface…