# PROGRAMMATION ORIENTÉE OBJET AVEC JAVA

# RETOUR SUR QUELQUES TYPES

# LE PENDANT OBJET DES ENTIERS ET FLOTTANTS

## Classes **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**

```
Integer a ;
a = new Integer(3) ;
```

- On déclare une variable `a` de type `Integer`
- On crée une nouvelle instance grâce à `new` puis le constructeur.
- Ex. : doc Java Integer

# LES TABLEAUX

## Rappel de la déclaration :

```
int[] tableau ; // Déclaration d'un tableau d'entiers
```

## Initialisation avec le mot-clé new

```
tableau = new int[N] ; // N est un entier
```

- Les tableaux sont des objets
- Peuvent être initialisés avec `new`
- le **champ** `length` permet d'obtenir le nombre d'éléments du tableau

```
int N = tableau.length ;
```

# LES CHAÎNES DE CARACTÈRES

- **String** est également une classe !
- La **méthode** `length()` permet d'obtenir leur longueur
- De nombreuses autres méthodes existent (voir doc)
  - exemple : `replace`, `charAt`,...

Les chaînes de caractères Java sont **immuables** :

# QUELQUES RAPPELS

# RETOUR SUR LA CLASSE POINT

```
public class Point {
  double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public Point(){
    this(0.,0.);
  }

  public double distance() {
    return Math.sqrt(x*x + y*y);
  }
}
```

# LE CONSTRUCTEUR

- Il porte toujours le nom de la classe et n’a pas de type de retour
- On peut en avoir plusieurs avec des signatures différentes : surcharge
- Un **constructeur par défaut** ne prend pas d’arguments

# MOT-CLÉ THIS

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

```java
public double distance2() {
    int x = 0 ;
    int y = x+1 ;
    return Math.sqrt(x*x + y*y); // Renvoie toujours racine(2)
}
```

et

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

# INSTANCIER UNE CLASSE

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

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

ou

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

Ici, `myPoint2` est le même objet que `myPoint` !

# LA SURCHARGE DE MÉTHODES

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);
}
```

# RÈGLES DE NOMMAGE

- **variables** : commencent par une minuscule puis une majuscule sur les mots suivants accolés
  - ex. : x, hexString,…
- **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(), toString(),…
- **classes** : commencent par une majuscules puis une majuscule sur les mots suivants accolés
  - ex. : Point, MainClass,…

# 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.

# LES CLASSES GÉNÉRIQUES

# PRINCIPE

```
public class GenericClass<T>{
  T val ;
  public GenericClass(T val){
    this.val = val ;
  }
  public T getVal(){
    return val ;
  }
}
```

Permet d’avoir des classes qui peuvent contenir des types non connus à l’avance

# UTILISATION

```
GenericClass<Integer> gen = new GenericClass<Integer>(5);
System.out.println(gen.getVal());
```

Renvoie

```
5
```

# PAS LIMITÉ À UN SEUL TYPE

```
public class GenericClass<T,U>{
  T val1 ;
  U val2 ;

  public GenericClass(T val1,U val2){
    this.val1 = val1 ;
    this.val2 = val2 ;
  }
  public T getVal1(){
    return val1 ;
  }

  public T getVal2(){
    return val2 ;
```

# EXEMPLES

Les classes qui implémentent `List` :

- ArrayList
- LinkedList

```
import java.util.ArrayList;

...

ArrayList<Integer> array = new ArrayList<Integer>();
array.add(3);
array.add(4);
```

# Les classes qui implémentent Map :

- HashTable
- HashMap

```
import java.util.HashMap;


...


HashMap<String,Integer> myMap = new HashMap<String,Integer>();
myMap.put("Bonjour",0);
myMap.put("Au revoir",1);
```

# TYPES FIXÉS POUR UNE INSTANCE

Une fois qu'une instance est créée, le type ne peut plus être changé.

```
ArrayList<Integer> array = new ArrayList<Integer>();
array.add(3);
array.add(4);
array = new ArrayList<Double>();
```

Provoque une erreur de compilation.

# HÉRITAGE EN JAVA

# UNE CLASSE DISQUE QUI RESSEMBLE À POINT...

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

# 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 Disque extends Point{
    // Attributs nouveaux

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

    // Méthodes nouvelles

}
```

# MOT-CLÉ SUPER

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

```
public class Disque extends Point{
    double r ;
    public Disque(double x,double y,double r)
    {
        super(x,y);
        this.r = r ;
    }
    public double aire(){
        return r*r*Math.PI ;
    }
    public double perimetre(){
        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 provoque une erreur de compilation si celui-ci n’existe pas.

# PROPRIÉTÉS DE LA CLASSE FILLE

- Possède tous les attributs de la classe mère (certains attributs peuvent ne pas être accessibles directement)
- Hérite des méthodes **public** ou **protected** de la classe mère
- A ses propres attributs et méthodes supplémentaires

# REDÉFINITION DE MÉTHODE (OVERRIDE)

# AUTRE UTILITÉ DE SUPER

Lorsqu'on redéfinit une méthode, celle-ci cache la méthode de la classe mère avec la même signature. Dans certain cas, on pourrait en avoir besoin dans le code de la classe fille.

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

# Redéfinition de *distance()* pour prendre en compte *r*

```
public double distance(){
  double dPoint = super.distance();
  if (dPoint < r)
    return 0;
  else
    return dPoint -r ;
}
```

```java
public class Main
{
    public static void main(String[] args){
        Point p = new Point(2,0);
        Point d = new Disque(2,0,1);
        Disque d2 = new Disque(2,0,1);

        System.out.println(p.distance());
        System.out.println(d.distance());
        System.out.println(d2.distance());
    }
}
```

# Sortie :

```
2
1
1
```

**super** permet également d’accéder aux attributs du parent lorsque la classe fille définit un attribut avec le même nom (règle les problèmes d’intersection de noms).

# RAPPEL : POLYMORPHISME

- Un objet de type `Disque` est utilisable partout où un `Disque` ou un `Point est requis``
- On peut déclarer un objet de type `Point` et en utilisant un constructeur de `Disque`

```
Point pointDisque = new Disque(1,1,1);
```

Dans ce cas, `PointDisque` n'est pas utilisable là où un `Disque` est demandé

# CAST EXPLICITE

```
Disque d = new Disque(1,1,1);
Point d2 = new Disque(2,2,2);

Point[] tab = new Point[2] ;
tab[0] = d ; tab[1] = d2 ;

for (int i=0 ; i<2 ;i++){
  tab[i].aire();
}
```

Provoque une erreur (*cannot find symbol - method aire()*)

# Il faut demander à transtyper **d2** en **Disque** !

```
Disque d = new Disque(1,1,1);
Point d2 = new Disque(2,2,2);

Point[] tab = new Point[2] ;
tab[0] = d ; tab[1] = d2 ;

for (int i=0 ; i<2 ;i++){
  ((Disque)tab[i]).aire();
}
```

mais ce n’est pas tout…

# DANGER DU CAST EXPLICITE

```
Point p = new Point(1,2) ;
((Disque)p).aire() ;
```

Lève une exception

```
java.lang.ClassCastException: class Point cannot be cast to
class Disque
```

# MOT CLÉ INSTANCEOF

Permet de vérifier si un objet est bien une instance d'une classe fille

```
Disque d = new Disque(1,1,1);
Point d2 = new Disque(2,2,2);
Point p = new Point(1,2);

Point[] tab = new Point[2] ;
tab[0] = d ; tab[1] = d2 ; tab[2] = p ;

for (int i=0 ; i<3 ;i++){
  if (tab[i] instanceOf Disque){
      ((Disque)tab[i]).aire();
  }
}
```

# PRINCIPE DE SUBSTITUTION DE LISKOV

*Si q(x) est une propriété démontrable pour tout objet x de T, alors q(y) est vrai pour tout objet y de type S tel que S est un sous-type de T*

- Une propriété de la classe `Point` : si `p` est de type `Point`, alors `p.distance()==0` est équivalent à `p` est nul (`p.x==0`et `p.y==0`)
- Soit `d` de type `Disque`:

```
Disque d2 = new Disque(2,0,3);
```

- Alors `d2.distance()==0` mais `d.x!=0`)

```
boolean isOrigin(Point p){
  return p.distance()==0 ;
}
```

Cette fonction ne se comportera pas comme on aimerait qu'elle se comporte…

- Violer le principe de substitution de Liskov peut conduire à des bugs difficiles à identifier et rend la maintenabilité du code délicate.
- Pourrait-on faire en sorte que `Disque` ne viole pas ce principe ?
  - Renoncer à l'héritage dans ce cas
  - Ou ne pas redéfinir `distance` mais utiliser redéfinir une autre fonction
- Autre exemple de violation : utilisation de `instanceOf`

# Le contre-exemple du canard

*If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction*

# RAPPEL : HÉRITAGE MULTIPLE ?

- Java ne permet pas d'héritage multiple
- En remplacement, Java propose les **Interfaces** : propose une liste de méthodes *non implémentées*
- On dit qu'une classe **implémente** une interface : elle doit implémenter toutes les méthodes de l'interface.
- Une interface peut être utilisée comme un type : le polymorphisme s'applique également ici

# CRÉATION D’UNE INTERFACE

Comme un fichier de classe, mais pas d’attribut !

```
public interface formePleine(){
  // Pas d'attributs !

  // Liste des méthodes à implémenter
  double aire();
  double perimetre();
}
```

# UTILISATION D'UNE INTERFACE

Grâce au mot-clé `implements`

```
public class Disque extends Point implements formePleine{
    ...
}
```

Une objet de type `Disque` peut alors également être utilisé partout où une `formePleine` est demandée.

Le mot-clé `instanceOf` fonctionne également avec les interfaces

# RETOUR SUR LES EXCEPTIONS

# UNE EXCEPTION EST UN OBJET !

```java
public class Main {
  public static void main(String[] args) {
    try{
      System.out.println(1/0);
    }catch(ArithmeticException e){
      System.out.println("Ce n'est pas très bien !");
    }
  }
}
```

`ArithmeticException` est une classe qui dérive de la classe `Exception`

# DEUX MÉTHODES INTÉRESSANTES SUR LES EXCEPTIONS

- *String* getMessage() : permet de récupérer le message
- *void* printStackTrace() : permet d’afficher la pile d’exécution

# EXCEPTIONS PERSONNALISÉES

```
public class SaisieErroneeException extends Exception {
  public SaisieErroneeException() {
    super();
  }

  public SaisieErroneeException(String s) {
    super(s);
  }
}
```

**super** permet d'appeler les deux constructeurs de la classe Exception

# 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.prinlnt(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

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

# Exemples :

```
String s1 = new String("Hello World");
String s2 = new String("Hello World");
System.out.println(s1==s2);
System.out.println(s1.equals(s2));
```

# Renvoie :

```
false
true
```

## Quelques règles pour redéfinir `equals` (voir ici)

- reflexivité : `x.equals(x)` devrait valoir toujours true
- symmétrique : si `x.equals(y)` vaut true, alors y.equals(x) aussi.
- jamais égal à null : x.equals(null) devrait être toujours faux
- si `x.equals(y)` vaut true, alors `x.hashCode() == y.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());
  }
}
```

```java
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() ;
    }
  }
}
```

- De cette façon, on accède à l'implémentation par défaut de **clone** qui réalise une **copie de surface**.
- Peut être suffisant dans certains cas (les attributs sont des types primitifs)
- Dangereux lorsque certains attributs sont des objets non immuables…
- Dans ce cas, redéfinir **clone** pour s'assurer de cloner les objets membres

# 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)**

# “REDÉFINIR” UN MÉTHODE DE CLASSE"

Dans point :

```
static void whoAmI(){
  System.out.println("Un point");
}
```

Dans disque :

```
static void whoAmI(){
  System.out.println("Un disque");
}
```

```
Disque d = new Disque(1,1,1);
Point d2 = new Disque(3,3,1);
d.whoAmI();
d2.whoAmI();
((Disque)d2).whoAmI();
```

# Renvoie…

```
Un Disque
Un Point
Un Disque
```

- Les méthodes statiques ne sont pas redéfinies au sens “override”.
- Étant des méthodes de classe, le choix de la méthode est réalisé directement au moment de la compilation en fonction du type déclaré
- **Éviter d’appeler des méthodes de classe depuis des instances**, utiliser directement la classe

```
Disque.whoAmI();
Point.whoAmI();
```

# LE MOT CLÉ FINAL

# ACCOLÉ À UN ATTRIBUT D'UNE CLASSE

```
public final double x ;
```

Indique que la variable *x* ne peut pas être modifiée. Elle ne peut être assignée qu'une seule fois lors de l'appel du constructeur.

Permet de créer des classes immuables.

# ACCOLÉ À UNE MÉTHODE

```
public final void methodeFinal();
```

La méthode *methodeFinal* ne peut pas être redéfinie dans une classe dérivée.

Par extension, les méthodes *private* sont implicitement *final*.

# ACCOLÉ À UNE CLASSE

```
public final class ClassFinal
```

La classe *ClassFinal* ne peut pas être dérivée.

# AUTRES UTILISATIONS

```
public void methode(final int x,int y){
  final int z = 2*y ;
  ...
  ...
}
```

Les valeurs de l'argument *x* et la variable *z* ne peuvent pas être modifiées par la méthode. Le compilateur utilise cette information pour optimiser le code.

# CLASSES ABSTRAITES

# 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 :

- Certaines méthodes (`aire`,`perimetre`,...) existeraient dans les deux mais seraient différentes
- Peut-on imaginer une classe **Forme** qui serait parent de **Carre** et **Cercle** ?

# RAPPORT AVEC LES INTERFACES

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

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

On peut néanmoins définir des implémentations par défaut grâce au mot clé **default**

- Une interface n’a pas d’attributs
- Un classe abstraite peut en avoir
- Une classe abstraite devrait implémenter le maximum possible de méthodes

- **Classe abstraite** : classe partiellement implémentée, en raison d'un manque de spécialisation
- Une classe fille ne peut dériver que d'une seule classe (abstraite ou non)
- **Interface** : "contrat" d'implémentation : garanti que certaines fonctionnalités seront présentes
- Une classe peut implémenter plusieurs interfaces
- Une classe qui n'implémente pas complètement une interface est forcément abstraite.