Classes et Objets
(2ème partie)

dernière mise à jour 20/12/2023 08:34

Philippe Genoud
Philippe.Genoud@imag.fr

Plan

- Constructeurs
- Surcharge des méthodes
- Variables de classe (variables statiques)
- Méthodes de classe (méthodes statiques)
- Constantes
- Le **`main()`**
- Initialiseur statique
- Finalisation

Constructeurs

le retour

- **Constructeurs** d'une classe :
  - méthodes particulières pour la création d'objets de cette classe, invoquées par l'opérateur **`new`**
  - méthodes dont le nom est identique au nom de la classe.
  - pas de type de retour ni mot **`void`** dans la signature
  - retourne implicitement instance de la classe (this)
  - pas d 'instruction **`return`** dans le constructeur

- Rôle d'un constructeur
  - effectuer certaines initialisations nécessaires pour le nouvel objet créé.

- Toute classe JAVA possède au moins un constructeur
  - Constructeur définit explicitement dans le code de la classe,
  - Ou un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoqué

Constructeurs

Définition explicite

- si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoqué

```
public class Point {
   private double x;
   private double y;

   public Point(){
   }

   public void translater(double dx, double dy) {
         x += dx; y += dy;
   }
   public double distance() {
         return Math.sqrt(x*x+y*y);
   }
   public void setX(double x){
         this.x = x;
   }
   public double getX(){
         return x;
   }
   … idem pour y
}
```

```
Point p1 = new Point();
```

pas de constructeur déclaré :
constructeur par défaut (sans paramètre et sans instructions)

Constructeurs

Définition explicite

- si une classe définit explicitement un constructeur le constructeur par défaut est masqué, seuls le ou les constructeurs définis peuvent être invoqués

```
public class Point {
   private double x;
   private double y;
   public Point(double x, double y){
      this.x = x;
      this.y = y;
   }
   public void translater(double dx, double dy) {
         x += dx; y += dy;
   }
   public double distance() {
         return Math.sqrt(x*x+y*y);
   }
   public void setX(double x){
         this.x = x;
   }
   public double getX(){
         return x;
   }
   … idem pour y
}
```

~~`Point p1 = new Point();`~~ 

Le constructeur par défaut (sans paramètre et sans instructions) ne peut plus être invoqué, il faut obligatoirement passer par le constructeur déclaré

```
Point p1 = new Point(10,10);
```

Constructeurs

Constructeurs multiples

- Parfois nécessité d'initialiser un objet de plusieurs manières différentes
  - Possibilité de définir plusieurs constructeurs dans une même classe

```
public class Point {
   private double x;
   private double y;

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

   public Point(){

   }

   public Point(Point p){
      this.x = p.x;
      this.y = p.y;
   }

   ...
}
```

- créer un point à l'origine
- créer un point à partir d'un autre point
- chaque constructeur possède le même nom (le nom de la classe)
- le compilateur distingue les constructeurs en fonction :
  - du nombre
  - du type
  - de la position des arguments

```
Point p1 = new Point(10,10);
Point p2 = new Point();
Point p3 = new Point(p1);
```

- on dit que les constructeurs peuvent sont **surchargés** (*overloaded*)

Constructeurs

Appel d'un constructeur par un autre constructeur

- dans une classe définissant plusieurs constructeurs, un constructeur peut invoquer un autre constructeur de cette classe
  - appel **`this(...)`**
    - fait référence au constructeur de la classe dont les arguments correspondent,
    - ne peut être utilisé que comme première instruction dans le corps d'un constructeur, il ne peut être invoqué après d'autres instructions*
  - Intérêt
    - factorisation du code,
    - un constructeur général invoqué par des constructeurs particuliers,
    - possibilité de définir des constructeurs privés.

** on comprendra mieux cela lorsque l'on parlera de l'héritage et de l'invocation automatique des constructeurs de la super-classe*

Constructeurs

Appel d'un constructeur par un autre constructeur

- exemple

```
public class Point {
   private double x;
   private double y;

   // constructeurs
   private Point(double x, double y) {
     this.x = x;
     this.y = y;
   }

   public Point(Point p) {
     this(p.x,p.y);
   }

   public Point() {
     this(0.0,0.0);
   }

   ...
}
```

Constructeur privé uniquement accessible dans la classe où il est défini

Surchage des méthodes

- **surcharge** (*overloading*) pas limitée aux constructeurs, elle est possible pour n'importe quelle méthode
- possible de définir des méthodes possédant le même nom mais dont les arguments diffèrent
- lorsque qu'une méthode surchargée est invoquée
  - le compilateur sélectionne automatiquement la méthode dont le nombre et le type des arguments correspondent au nombre et au type des paramètres passés dans l'appel de la méthode
- des méthodes surchargées peuvent avoir des types de retour différents mais à condition qu'elles aient des arguments différents

Surcharge des méthodes

exemple

- Exemple de surcharge de méthode : distance d'un point à l'origine, distance d'un point avec un autre point

```
public class Point {
   // attributs
   private double x;
   private double y;

   // constructeurs
   public Point(double x, double y){
        ...
   }

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

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

```
Point p1=new Point(10,10);
Point p2=new Point(15,14);
```

```
double d = p1.distance();
```

```
double d12 =p1.distance(p2);
```

La méthode distance est surchargée

Variables de classes

```
public class Point {
   /**
    * abscisse du point
    */
   private double x;

   /**
    * ordonnée du point
    */
   private double y;

   ...

   /**
    * Compare 2 points cartésiens
    * @param p l'autre point
    * @return true si les points sont égaux à 1.0e-5 près
    */
    public boolean egale(Point p) {
          double dx= x - p.x
          double dy= y – p.y;
          if(dx<0)
             dx = -dx;
          if(dy<0)
             dy= - dy;
          return (dx < 1.0e-5 && dy < 1.0e-5);
    }


    ...
}
```

Modifier la classe **`Point`** afin de pouvoir modifier la valeur de la constante d'imprécision

** En général on ne définit pas une méthode d'égalité de cette manière, ont préfère redéfinir la méthode* ***`equals(Object)`*** *, mais on en reparlera lors du cours sur l'héritage*

Variables de classes

```
public class Point {
   /**
    * abscisse du point
    */
    private double x;

   /**
    * ordonnée du point
    */
   private double y;

   ...

   /**
    * Compare 2 points cartésiens
    * @param p l'autre point
    * @return true si les points sont égaux à  eps  près
    */
    public boolean egale(Point p) {
          double dx= x - p.x
          double dy= y – p.y;
          if(dx<0)
             dx = -dx;
          if(dy<0)
             dy= - dy;
          return (dx <  eps  && dy <  eps  );
    }

    ...
}
```

Modifier la classe **Point** afin de pouvoir modifier la valeur de la constante d'imprécision

1ère solution :

Ajouter une variable (un attribut) **eps** à la classe

```
/** imprécision pour tests d'égalité
 */
 private double eps = 1.0e-5;
```

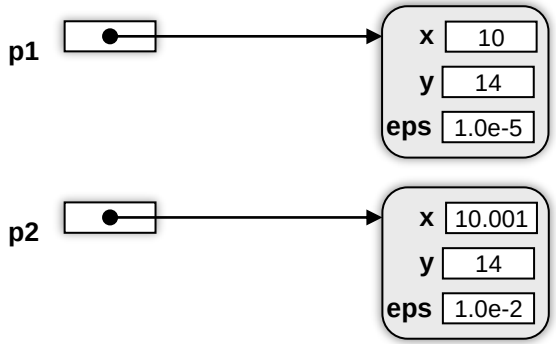
Définir un accesseur et un « modifieur »

```
/** fixe imprécision pour tests d'égalité
 * @param newEps valeur de l'imprécision
 */
public void setEps(double newEps) {
   this.eps = Math.abs(newEps);
}

/**
 * @return valeur imprécision
 */
public double getEps() {
   return eps;
}
```

Quels sont les problèmes liés à cette solution ?

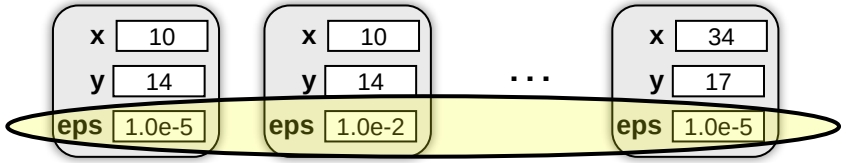
Variables de classe

- Chaque instance possède sa propre valeur de précision `egale` n'est plus garantie comme étant symétrique

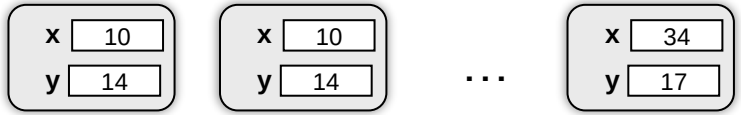



```
Point p1 = new Point(10,14);
Point p2 = new Point(10.001,14.001);
p2.setEps(10e-2);
System.out.println(p1.egale(p2)); → false  utilise la précision de p1 (0.00001)
System.out.println(p2.egale(p1)); → true   utilise la précision de p2 (0.01)
```

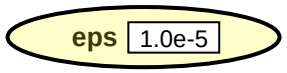
- Duplication des valeurs au niveau de chaque instance



- Cette information ne concerne pas une instance particulière mais à l'ensemble des instances de `Point`



Valeur associée à l'ensemble des points cartésiens



Mais en Java, rien ne peut être défini en dehors d'un objet.
Pas de variables globales ☺ Comment procéder ?

Variables de classe

- Pour chaque classe utilisée il existe un objet-classe la représentant, instance de la classe `Class` du package `java.lang`

> Instances of the class `Class` represent classes and interfaces in a running Java application. An enum is a kind of class and an annotation is a kind of interface. Every array also belongs
>
> `Class` has no public constructor. Instead `Class` objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the `defineClass` method in the class loader.

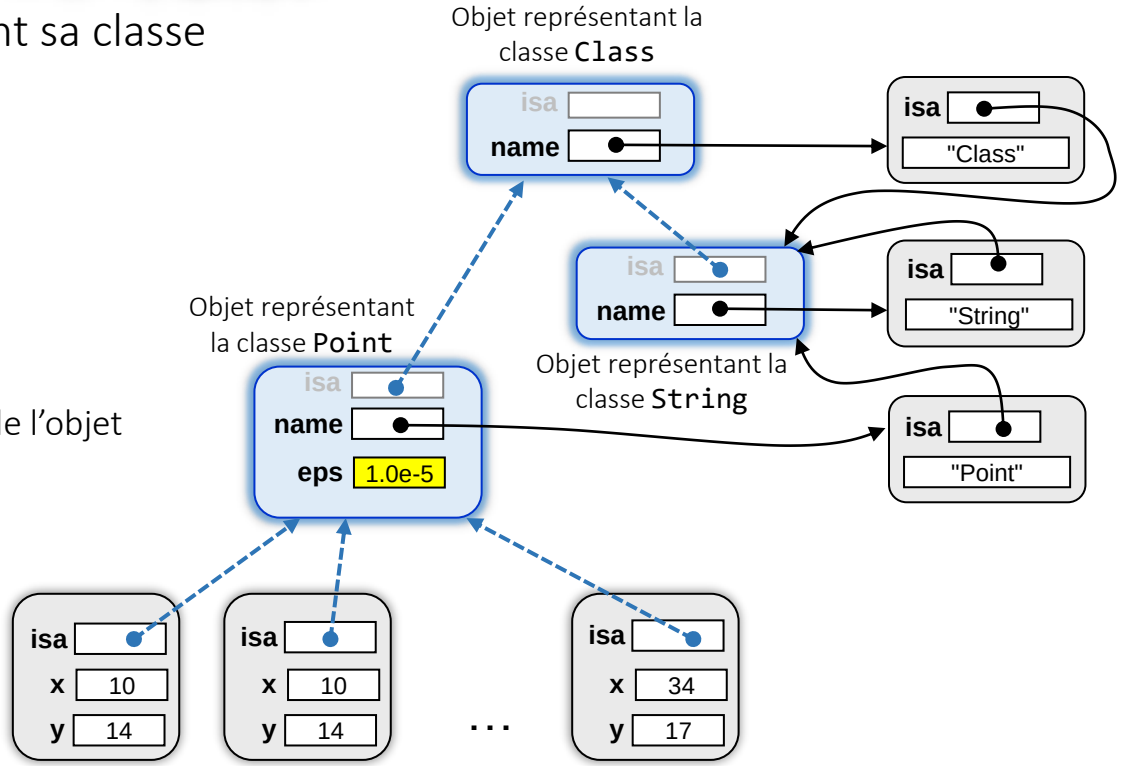
Extraits de la java doc de la classe `Class`

- Chaque instance possède une référence sur l'objet représentant sa classe

```
Point p1 = new Point(10,14);
System.out.println(p1.getClass().getName()); ----> Point
```

objet de type `Class` représentant la classe des Points

Définir `eps` comme attribut de l'objet représentant la classe `Point`



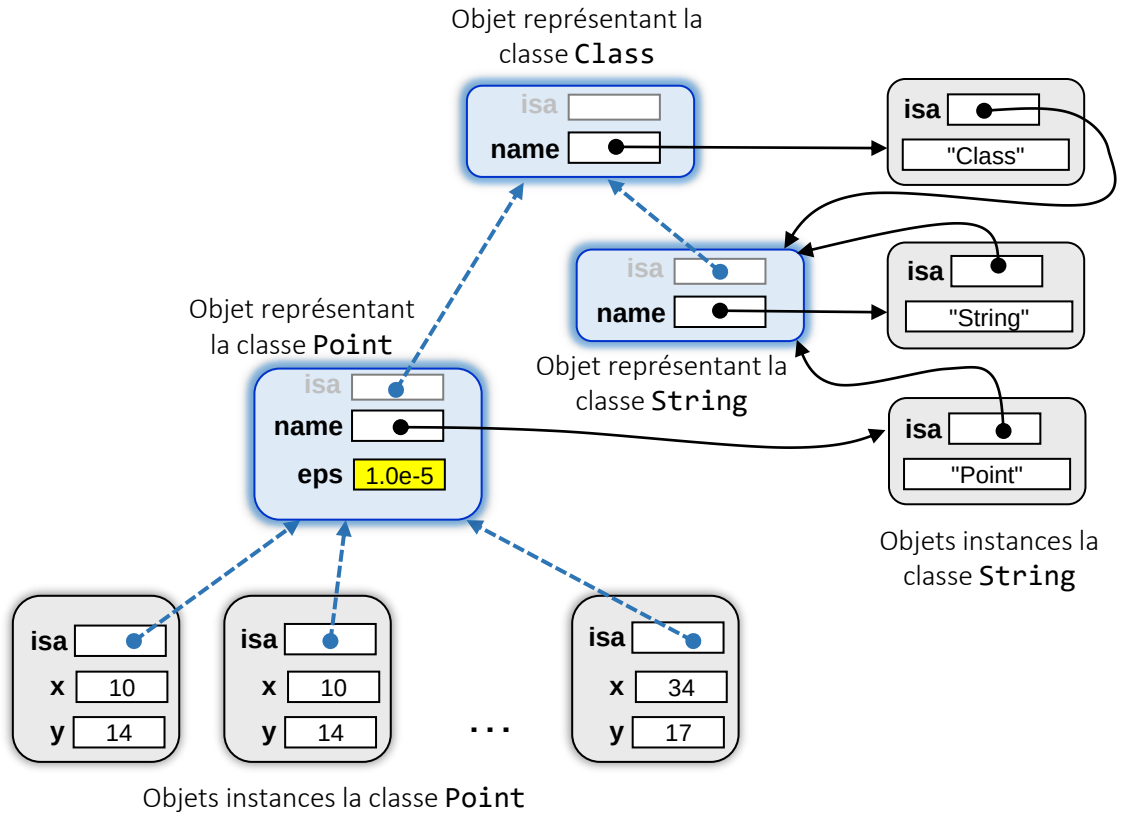
Variables de classe

- Comment définir **eps** comme attribut de l'objet-classe représentant la classe **Point** ?
  - Dans le code source la classe **Point**
  - Pour cela il faut distinguer le niveau auquel s'appliquent les descriptions (à l'objet-classe représentant **Point** ou aux instances de la classe **Point**)
  - Les descriptions qui concernent l'objet-classe sont précédées du mot clé **static**

Variable statique : variable de classe

variables d'instances

```
public class Point {

   private static double eps = 1.e-5;

   /**
    * abscisse du point
    */
    private double x;

   /**
    * ordonnée du point
    */
   private double y;

   ...
```



Variables de classe

- Comme les autres attributs, les variables de classe sont accessibles partout dans le code de la classe

Variable statique : variable de classe

variables d'instances

```
public class Point {

    private static double eps = 1.e-5;

    /**
     * abscisse du point
     */
     private double x;

    /**
     * ordonnée du point
     */
    private double y;

    ...
```

```
  /** fixe imprécision pour tests d'égalité
  * @param newEps valeur de l'imprécision
  */
  public void setEps(double newEps) {
     eps = newEps;
  }

  /** @return valeur imprécision
   */
  public double getEps() {
     return eps;
  }

  /** Compare 2 points cartésiens
  * @param p l'autre point
  * @return true si les points sont égaux à eps près
  */
  public boolean egale(Point p) {
     double dx= x - p.x
     double dy= y – p.y;
     if(dx < 0)
         dx = -dx;
     if(dy < 0)
         dy= - dy;
     return (dx < eps && dy < eps);
  }

  ...
}
```

Variables de classe

- Comment accéder à `esp` depuis un code extérieur à la classe `Point` ?
- Appeler les méthode `getEps` et `setEps` ?
  Mais nécessité de passer par une instance de la classe `Point`

```
Point p = new Point (…);
System.out.println(p.getEps());
p.setEps(1.e-8);
```

```
public class Point {

    private static double eps = 1.e-5;
    ...
```

```
    /**
     *  fixe imprécision pour tests d'égalité
     * @param newEps valeur de l'imprécision
     */
    public void setEps(double newEps) {
       eps = newEps;
    }

    /*
     * @return valeur imprécision
     */
    public double getEps() {
       return eps;
    }

    ...
}
```

Variables de classe

- Comment accéder à **esp** depuis un code extérieur à la classe **Point** ?
- Comme pour les variables, possibilité d'associer des méthodes à un objet-classe en ajoutant le modifieur **static** à sa déclaration → méthodes statiques ou méthode de classe
- Méthodes statiques peuvent être invoquées sans nécessité de passer par une instance de la classe, on y accède directement par l'objet-classe qui peut être désigné par le nom de la classe

```
System.out.println(Point.getEps());
Point.setEps(1.e-8);
```

Méthode statique

désigne l'objet-classe représentant la classe **Point**
le nom de la classe est l'identificateur de la référence créée par défaut pour désigner l'objet-classe

- **Attention** : à l'intérieur du corps d'une méthode statique il n'est possible d'accéder qu'aux membres (attributs ou méthodes) statiques de la classe (qui serait this ?)

```
public class Point {

    private static double eps = 1.e-5;
    ...
```

```
    /**
     *  fixe imprécision pour tests d'égalité
     * @param newEps valeur de l'imprécision
     */
    public static void setEps(double newEps) {
       eps = newEps;
    }

    /*
     * @return valeur imprécision
     */
    public static double getEps() {
       return eps;
    }

    ...
}
```

Les méthodes **setEps** et **getEps** ne sont plus associées aux instances de la classe **Point** mais à l'objet-classe **Point**

Api de la classe `Class`

```
public String getName()
```
Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.

```
public Class getSuperclass()
```
Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class

```
public Object newInstance() throws InstantiationException,
                                   IllegalAccessException
```
Creates a new instance of the class represented by this Class object.

…

Les membres d'une classes sont eux-mêmes représentés par des objets dont les classes sont définies dans java.lang.reflect. (Introspection des objets)

```
public Field getField(String name) throws NoSuchFieldException, SecurityException
```
Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.

public Method[] getMethods() throws SecurityException

Returns an array containing Method objects reflecting all the public member methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.

Variables de classe

Constantes

- Des **constantes nommées** peuvent être définies par des variables de classe dont la valeur ne peut être modifiée

```
public class Pixel {

    public static final int MAX_X = 1024;
    public static final int MAX_Y = 768;

   // variables d'instance
    private int x;
    private int y;
    ...
    // constructeurs

    // crée un Pixel de coordonnées x,y
    public Pixel()
    {
      ...
    }
    ...
 }
```

Déclarations de variables de classe

Le modifieur **`final`** est utilisé pour indiquer que la valeur d'une variable (ici de classe) ne peut jamais être changée

Membres statiques

- membres dont la déclaration est précédée du modifieur *`static`*
  - **variables de classe** : définies et existent indépendamment des instances
  - **méthodes de classe** : dont l'invocation peut être effectuée sans passer par l'envoi d'un message à l'une des instances de la classe.

- accès aux membres statiques
  - directement par leur nom dans le code de la classe où ils sont définis,
  - en les préfixant du nom de la classe en dehors du code de la classe
    - `NomDeLaClasse.nomDeLaVariable`
    - `NomDeLaClasse.nomDeLaMéthode(liste de paramètres)`
    - n'est pas conditionné par l'existence d'instances de la classe,

      `Math.PI` `Math.cos(x)` `Math.toRadians(90)` ...

System.out.println

???

```
System.out.println("COUCOU");
```

Classe `System`
du package `java.lang`

Variable de classe
(référence un objet de type
`PrintStream`)

Méthode d 'instance de
la classe `PrintStream`
du package `java.io`

Le main()

- Le point d ’entrée pour l’exécution d ’une application Java est la méthode statique **`main`** de la classe spécifiée à la machine virtuelle
  - Signature (profil) de cette méthode **`public static void main(String[] args)`**
    - **`args`** : tableau d’objets String (chaînes de caractères) contenant les arguments de la ligne de commande

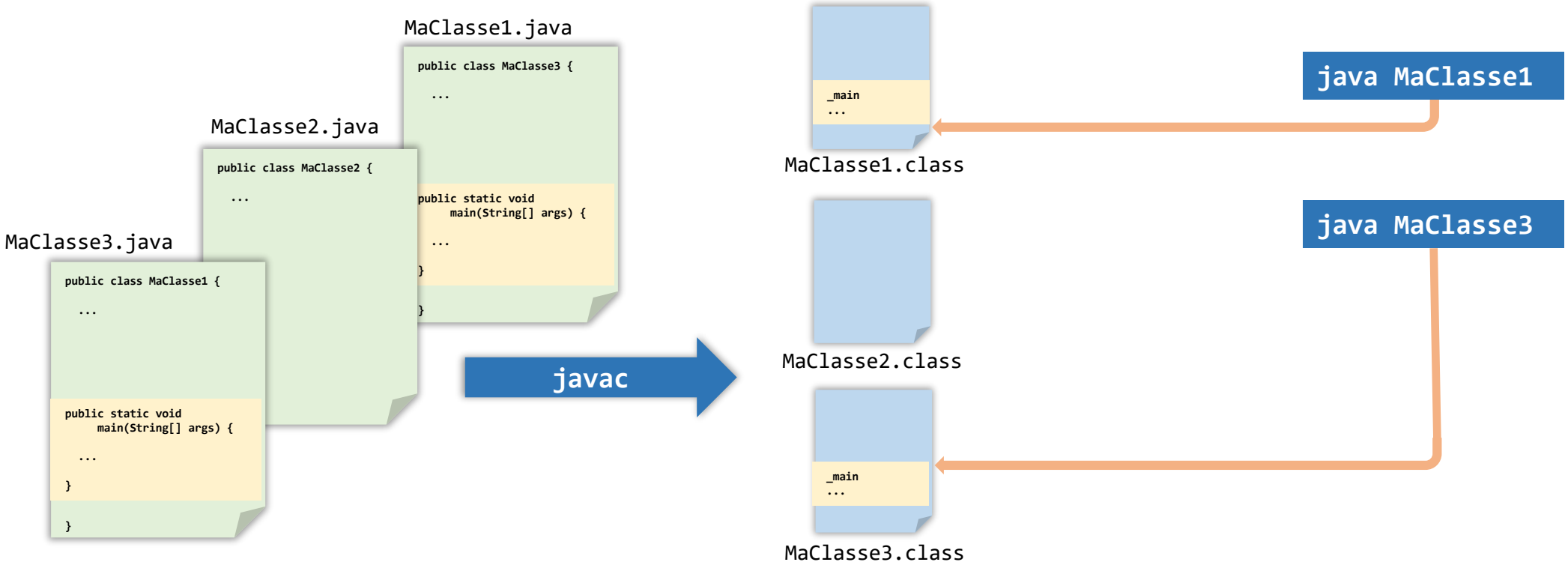
```
public class TestArgs {

  public static void main(String[] args) {
     System.out.println("nombre d ’arguments : " + args.length);
     for (int i =0; i < args.length; i++) {
         System.out.println(" argument " + i + " = " + args[i]);
     }
  }

}
```

`java TestArgs arg1 20 35.0` →

```
nombre d ’arguments : 3
argument 0 : arg1
argument 1 : 20
argument 2 : 35.0
```

Le main()

- Les différentes classes d'une même application peuvent éventuellement chacune contenir leur propre méthode `main()`
- Au moment de l'exécution pas d'hésitation quant à la méthode `main()` à exécuter
  - *c'est celle de la classe indiquée à la JVM*



Initialisation des variables à la déclaration

- les variables d'instance et de classe peuvent avoir des "**initialiseurs**" associés à leur déclaration *modifieurs type nomDeVariable = expression ;*

```
private double x = 10;
private double y = x + 2;
private double z = Math.cos(Math.PI / 2);
private static int nbPoints = 0;
```

- variables de classe initialisées la première fois que la classe est chargée.
- variables d'instance initialisées lorsqu'un objet est créé.
- les initialisations ont lieu dans l'ordre des déclarations.

```
public class TestInit {

    private double y = x + 1;
    private double x = 14.0;

    ...
}
```

javac

```
TestInit.java [4:1] illegal forward reference
    private double y = x + 1;
                       ^
1 error
Errors compiling TestInit.
```

Initialisation des variables

initialiseurs statiques

- si les initialisations nécessaires pour les variables de classe ne peuvent être faites directement avec les initialiseurs (expression) JAVA permet d'écrire une ou des instructions (algorithme) d'initialisation pour celles-ci : **initialiseurs statiques**

xs un nombre tiré au hasard entre 0 et 10
ys somme de n nombres tirés au hasard, n étant la partie entière de xs
zs somme de xs et ys

```
private static double xs = 10 * Math.random();
private static double y ;
static {
   int n = (int) xs;
   ys = 0;
   for (int i = 0; i < n; i++)
      ys =  ys + Math.random();
}
```

- Déclaration d'un initialiseur statique
  - `static { bloc de code }`
  - méthode
    - sans paramètres
    - sans valeur de retour
    - sans nom
- Invocation
  - **automatique** et **une seule fois** lorsque la classe est chargée
  - dans l'ordre d'apparition dans le code de la classe

Destruction des objets

- libération de la mémoire alloué aux objets est automatique
  - lorsqu'un objet n'est plus référencé le "ramasse miettes" ("garbage collector") récupère l'espace mémoire qui lui était réservé.
- le "ramasse miettes" est un processus (thread) qui s'exécute en tâche de fond avec un priorité faible
  - s'exécute :
    - lorsqu'il n'y a pas d'autre activité (attente d'une entrée clavier ou d'un événement souris)
    - lorsque l'interpréteur JAVA n'a plus de mémoire disponible
      - seul moment où il s'exécute alors que d'autres activités avec une priorité plus forte sont en cours (et ralentit donc réellement le système)
- peut être moins efficace que gestion explicite de la mémoire , mais programmation beaucoup plus simple et sure.

Destruction des objets

finalisation

- "ramasse miettes" gère automatiquement ressources mémoire utilisées par les objets
- un objet peut parfois détenir d'autres ressources (descripteurs de fichiers, sockets....) qu'il faut libérer lorsque l'objet est détruit.
- méthode dite de "**finalisation**" prévue à cet effet
  - permet de fermer les fichiers ouverts, terminer les connexions réseau... avant la destruction des objets
- la méthode de "finalisation" :
  - méthode d'instance
  - doit être appelée `finalize()`
  - ne possède pas de paramètres , n'a pas de type de retour (retourne `void`)
  - est invoquée juste avant que le "ramasse miette" ne récupère l'espace mémoire associé à l'objet

Destruction des objets

finalisation exemple

```
class Point {
    private double x;
    private double y;

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

    public void translater(double dx, double dy) {
        x += dx;
        y += dy;
    }

    public String toString() {
        return "Point[x:" + x + ", y:" + y + "]";
    }

   public void finalize() {
     System.out.println("finalisation de " + this);
    }
}
```

```
Point p1 = new Point(14,14);
Point p2 = new Point(10,10);
System.out.pritnln(p1);
System.out.pritnln(p2);
p1.translater(10,10);
p1 = null;
System.gc();
System.out.pritnln(p1);
System.out.pritnln(p2);
```

Appel explicite au garbage collector

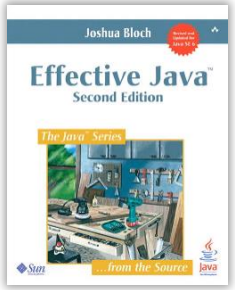
```
Point[x:14.0, y:14.0]
Point[x:10.0, y:10.0]
finalisation de Point[x:24.0, y:24.0]
null
Point[x:10.0, y:10.0]
```

Destruction des objets

finalisation

- JAVA ne donne aucune garantie sur quand et dans quel ordre la récupération de la mémoire sera effectuée,
  - impossible de faire des suppositions sur l'ordre dans lequel les méthodes de finalisation seront invoquées.
  - il vaut mieux éviter de définir et de compter sur des "finaliseurs" pour libérer des ressources critiques
  - il vaut mieux compter sur une méthode explicite de terminaison
    - ex : méthode `close()` d'un connexion JDBC (base de données)

voir Chapter 2. Creating and Destroying Objects - Item 7
du livre *Effective Java™, Second Edition*, Joshua Bloch, Ed. Addison-Wesley Professional - 2008
http://www.codeproject.com/Articles/30593/Effective-Java



Classes imbriquées

- **classes imbriquées** (*nested classes*) : possibilité de définir des classes à l'intérieur d'autres classes
- trois types de classes imbriquées
  - Classe **membre** : définition de la classe à l'intérieur d'une classe, au même niveau que les attributs et méthodes

```
class X {

    static class MemberS extends Superclass {…}
    class Member extends Superclass {…}
    ...
}
```

  - Classe **locale** : définition de la classe à l'intérieur d'une méthode

```
class X {
    void work() {
       class Member extends Superclass {…}
       ...
    }
}
```

  - Classe **anonyme** : définition de la classe à l'intérieur d'une expression

```
class X {
    void work() {
       obj = new Superclass(){…}
       ...
    }
}
```

```
class Anonymous extends/implements Superclass {}
obj = new Anonymous();
```

Les classes imbriquées sont divisées en deux catégories : non statiques et statiques.

Les classes imbriquées non statiques sont appelées **classes internes** (*inner classes*)

Les classes imbriquées déclarées statiques sont appelées **classes imbriquées statiques** (*static nested classes*).

Pour le moment on ne va s'intéresser qu'aux classes membres

Classes imbriquées

- Classes membres non statiques
  - Toute instance d'une classe membre est associée de manière interne à une instance de la classe englobante
  - Accès implicite aux membres (attributs/méthodes) définis dans la(les) classe(s) englobante(s), (y compris les membres privés)

exemple d'après *Understanding Java Nested Classes and Java Inner Classes*
By Manoj Debnath, March 6, 2017
https://www.developer.com/design/understanding-java-nested-classes-and-java-inner-classes/

accès direct aux membres de la classe externe autorisés quel que soit leur niveau de visibilité

```
public class Oyster {

   class Pearl{
      String pearl="member";
      private String privatePearl="private member";
      protected String protectedPearl="protected member";
      public String publicPearl="public member";

      public Pearl() {
         oyster="sdfsds";              // OK!
         privateOyster="sdcdscs";      // OK!
         protectedOyster="svsfdf";     // OK!
         publicOyster="sdfdfsd";       // OK!
      }
   }

   String oyster="member";
   private String privateOyster="private member";
   protected String protectedOyster="protected member";
   public String publicOyster="public member";

   public Oyster(){
      pearl="sddsfsd";                 // not OK!
      privatePearl="sdfdfdfs";         // not OK!
      protectedPearl="svsfdf";         // not OK!
      publicPearl="sdfdfsd";           // not OK!
   }
}
```

accès direct aux membres de la classe interne impossibles : il faut passer par une instance de `Pearl`

Classes imbriquées

- Classes membres non statiques
  - comme les autres membres d'une classe peuvent avoir visibilité **`public`**, **`private`**, **`protected`** ou *package private*

src > Book.java > ...

```java
public class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    public String toString() {
        return title;
    }
}
```

src > BookShelf.java > ...

```java
public class BookShelf {
    Run | Debug
    public static void main(String[] args) {

        BookList shelf = new BookList();
        shelf.add(new Book("Harry Potter"));
        shelf.add(new Book("Around the World in 80 Days"));
        shelf.add(new Book("Count of Monte Cristo"));
        shelf.add(new Book("Jataka Tales"));
        shelf.add(new Book("Aesop's Fables"));

        System.out.println(shelf);
    }
}
```

src > BookList.java > ...

```java
public class BookList {

    private BookNode list;

    public BookList() {
        list = null;
    }

    public void add(Book book) {
        BookNode node = new BookNode(book);
        BookNode tmpnode;
        if (list == null)
            list = node;
        else {
            tmpnode = list;
            while (tmpnode.next != null)
                tmpnode = tmpnode.next;
            tmpnode.next = node;
        }
    }

    public String toString() {
        String str = "";
        BookNode tmpnode = list;
        while (tmpnode != null) {
            str += tmpnode.book + "n";
            tmpnode = tmpnode.next;
        }
        return str;
    }

    // BookNode is a inner class
    private class BookNode {
        public Book book;
        public BookNode next;

        public BookNode(Book book) {
            this.book = book;
            next = null;
        }
    }
}
```

Classe interne privée

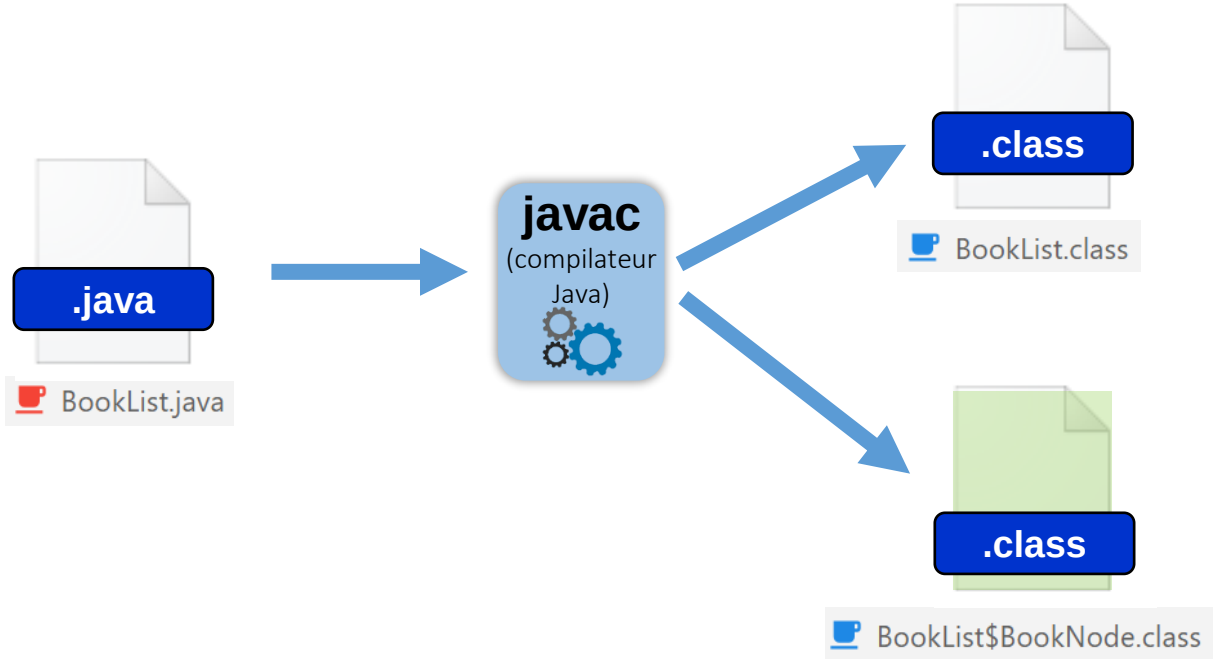
attributs publics facilite l'accès aux données dans le code de **`BookList`** (la classe étant interne et privée, il n'y a pas de risque d'atteinte au principe d'encapsulation)

exemple d'après *Understanding Java Nested Classes and Java Inner Classes* By Manoj Debnath, March 6, 2017
https://www.developer.com/design/understanding-java-nested-classes-and-java-inner-classes/

Classes imbriquées

- Compilation de classes membres
  - La machine virtuelle JAVA n 'a pas été modifiée pour la prise en compte des classes internes lors de leur introduction
  - Le compilateur convertit les classes internes en classes standard (top-level classes) que l 'interpréteur JAVA peut comprendre
  - C'est effectué à l 'aide de transformations du code source (insertion ' $ ' dans le nom des classes internes, insertion d'attributs et paramètres cachés pour gérer la liaisons entre les instances de la classe interne et instance de la classe englobante...).

```java
src > BookList.java > ...
public class BookList {

    private BookNode list;

    public BookList() {
        list = null;
    }

    public void add(Book book) {
        BookNode node = new BookNode(book);
        BookNode tmpnode;
        if (list == null)
            list = node;
        else {
            tmpnode = list;
            while (tmpnode.next != null)
                tmpnode = tmpnode.next;
            tmpnode.next = node;
        }
    }

    public String toString() {
        String str = "";
        BookNode tmpnode = list;
        while (tmpnode != null) {
            str += tmpnode.book + "\n";
            tmpnode = tmpnode.next;
        }
        return str;
    }

    // BookNode is a inner class
    private class BookNode {
        public Book book;
        public BookNode next;

        public BookNode(Book book) {
            this.book = book;
            next = null;
        }
    }
}
```



Classes imbriquées

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared `private`. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html **The Java™ Tutorials**

- si on élimine « les cas obscurs et pathologiques » (D. Flanagan in Java in a nutshell, 2nd Edition, O 'Reilly) c'est un ajout qui peut s'avérer élégant et très utile
- Pour aller plus loin :
  - *Static classes and inner classes in Java - Learn how to use the four types of nested classes in your Java code*, By Jeff Friesen, JavaWorld | Feb 20, 2020 https://www.infoworld.com/article/2074000/core-java-classes-within-classes.html