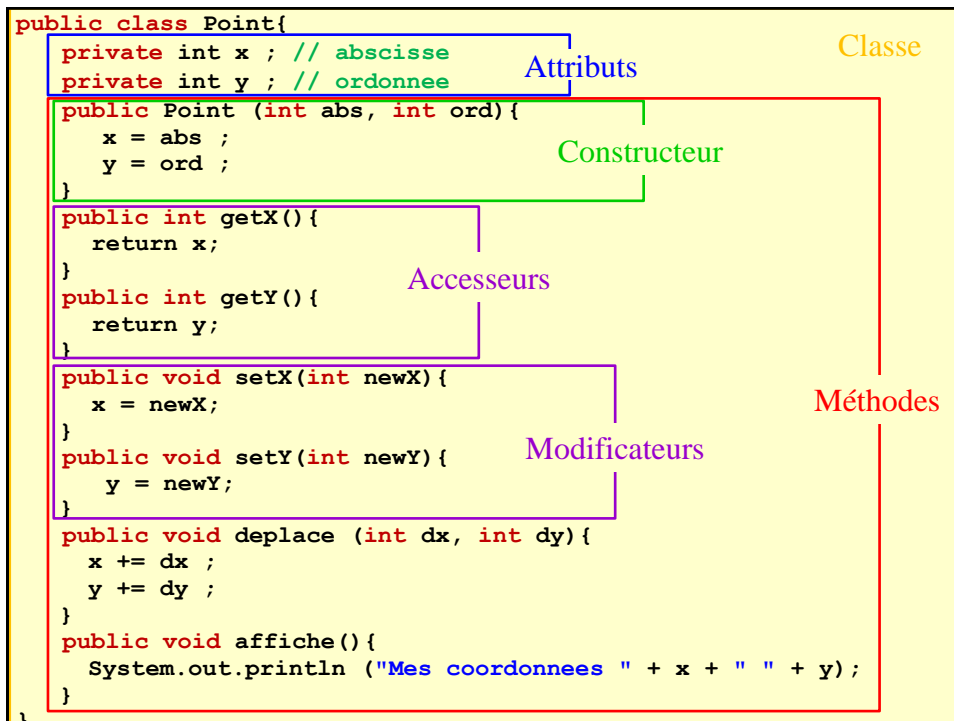


Part 3

CLASSES ET OBJETS

Class et objet

- **Objet**
 - Représente une entité dans le monde réel.
 - Possède des **propriétés** et des **comportements** (des opérations)
- **Classe**
 - La **classe** est la description d'un objet.
 - Un objet est une **instance** d'une classe → la classe décrit l'objet.
 - Description des propriétés → des **attributs** de la classe
 - Description des comportements → des **méthodes** de la classe.
 - Une classe permet **d'encapsuler** les objets:
 - les membres **public** sont **vus de l'extérieur**
 - mais les membres **private** sont **cachés**



Classe

Classe: syntaxe

- La syntaxe de déclaration d'une classe est la suivante :

```
modificateurs class nom_de_classe [extends classe_mere]
[implements interfaces] {
    ...
}
```

- Exemple:

```
public class Point{
    ...
}
```

Modificateurs de classe

Modificateur	Rôle
abstract	La classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	La classe n'est accessible qu'à partir de la classe où elle est définie (uniquement pour les classes internes).
public	La classe est accessible partout

Modificateurs de classe

- Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs.
- Le mot clé **extends** permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.
- Le mot clé **implements** permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance.

Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

Encapsulation

- L'accès direct aux variables d'un objet est possible en JAVA
- **MAIS** ... n'est pas recommandé car contraire au principe d'**encapsulation**
 - les données d'un objet doivent être privées (c'est à dire protégées et accessibles (et surtout modifiables) qu'au travers de méthodes prévues à cet effet).
- En JAVA, il est possible lors de leur définition d'agir sur la visibilité (accessibilité) des membres (attributs et méthodes) d'une classe vis à vis des autres classes
- Plusieurs niveaux de visibilité peuvent être définis en précédant la déclaration de chaque **attribut** ou **méthode** d'un **modificateur** (**private**, **public**, **protected**, -)

Les modificateurs d'accès

	public	protected	private
Classe	La classe peut être utilisée dans n'importe quelle classe.	La classe est accessible que dans les classes filles (uniquement pour les classes internes)	La classe n'est accessible qu'à partir de la classe où elle est définie.(uniquement pour les classes internes)
Attributs	Attribut accessible depuis code de n'importe quelle classe.	Attribut accessible dans les classes filles et classes du même package.	Attribut accessible uniquement dans le code de la classe qui le définit
Méthode	Méthode pouvant être invoquée depuis n'importe quelle classe.	Méthode pouvant être invoquée dans les classes filles et classes du même package.	L'usage de la méthode est réservé aux méthodes de la même classe.

Sans modificateur, une entité (classe, méthode ou attribut) est visible par toutes les classes se trouvant dans **le même package**.

Objet

Objets

- La classe est la description d'un objet.
- Un **objet** est une **instance** d'une classe.
- L'opérateur **new** crée une instance d'une classe en faisant appel à une méthode particulière appelée **constructeur**, et renvoie une référence sur l'objet instancié.
- Pour chaque instance d'une classe, le **code est le même**, seules les **données sont différentes** à chaque objet.

La création d'un objet

- Il est nécessaire de déclarer une variable ayant le type de l'objet désiré.
 - La déclaration est de la forme: **<nomClasse> <variable>;**
 - Exemple: **Point a;**
 - Contrairement à la déclaration d'une variable d'un type primitif (comme **int n ;**), **elle ne réserve pas d'emplacement pour un objet** de type Point, mais seulement **un emplacement pour une référence** à un objet de type Point.
- L'emplacement pour l'objet proprement dit sera alloué sur une demande explicite du programme, en faisant appel à l'opérateur unaire nommé **new**.

La création d'un objet

- L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable (place sa référence dans variable):

- `<variable> = new <nomClasse>(<paramètres>) ;`

- Exemple: `a = new Point(3,5) ;` // crée un objet de type Point et place sa référence dans a



- Les valeurs de x et y sont initialisées ici à 3 et 5
- Il est possible de tout réunir en une seule instruction:
 - `<nomClasse> <variable> = new <nomClasse>(<paramètres>) ;`
 - Exemple: `Point a = new Point(3,5) ;`

Initialisation d'un objet

- Les attributs d'un objet **sont toujours initialisés par défaut**.
- Dès qu'un objet est créé, et avant l'appel du constructeur, ses attributs sont initialisés à une valeur par défaut ainsi définie:

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
entier (byte, short, int, long)	0
flottant (float, double)	0.f ou 0.
objet	null

Les références et la comparaison d'objets

- Les variables de type objet ne contiennent pas un objet mais une **référence vers cet objet**.
- Lorsque l'on écrit **c1 = c2** (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1.
- L'opérateur **==** compare les références.
- Pour comparer l'égalité des variables de deux instances, il faut utiliser la méthode **equals()**.

```
Point p1 = new Point(2,20);
Point p2 = new Point(2,20);
if (p1 == p1) { ... }           // vrai
if (p1 == p2) { ... }           // faux
if (p1.equals(p2)) { ... }      //vrai
```

Objets de type String

- Un objet **String** est automatiquement créé lors de l'utilisation d'une constante chaîne de caractères sauf si celle-ci est déjà utilisée dans la classe.

```
public class TestChaines1 {
    public static void main(String[] args) {
        String s1 = "bonjour";
        String s2 = "bonjour";
        System.out.println("(s1 == s2) = " + (s1 == s2));
        // Résultat: (s1 == s2) = true
        String s3 = new String("bonjour");
        System.out.println("(s1 == s3) = " + (s1 == s3));
        // Résultat: (chaine1 == chaine3) = false
    }
}
```

- Pour obtenir une seconde instance de la chaîne, il faut explicitement demander sa création en utilisant l'opérateur **new**.

Accès aux variables et méthodes d'un objet

- Pour l'accès aux attributs et méthodes d'un objet on utilise la syntaxe suivante:
 - `<nomObjet>.<attribut>`
pour accéder à un attribut sous réserve de visibilité et des modificateurs
 - `<nomObjet>.<méthode>([<paramètres effectifs>])`
pour appeler/exécuter une méthode sur l'objet désigné
- Le mode de passage des paramètres est **par valeur** pour les types de base et **par référence** pour les objets.
 - Si un objet est passé en paramètre à une méthode alors la méthode pourra le modifier en utilisant une méthode de l'objet passé en paramètre

La durée de vie d'un objet

- Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.
- La durée de vie d'un objet passe par trois étapes :
 - la déclaration de l'objet et l'instanciation grâce à l'opérateur **new**
 - l'utilisation de l'objet en appelant ces méthodes
 - la destruction de l'objet

Destruction de l'objet

- Les objets **non référencés** sont automatiquement détruits
- La restitution de la mémoire inutilisée est prise en charge par le ramasse miettes ([garbage collector](#)).

```
Point v = new Point(3,5);  
v = null;  
// v ne référence plus l'objet, donc il sera détruit
```

```
public void m() {  
    Point v = new Point(3,5);  
}  
// v n'existe pas en dehors de la méthode, donc  
l'objet sera détruit à la fin d'exécution de la méthode
```

Le Garbage Collector

- Il prend en charge la gestion de la mémoire.
- Il alloue l'espace mémoire lors de la création des objets.
- Il libère la mémoire occupé par un objet dès qu'il n'y a plus aucune référence qui pointe vers cet objet.
- Il est capable de compacter la mémoire pour éviter la fragmentation.

La valeur « null »

- La valeur **null** peut être assignée à n'importe quelle référence sur un objet.
- Une méthode peut retourner **null**.
- L'appel d'une méthode sur une référence d'objet valant **null** provoque une erreur d'exécution .
- On peut tester une référence pour savoir si elle vaut **null**.

```
Point p = new Point(3,5);
.....
if (p != null) {
    System.out.println("le point p existe toujours! ");
}
```

Opérateurs sur les références

- Les seuls opérateurs applicables sur les références sont:
 - Egalité de deux références : **==**
 - Compare si 2 références pointent vers le même objet
 - Différence de deux références : **!=**
 - Compare si 2 références pointent vers des objets différents
 - Type d'instance de la référence : **instanceof**
 - Permet de savoir si l'objet référencé est une instance d'une classe donnée ou d'une de ses sous-classes

```
Point p = new Point(10,13);
.....
if (p instanceof Point) {
    System.out.println(" p est un point! ");
}
```

La création d'objets identiques

- Pour créer une copie d'un objet, il faut utiliser la méthode `clone()`.
- Cette méthode permet de créer un deuxième objet indépendant mais identique à l'original.

```
Point p1 = new Point(3,5);  
Point p2 = p1.clone();
```

p1 et p2 ne contiennent pas la même référence et pointent donc sur des objets différents.

Variable This

- Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation.
- **this** est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

```
public class Point{  
    private int x ;  
    private int y ;  
    public Point(int x,int y){  
        x = x ;  
        y = y ;  
    }  
}
```

```
public class Point{  
    private int x ;  
    private int y ;  
    public Point(int x,int y){  
        this.x = x ;  
        this.y = y ;  
    }  
}
```

Attributs

Attributs

- Les données d'une classe sont contenues dans des variables nommées propriétés, champs ou **attributs**.
- Ce sont des variables qui peuvent être des:
 - variables d'instances
 - variables de classes
 - constantes

Variables d'instance

- Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

```
public class MaClasse {  
    public int valeur1 ;  
    int valeur2 ;  
    protected int valeur3 ;  
    private Point valeur4 ;  
}
```

- Chaque objet de la classe possède sa propre copie de ces variables et par conséquent, des valeurs qui lui sont spécifiques.
- Cela permet de différencier un objet d'un autre.

Variables de classes

- Les variables de classes sont définies avec le mot clé **static**
- Chaque instance de la classe partage la même variable.
- Lorsqu'un objet modifie une telle variable, cette modification est accessible par tous les objets.
- Il est possible (et même préférable) de s'y référer via le nom de la classe : **classeB.n** // attribut (statique) n de la classe B

```
public class Point{  
    private int x ;  
    private int y ;  
    static int compteur = 0;  
  
    public Point(int x,int y){  
        this.x = x ;  
        this.y = y ;  
        compteur++;  
    }  
}
```

```
...  
Point p1 = new Point(3,8);  
Point p2 = new Point(7,10);  
Point p3 = new Point(10,20);  
  
System.out.println("Il y a " +  
    Point.compteur + " points");  
  
// l'instruction affichera :  
Il y a 3 points
```

Constantes

- Les constantes sont définies avec le mot clé **final** : leur valeurs ne peuvent pas être modifiées une fois qu'elles sont initialisées.

```
public class MaClasse {  
    final double PI = 3.14 ;  
}
```

Visibilité des attributs (exemple)

- Les attributs déclarés comme **private** sont totalement protégés.

```
Point p1 = new Point();  
p1.x = 10; p1.setX(10);  
p1.y = 10; p1.setY(10);  
Point p2 = new Point();  
p2.x = p1.x;  
p2.y = p1.x + p1.y;  
p2.setX(p1.getX());  
p2.setY(p1.getX()+p1.getY());
```

- Pour accéder à leur valeur il faut passer par une méthode de type fonction.
- Pour les modifier il faut passer par une méthode de type procédure.

```
public class Point{  
    private int x ;  
    private int y ;  
  
    .....  
    .....  
  
    // idem pour x  
  
    public int getY(){  
        return y;  
    }  
    public void setY(int newY){  
        y = newY;  
    }  
}
```

Méthodes

Méthodes

- Les méthodes sont des fonctions qui implémentent les traitements de la classe.
- La syntaxe de la déclaration d'une méthode est :
`modificateurs type_retourné nom_méthode (arg1, ...) { ... }`
 - **Les modificateurs** assurent le contrôle de la visibilité de la méthode. Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe
 - **Le type retourné** peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void**.
 - **Le corps** d'une méthode contient des instructions comme la programmation classique.

Les modificateurs de méthode

Modificateur	Rôle
public	La méthode est accessible aux méthodes des autres classes.
private	L'usage de la méthode est réservé aux méthodes de la même classe.
protected	La méthode ne peut être invoquée que par les classes filles ou les classes du même package.
final	La méthode ne peut être modifiée (redéfinition lors de l'héritage interdite).
static	La méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est donc inutile d'instancier la classe pour appeler la méthode. Elle ne peut utiliser que des variables de classes.
synchronized	La méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	Le code source de la méthode est écrit dans un autre langage.

Types de Méthodes

- Parmi les différentes méthodes que comporte une classe, on a souvent tendance à distinguer :
 - **Les constructeurs** qui sont appelées au moment de la création d'une instance par l'opérateur **new**.
 - **Les méthode de classe** définit un comportement de la classe, un traitement qui peut être déclenché sans instance de la classe .
 - **Les méthodes accesseurs** (en anglais **getters**) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses attributs (généralement privés), sans les modifier .
 - **Les méthodes modificateurs** (en anglais **setters**) qui modifient l'état d'un objet, donc les valeurs de certains de ses attributs.

Constructeurs

- C'est une ou plusieurs méthode(s) permettant de créer et d'initialiser les objets.
- Le constructeur est appelé lors de la création de l'objet.
- Le constructeur a **le même nom que la classe**.
- Il n'a **pas de valeur de retour** (**void** est un type de retour !!!).
- Le constructeur peut être **surchargé**.
- Toute classe JAVA possède au moins un constructeur:
 - 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 int x ; // abscisse
    private int y ; // ordonnee
    public Point (int abs, int ord){
        x = abs ;
        y = ord ;
    }
    public int getX(){
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
    // idem pour y
    public void deplace (int dx, int dy){
        x += dx ;
        y += dy ;
    }
    public void affiche(){
        System.out.println ("Mes coordonnees " + x + " " + y);
    }
}
```

Définition explicite
du constructeur

Constructeurs multiple

- C'est **possible de définir plusieurs constructeurs** dans une même classe:
 - possibilité d'initialiser un objet de plusieurs manières différentes
- Une classe peut définir un nombre quelconque de constructeurs.
- 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
- On dit que les constructeurs peuvent être **surchargés** (overloaded).

Exemple de constructeurs

```
public class Point{
    private int x ; // abscisse
    private int y ; // ordonnee

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

    public Point () {
        this.x = this.y = 1 ;
    }

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

    .....
}
```

Appel d'un constructeur par un autre

- Dans les classes définissant plusieurs constructeurs, un constructeur peut invoquer un autre constructeur de cette classe.
- L'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.

Appel d'un constructeur par un autre

```
public class Point{
    private int x ;
    private int y ;
    public Point (int x, int y){
        this.x = x ;
        this.y = y ;
    }
    public Point (){
        this(1,1) ;
    }
    public Point (Point p){
        this(p.x,p.y) ;
    }
    .....
}
```

Surcharge des méthodes

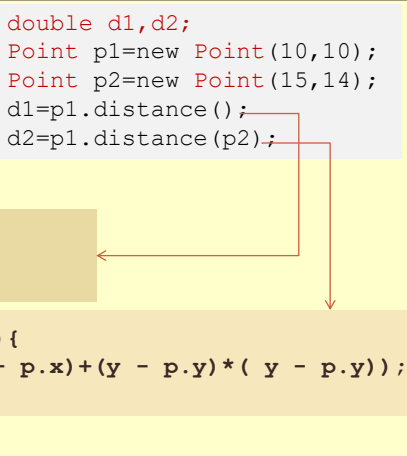
- La **surcharge** (overloading) n'est pas limitée aux constructeurs, elle est possible pour n'importe quelle méthode
- C'est possible de définir des méthodes **possédant le même nom mais dont les arguments diffèrent**.
- Lorsque qu'une méthode surchargée est invoquée le compilateur sélectionne automatiquement 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.
- Les méthodes surchargées peuvent avoir des **types de retour différents** mais à condition qu'elles aient des **arguments différents**.

Surcharge des méthodes (exemple)

```
public class Point{
    private int x ;
    private int y ;
    public Point (int x, int y){
        this.x = x ;
        this.y = y ;
    }
    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)) ;
    }
    .....
}
```

```
double d1,d2;
Point p1=new Point(10,10);
Point p2=new Point(15,14);
d1=p1.distance();
d2=p1.distance(p2);
```



Méthodes de classe

- Une **méthode de classe** est une méthode qui ne nécessite pas la création d'un objet, elle peut être appelée sans instance de la classe.
- On utilise là aussi le mot réservé **static**
- Une méthode de classe **ne peut accéder qu'aux attributs de la classe déclarés comme static**.
- L'appel d'une méthode de classe ne se fait **pas sur un objet**, mais **sur une classe**.
 - Exemple : **Math.cos(3.14);**

Exemple de méthode de classe (1/2)

```
public class A{
    .....
    private float x ;           // variable usuel
    private static int n ;     // variable de classe
    .....
    public static void f() // méthode de classe
    { ..... // ici, on ne peut pas accéder à x, champ usuel,
      ..... // mais on peut accéder a la variable de classe n
    }
}
```

```
A.f() ; // appelle la méthode de classe f de la classe A

A a = new A();
a.f() ; // reste autorisé, mais déconseillé
```

Exemple de méthode de classe (2/2)

```
public class MathUtil {  
    final static double PI = 3.14 ;  
  
    static double getPI() {  
        return PI;  
    }  
  
    static double carre(double x) {  
        return x * x;  
    }  
  
    static double demi(double x) {  
        return x / 2;  
    }  
}
```

```
double i = MathUtil.carre(5);  
double x = MathUtil.getPI();
```

Méthodes accesseurs / modificateurs

- L'encapsulation permet de sécuriser l'accès aux données d'une classe.
- Ainsi, les données déclarées **private** à l'intérieur d'une classe ne peuvent être **accédées** et **modifiées** que par des méthodes définies dans la même classe.
- Par convention, les accesseurs commencent par **get** et les modificateurs commencent par **set**.
- Pour un attribut de type **booléen**, il est possible de faire commencer l'accesseur en lecture par **is** au lieu de **get**.

Méthodes accesseurs / modificateurs

```
public class Point{  
    private int x ;  
    private int y ;  
    public Point (int x, int y){  
        this.x = x ;  
        this.y = y;  
    }  
    public int getX(){  
        return x;  
    }  
    public int getY(){  
        return y;  
    }  
    public void setX(int x){  
        this.x = x;  
    }  
    public void setY(int y){  
        this.y = y;  
    }  
}
```

Accesseurs

Modificateurs

Prof. Asmaa El Hannani

ISIC-S1

191

Utilisation d'une classe

Utilisation d'une classe

- L'utilisation d'une classe dans une application passe par trois étapes :
 - la déclaration d'une variable permettant l'accès à l'objet ;
 - la création de l'objet ;
 - l'initialisation d'une instance.

Utilisation d'une classe

```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;

    public Point(int abs, int ord) {
        x = abs; y = ord;
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
    ...
}

/** Teste la classe Point */
public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
}
```

Ecriture des classes

- Jusqu'à présent on a toujours dit :
 - une classe par fichier
 - le nom du fichier source : le nom de la classe avec extension **.java**
- En fait la vraie règle est :
 - une classe **public** par fichier
 - le nom du fichier source : le nom de la classe publique avec extension **.java**

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;
    public Point(int x1, int y1) {
        x = x1; y = y1;
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}
/** Teste la classe Point */
class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

Fichier:
Point.java

Compilation et exécution

- La compilation du fichier **Point.java**
 - **javac Point.java**
 - fournit 2 fichiers classes : **Point.class** et **TestPoint.class**
- On lance l'exécution de la classe **TestPoint** qui a une méthode **main()**
 - **java TestPoint**

2 classes dans 2 fichiers

```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;
    public Point(int x1, int y1) {
        x = x1; y = y1;
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}
```

Fichier:
Point.java

```
/** Teste la classe Point */
public class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

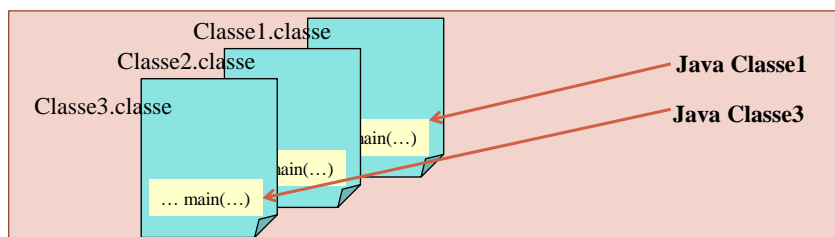
Fichier:
TestPoint.java

Compilation et exécution

- La compilation du fichier **Point.java**
 - **javac Point.java**
 - **javac TestPoint.java**
 - fournit 2 fichiers classes : **Point.class** et **TestPoint.class** (on peut aussi faire la compilation suivante d'un seul coup **javac *.java**)
- On lance l'exécution de la classe **TestPoint** qui a une méthode **main()**.
 - **java TestPoint**

Le main()

- Le point d'entrée pour l'exécution d'une application Java est la méthode statique **main** de la classe spécifiée à JVM.
- 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 classe indiquée à la JVM.



Exercice 4

- Réaliser une classe **Livre** permettant de représenter un Livre dans une bibliothèque.

Chaque Livre sera caractérisé par le *titre*, *l'auteur* et le *nombre de page*. On prévoira :

- Un constructeur recevant en arguments l'auteur et le titre,
- Une méthode **afficheLivre** imprimant (en fenêtre console) les information sur un livre,
- Des méthodes accesseurs / modificateurs.

Exercice 4 - suite

- Ecrivez une classe **TestLivre** dans un fichier TestLivre.java. Cette classe a une seule méthode main() qui se charge de:
 - Créer 2 livres,
 - Faire afficher ces 2 livres,
 - Donner le nombre de pages de chacun des 2 livres,
 - Calculer le nombre de pages total de ces 2 livres et affichez-le.
- Exécuter et tester votre implémentation.