Méthodes

- `visibilité` définit les droits d'accès à la méthode :
  - Une méthode privée (`private`) n'est accessible qu'à l'intérieur de la classe où elle est définie.
  - Une méthode publique (`public`) est accessible de n'importe quel point du programme.
  - Une méthode protégée (`protected`) n'est accessible qu'à l'intérieur de la classe où elle est définie, à partir des classes dérivées de celle-ci (quel que soit leur paquetage) et à partir de toutes les classes de son paquetage.
  - Si la visibilité n'est pas indiquée, la méthode est accessible *dans toutes les classes de son paquetage* : c'est un peu l'équivalent des méthodes `friend` de C++. Toutes les classes n'appartenant pas explicitement à un paquetage sont considérées comme appartenant à un paquetage « par défaut ».

Méthodes

- Le mot-clé `static`, s'il est présent, indique qu'il s'agit d'une *méthode de classe*, qui ne s'applique pas à une instance particulière. En ce cas, le membre est créé lors du chargement de la classe, indépendamment de la création d'une instance.
- Le type du résultat peut être `void` si la méthode ne renvoie pas de valeur, ou un type connu quelconque. Une méthode renvoie son résultat à l'aide de l'instruction `return`.
- Même si la méthode n'attend pas de paramètres, son nom doit être suivi d'une paire de parenthèses.
- Les paramètres sont toujours passés *par valeur*, c'est-à-dire que la méthode travaille sur des copies des paramètres réels, qui ne seront donc pas modifiés.
- À la différence de C++, Java n'autorise pas les paramètres par défaut (mais ils peuvent être simulés à l'aide de la surcharge de méthodes).

Surcharge de méthodes

- Plusieurs méthodes peuvent porter le même nom pourvu qu'elles puissent être distinguées à l'appel. On dit alors que la méthode est *surchargée* (overload).
- Lors de l'appel, Java distingue les méthodes surchargées en utilisant *uniquement la liste des paramètres* (qui doit donc être discriminante). *Le type du résultat et les qualificateurs ne sont pas pris en compte*.
- Java, contrairement à C++ et à C#, n'autorise pas la surcharge des opérateurs.

Exemple de surcharges

```
void afficher(int i) { ... }
void afficher(String ch) { ... }
void afficher(String ch, int largeur) { ... }
void afficher(String ch, int largeur, char mode) { ... }
void afficher(int nbFois, String ch) { ... }

// Exemples d'appels

afficher(10);                 // afficher(int)
afficher("Bonjour", 20, 'c'); // afficher(String, int, char)
afficher("Bonjour");          // afficher(String)
afficher(true);               // erreur de compilation
```


Passage des paramètres aux méthodes

- Les paramètres réels étant passés *par valeur* (on dit aussi *par copie* car ils sont *copiés* dans les paramètres formels au moment de l'appel), les méthodes ne peuvent pas les modifier.
- Lorsqu'un paramètre réel est une instance de classe (donc une *référence*), ce n'est pas l'objet réel qui est copié, mais cette référence : la méthode ne peut pas la modifier, mais peut s'en servir pour modifier l'objet (si celui-ci est modifiable).

Exemples

```
static void echanger(int x, int y) {
  int temp = x;
  x = y; y = temp;
}
static void echanger(int[] tab) {
  int temp = tab[0];
  tab[0] = tab[1]; tab[1] = temp;
}
(...)
int val1 = 10, val2 = 100;
int[] tab = {10, 100};

echanger(val1, val2);   // val1 et val2 non échangés
echanger(tab);          // tab[0] et tab[1] échangés
```

Nombre variable de paramètres

L'ellipse (...) indique un nombre variable de paramètres (éventuellement nul) *de même type* :

```
int somme(int ... n) {
  int res = 0;
  for (int val : n) // n est considéré comme un tableau
    res += val;
  return res;
}
(...)
System.out.println(somme(2, 3));         // 5
System.out.println(somme(1, 2, 3, 4));   // 10
System.out.println(somme());             // 0
```

Remarques

- Il ne peut y avoir qu'un seul paramètre variable dans une liste de paramètres et *il doit être le dernier*.
- Si l'on veut imposer un nombre minimum de paramètres, il suffit de faire précéder le paramètre variable du nombre de paramètres voulu :

```
int somme(int x, int y, int ... n) { ... }  // au moins 2 paramètres
```


Nombre variable de paramètres

Les méthodes utilisant des paramètres variables peuvent être surchargées comme les autres, mais attention aux ambiguïtés :

```
int somme(int x, int y)        { ... }
int somme(int ... vals)        { ... }
int somme(int x, int ... vals) { ... }
float somme(float ... vals)    { ... }
(...)
somme(2, 3);         // Pas d'ambiguïté : c'est la première qui est appelée
somme();             // Qui est appelée ?
somme(3);            // Qui est appelée ?
```

Paramètres de la fonction main()

- La fonction `main()` est particulière et utilise un mécanisme spécial pour récupérer les paramètres passés au programme via la ligne de commande.
- Le prototype `public static void main(String[] args)` indique que tous les paramètres passés au programme seront stockés dans le tableau de chaînes `args`.

Attention :

À la différence de C et C++, `args[0]` n'est pas le nom du programme mais le premier paramètre passé : `args.length` indique donc le nombre de paramètres passés au programme.

Exemple :

```
public class TestParams {
  public static void main(String[] args) {
    System.out.printf("Vous avez passé %d paramètre(s) au programme :%n", args.length);
    for (String param : args)
      System.out.println(param);
  }
}
```


Paramètres de la fonction main()

Cas des paramètres numériques :

- Les paramètres de la ligne de commande sont toujours des chaînes.
- Si le programme veut considérer l'un de ces paramètres comme un nombre, il devra donc le convertir (avec `parseXXX`, par exemple).

Exemple de paramètre numérique

```
class TestParams {

  static public void main(String[] args) {
    if ( (args.length != 1) || (Integer.parseInt(args[0]) < 2) ) {
      System.err.println("Usage : java TestParams nbre (avec nbre >= 2)");
      System.exit(1);
    }
    int limite = Integer.parseInt(args[0]);
    // Calcul des nombres premiers <= à limite...
    ...
  }
}
```

Remarque

*Ce code est **incorrect** car si le paramètre passé au programme n'est pas un entier, la méthode `parseInt()` lèvera une exception qui n'est pas gérée ici...*

Les exceptions

- Certaines méthodes ou opérations de Java peuvent produire des *exceptions* (accès en dehors d'un tableau, division par zéro, ouverture d'un fichier inexistant, etc.).
- Ces méthodes ou opérations doivent être traitées de façon spéciale pour traiter l'exception si elle survient, afin que que le programme ne se termine pas brutalement.

Remarque

- Il y a deux sortes d'exceptions : celles que l'on peut prévoir et celles que l'on ne peut pas éviter... On peut éviter d'accéder en dehors d'un tableau ou de diviser par zéro, par exemple, mais on ne peut pas éviter qu'un utilisateur tente d'ouvrir un fichier qui n'existe pas.
- La gestion des exceptions ne doit pas remplacer les bonnes pratiques de programmation, ni les tests dans les programmes, car leur traitement est beaucoup plus lourd qu'un simple test de variable...

Les exceptions

La gestion des exceptions utilise la syntaxe suivante :

```
try {
  code pouvant lever une exception
} catch (type_exception1 e) {
  traitement du type exception1
} catch (type_exception2 e) {
  traitement du type exception2
} ...
```

- Si le code de la clause `try` déclenche une exception, le code de la clause `catch` correspondant au type d'exception rencontré est exécuté avant de passer à la suite.
- Si un bloc de code lève une exception sans la capturer, celle-ci est *propagée* au bloc englobant.
- Si une méthode ne capture pas une exception, celle-ci est *propagée* à la méthode appelante, etc.
- Si l'exception n'est jamais capturée, elle est repercutée à la fonction `main()`. Si celle-ci ne la traite pas non plus, l'interpréteur met alors fin au programme en signalant l'erreur.

Les exceptions

Remarques

- Toutes les classes exceptions dérivent de la classe `Throwable`. Cette classe a deux classes filles, `Error` et `Exception`.
- Les exceptions de type `Error` sont considérées comme *irrécupérables* et il est donc inutile de tenter de les capturer.
- La classe `Exception` a deux classes filles prédéfinies : `IOException` et `RuntimeException`. Les exceptions de ce dernier type étant des erreurs de programmation (`ArrayIndexOutOfBoundException`, par exemple), elles ne devraient jamais se produire.
- Une clause `catch(Exception e)` capturera *toutes* les exceptions. Si elle est présente, elle doit donc apparaître *en dernier* puisque les clauses `catch` sont testées dans l'ordre de leur apparition et la première qui correspond s'exécute.

Les exceptions

- À partir de Java 7, la clause `catch` peut concerner plusieurs exceptions à la fois :

catch multiples

```
catch (ArithmeticException | ArrayOutOfBoundsException e) {
    ...
}
```

- L'objet `e` dispose d'une méthode `getMessage()` renvoyant le message d'erreur détaillé de l'exception :

```
...
catch (Exception e) {
  System.err.printf("Erreur %s", e.getMessage());
}
```

- Les clauses `catch` peuvent, éventuellement, être suivies d'une clause `finally {...}` qui sera exécutée *quelle que soit la façon dont on est sorti du* `try {...}`, que ce soit normalement ou à cause d'une exception. Cette clause permet donc d'écrire du code de nettoyage (fermeture des fichiers ou d'une connexion réseau, par exemple).

Les exceptions

- La documentation des méthodes et des opérations indique les exceptions qui sont susceptibles d'être levées.
- Si une méthode annonce qu'elle est susceptible de propager une exception (à l'aide de la clause `throws`), son appel doit se trouver dans une clause `try` et l'exception doit être traitée dans une clause `catch`.
- Si votre méthode ne capture pas elle-même l'exception, mais qu'elle se contente de la *propager*, vous devez l'indiquer avec une clause `throws` lors de la définition de la méthode (sauf pour les exception `Error` et `RuntimeException`).

Propagation d'une exception

```
void maMethode(String filename) throws IOException {
   /* code susceptible de lever une IOException
      qui n'est pas traitée dans cette méthode */
}
```


Les exceptions

- `java.lang` définit un grand nombre d'exceptions et on en trouve également notamment dans `java.io` et d'autres paquetages (l'idée est que chaque package définit les exceptions qui le concernent).
- Une exception de la classe `ClasseException` peut être levée explicitement dans le code en créant un nouvel objet de cette classe et en le signalant à l'aide de l'instruction `throw` :

Levée explicite d'une exception

```
...
if (...) throw new ClasseException(...); // appel du constructeur de ClasseException
```

- Nous verrons plus tard comment créer des classes d'exception personnalisées.

Les exceptions

On peut également exploiter les exceptions pour alléger le code :

```
// Nombres premiers inférieurs ou égaux à une limite
public static void main(String[] args) {
    int limite = 0;

   /* Cas d'erreur :
     pas de paramètre passé => args[0] provoque une exception
     paramètre non entier   => parseInt() provoque une exception
     entier <= 2            => on déclenche une exception
   */
    try {
      limite = Integer.parseInt(args[0]);
      if (limite <= 2) throw new NumberFormatException();
    } catch(Exception e) {
      System.err.println("Usage: java Premiers <limite> (avec limite > 2)");
      System.exit(1);
    }
    // Suite du programme...
}
```


Les chaînes de caractères

- Les chaînes de caractères sont des *objets* : ce sont des instances de la classe `java.lang.String`.
- Ces chaînes contiennent des caractères Unicode (UTF-16) et, à la différence des chaînes de C, *ne sont pas des tableaux de caractères terminés par 0*.
- On ne peut pas utiliser la notation entre crochets pour accéder à un caractère particulier d'une chaîne (il faut utiliser `charAt()`).
- La classe `String` fournit des méthodes permettant de créer une chaîne à partir d'un tableau de caractères et de récupérer dans un tableau de caractères le contenu d'une chaîne.
- La longueur d'une chaîne peut être connue en appelant la méthode `length()` (à ne pas confondre avec l'attribut `length` des tableaux...).

Les chaînes de caractères

- Le seul cas de surcharge d'opérateur Java est celui de l'opérateur `+` pour la concaténation de deux chaînes :

```
String str1 = "Bonjour",
       str2 = "les amis";
String str3 = str1 + str2;  // str3 contient "Bonjourles amis"
```

- La classe `String` est déclarée `final`, ce qui signifie qu'on ne peut pas créer de classe dérivée (voir la deuxième partie du cours).
- En outre, les objets `String` sont *immutables* : on ne peut pas les modifier. Toutes les instructions qui semblent modifier un `String` créent en fait *un nouvel objet* et ne modifient *pas* l'objet existant, ce qui a un impact non négligeable sur les performances et l'occupation mémoire.
- Pour créer des chaînes modifiables, on dispose des classes `StringBuffer` et `StringBuilder`.

Les chaînes de caractères

- Un objet `String` étant, en réalité, une *référence*, l'opérateur `==` revient donc à tester si deux références sont égales. Pour comparer *leurs contenus*, il faut utiliser les méthodes `equals()` ou `compareTo()` (deux références différentes peuvent désigner le même contenu).
- Les chaînes ne pouvant être modifiées et se comparant selon leur contenu impliquent qu'elles se comportent *quasiment comme des variables d'un type primitif*, bien que ce soient en réalité des objets de *type référence* : cela signifie notamment qu'elles sont allouées sur le tas, pas sur la pile.

Les chaînes de caractères : exemple

```
class ComparerString {
    public static void main(String[] args) {
        String str1 = new String("Bonjour"),
               str2 = new String("Bonjour");

        System.out.println(str1 == str2);        // False car deux références différentes
        System.out.println(str1.equals(str2));   // True car même contenu

        // Mais...

        String str3 = "Bonjour",
               str4 = "Bonjour";

        System.out.println(str3 == str4);        // True car le compilateur a "interné" Bonjour...
        System.out.println(str3.equals(str4));   // True car même contenu
    }
}
```

Remarque

Utilisez toujours `equals()` pour comparer les contenus des chaînes...

Les chaînes de caractères

- Un littéral chaîne (entre apostrophes doubles, comme en C) crée un nouvel objet `String`. Ces littéraux peuvent contenir des caractères spéciaux, comme `\t`, `\n`, `\r`, `\\`, `\"`.
- Ce littéral chaîne est « interné » : s'il existe déjà dans le programme, il n'est pas recréé (voir exemple plus haut).
- Les principales méthodes d'un objet de la classe `String` sont `charAt`, `endsWith`, `startsWith`, `equals`, `indexOf`, `lastIndexOf`, `replace`, `split`, `substring`, `toLowerCase`, `toUpperCase`, `trim`...
- Différents constructeurs permettent de construire un objet `String` à partir d'un tableau de `byte` ou de `char`, ou à partir d'un objet `StringBuffer` ou `StringBuilder`.
- Il existe également un constructeur « de copie », qui permet de créer un objet `String` à partir d'un objet `String` existant.
- La méthode `toString`, redéfinie dans toutes les classes prédéfinies, renvoie un objet `String` représentant l'objet concerné.
- Consultez la documentation de l'API pour en savoir plus.

La classe StringBuilder

- Le fait qu'un objet `String` ne soit pas modifiable a un impact non négligeable sur les performances lorsque l'on doit modifier cet objet.
- Si l'on doit modifier une chaîne de caractères, il est préférable d'utiliser un objet `StringBuilder` qui, lui, est modifiable (la classe `StringBuffer` est identique à `StringBuilder` mais intègre un mécanisme de synchronisation des accès pour la programmation multi-thread).
- Nous allons comparer trois exemples qui montrent le problème. Dans les 3 cas, on veut ajouter 50000 fois le caractère `'x'` à une chaîne initialement vide. Les 3 programmes se termineront en affichant la longueur de la chaîne finale (qui doit donc être égale à 50000).

String vs. StringBuilder

```
// Première version, avec un objet String
public class Str1 {
   public static void main(String[] args) {
     String ch = "";
     for (int i = 1; i <= 50_000; i++) { ch = ch + 'x'; }
     System.out.println(ch.length());  // 50000
   }
}

// Deuxième version, avec un objet StringBuilder
public class Str2 {
   public static void main(String[] args) {
     StringBuilder ch = new StringBuilder("");
     for (int i = 1; i <= 50_000; i++) { ch.append('x'); }
     System.out.println(ch.length());  // 50000
   }
}

// Troisième version, avec un objet StringBuilder et capacité initiale
public class Str3 {
   public static void main(String[] args) {
     final int TAILLE_MAX = 50_000;
     StringBuilder ch = new StringBuilder(TAILLE_MAX);
     for (int i = 1; i <= TAILLE_MAX; i++)  { ch.append('x'); }
     System.out.println(ch.length());  // 50000
   }
}
```

String vs. StringBuilder

Comparaison des temps d'exécution :

```
java Str1  1,27s user 0,36s system 102% cpu 1,597 total
java Str2  0,07s user 0,02s system 99% cpu 0,090 total
java Str3  0,07s user 0,02s system 98% cpu 0,089 total
```

- Le résultat se passe de commentaires...
- À chaque concaténation, la première version recrée un *nouvel* objet `String`, ce qui consomme à la fois du temps CPU et de l'espace mémoire puisque les anciens objets restent dans le tas jusqu'à ce qu'ils soient libérés par le ramasse-miettes.
- Les versions avec `StringBuilder`, en revanche, ne créent qu'*un seul* objet et le modifient *sur place* à chaque itération.
- La troisième version fixe la capacité de la chaîne : par défaut, cette capacité est de 16 caractères, ce qui implique une réallocation lorsque la chaîne dépasse 16 caractères. En indiquant dès le départ la capacité totale à allouer, on évite cette réallocation.

String vs. StringBuilder

- La méthode `compareTo` ne pouvant pas s'appliquer à un `StringBuilder`, il suffit de comparer les objets `String` correspondants, qu'on obtient par un appel à `toString`.
- Inversement, pour obtenir un `StringBuilder` à partir d'un objet `String` existant, il suffit d'utiliser le constructeur `StringBuilder` qui prend un `String` en paramètre.
- Pour copier un `StringBuilder` dans un autre `StringBuilder`, on peut appeler le constructeur précédent en le combinant avec `toString` :

```
public class TestSB {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("Coucou");
    StringBuilder sbClone = new StringBuilder(sb.toString());
    sbClone.setCharAt(0, 'B');              // Modification de la copie
    System.out.printf("Copie : %s%nOriginal : %s%n", sbClone, sb);
    if (sbClone.toString().compareTo(sb.toString()) < 0) {
      System.out.println("Devrait être vrai...");
    }
  }
}
```

POO en Java

Classes

- Une définition de classe est introduite par le mot-clé `class`.
- Une classe Java peut contenir :
  - des *champs* ou *attributs* (variables ou constantes)
  - des *méthodes*
- Un même fichier Java peut contenir plusieurs définitions de classe, mais une seule peut être publique (`public`) : celle qui porte le nom du fichier.
- La notion de classe publique est liée à la notion de *paquetage*.

Unités de compilation

- Un fichier source Java (`.java`) est une *unité de compilation*, qui peut contenir, au plus, une seule classe *publique* portant le même nom que le fichier, sans l'extension.
- Une unité de compilation peut également contenir plusieurs classes non publiques, inaccessibles de l'extérieur mais utilisables par la classe publique.
- La compilation d'une unité de compilation produit un fichier (`.class`) *pour chaque classe*, publique ou non.
- Un *programme* Java est formé d'un ensemble de fichiers `.class`, qui peuvent être archivés et compressés dans un fichier JAR (*Java ARchive*).
- Une *bibliothèque* est également un ensemble de fichiers `.class`, à raison d'une classe par fichier. Pour lier ces différents composants, on les intègre dans un même *paquetage*.

Paquetages

- Pour indiquer qu'une classe fait partie du paquetage `mon_paquetage`, on indique `package mon_paquetage;` au début de l'unité de compilation de cette classe.
- La classe n'est alors accessible qu'en accédant d'abord au paquetage.
- Une classe ne peut appartenir qu'à *un seul paquetage*.
- On place tous les fichiers `.class` d'un même paquetage dans un unique répertoire (ou dans un fichier JAR).
- Le nom d'un paquetage reflète la structure des répertoires : `java.lang`, `java.util`, `mon.paquetage.a.moi` correspondent, respectivement, aux répertoires *java/lang/*, *java/util/* et *mon/paquetage/a/moi/* (noms relatifs au « *classpath* »).

Paquetages (suite)

- Pour localiser les paquetages utilisateurs, le compilateur utilise la variable d'environnement `CLASSPATH` (l'emplacement des paquetages système y est automatiquement ajouté).
- `CLASSPATH` doit contenir une liste de fichiers ZIP, JAR ou de répertoires. Par exemple :
  ```
  export CLASSPATH=.:$USER/mesclasses:/usr/local/javatools/classes.zip
  ```
- L'option `-classpath` (ou `-cp`) de `javac` écrase cette variable et les emplacements des classes systèmes n'y sont pas ajoutés.
- En l'absence de l'instruction `package`, Java considère que les classes du fichier appartiennent à un paquetage « *par défaut* » : le code sera interprété relativement au répertoire courant (qui doit faire partie du *classpath*).

Utilisation d'une classe d'un paquetage

- Utilisation de son nom pleinement qualifié :
  ```
  java.util.ArrayList monTab;  // classe ArrayList du paquetage java.util
  ```
- Importation d'une classe :
  ```
  import java.util.ArrayList;  // En début de fichier
  ...
  ArrayList monTab;            // classe ArrayList du paquetage java.util
  ```
- Importation d'un paquetage (accès à toutes ses classes) :
  ```
  import java.util.*;          // En début de fichier
  ...
  ArrayList monTab;            // classe ArrayList du paquetage java.util
  Vector monVect;              // classe Vector du paquetage java.util
  ```
- `java.lang.*` est importé par défaut : il contient les classes de base, les classes `System`, `Math`, etc.

Intérêt des paquetages

- Permettent de définir des *espaces de noms* : deux paquetages différents peuvent fournir des classes de même nom sans qu'il y ait conflit.
- Les noms des paquetages suivent une convention empêchant les collisions : les fournisseurs utilisent des noms comme `com.sybase.jdbc` ou `netscape.javascript`, reposant sur leur nom de domaine Internet (donc unique).
- Malgré tout, les conflits qui pourraient subsister sont résolus en utilisant des noms pleinement qualifiés.
- Les archives JAR permettent de distribuer facilement les programmes (un seul fichier `.jar` au lieu d'un ensemble de fichiers `.class`). Exécution avec `java -jar monProjet.jar`.

Classes et objets

Un objet est une *instance* particulière d'une classe (*cf.* cours de POO). En Java, on utilise l'opérateur `new`, qui crée un nouvel objet sur le tas.

- La déclaration `MaClasse monObjet;` *ne crée pas* d'objet mais une *référence* (initialisée à `null`) vers un objet de la classe `MaClasse`.
- L'objet est créé dynamiquement par l'instruction `new MaClasse(...)`, qui effectue la réservation mémoire, appelle le constructeur avec des paramètres éventuels et renvoie une référence sur la zone réservée pour l'objet.
- En conséquence, on utilisera le plus souvent la syntaxe suivante :

Instanciation d'une classe

```
MaClasse monObjet = new MaClasse();
TaClasse tonObjet = new TaClasse("bonjour", 20);
```

Classes et objets

- Java gère automatiquement la récupération de la mémoire occupée par les objets devenus inaccessibles (*garbage collector*). Il n'y a pas de destructeurs en Java (mais on peut faire en sorte qu'un certain code soit exécuté au moment de la suppression de l'objet).
- Une classe définit donc des *attributs* et des *méthodes*. Certains s'appliquent à tous les objets de cette classe, quels qu'ils soient (attributs et méthodes *de classe*), la plupart s'appliquent à une instance particulière de cette classe (attributs et méthodes d'*instance*).
- Ces attributs et méthodes sont associés à des *règles de visibilité*, qui définissent le contexte de leur accès en dehors de la classe où ils sont définis.
- Une classe a elle-même des règles de visibilité par rapport au paquetage dans lequel elle est définie.

Définition de classe en Java

Fichier Point.java

```
public class Point {

   private int x, y;

   // Constructeurs
   public Point(int abs, int ord) { this.x = abs; this.y = ord; }
   public Point() { this.x = this.y = 0; }
   public Point(Point aPoint) { this.x = aPoint.x; this.y = aPoint.y; }

   // Accesseurs
   public void setX(int abs) { this.x = abs; }
   public void setY(int ord) { this.y = ord; }
   public int getX() { return this.x; }
   public int getY() { return this.y; }

   public void deplace(int deltaX, int deltaY) { this.x += deltaX; this.y += deltaY; }

   @Override
   public String toString() { return "(" + this.x + ", " + this.y + ")"; }

  /* Méthode de test de la classe. */
   public static void main(String[] args) {
     Point p1 = new Point();
     Point p2 = new Point(10, 15);

     p1.deplace(5, 20);
     System.out.println("p1 : " + p1 + " p2 : " + p2);
    } // main
} // class Point
```