Programmation objets, web et mobiles (JAVA)

Cours 1 - Fondamentaux

Licence 3 Professionnelle - Multimédia
Philippe Esling (esling@ircam.fr)
Maître de conférences – UPMC
Equipe représentations musicales (IRCAM, Paris)

Présentation de l'UE

- Site pédagogique habituel de la L3 Multimédia
- Equipe pédagogique
  - Responsable : **Philippe Esling**

esling@ircam.fr

- **Description de l'UE**
  - Concepts fondamentaux de la programmation objets (sûrete et réutilisabilite)
  - Maîtriser leur mise en pratique dans le cadre du langage et savoir utiliser un environnement de développement pour Java
  - Commencer les modèles de programmation et structures de données
  - Connaître des modèles classiques de programmation objets (Design Patterns)
  - Maîtriser les bases de la programmation graphique (2D et 3D / OpenGL)
  - Pouvoir développer une application sur mobile (Androïd / iPhone)
  - Modèle d'enseignement basé sur la réalisation de projets (sexys) et par l'apprentissage manuel (l'échec + Google)

Présentation de l'UE – *Plan I*

1. Présentation de l'UE et fondamentaux.
   ****Bataille de carte****
2. Énumération, héritage et polymorphisme.
   ****Lecteur de fichiers complexe factorisé****
3. Règles des surcharges et surdéfinition
   ****Types de données complexes et parcours****
4. Gestion des erreurs et architecture d'un projet
   ****GitHub / Pokedeck****
5. Interfaces Homme/Machine
   ****Pokedeck****
6. Design Patterns I
   ****Pokedeck****
7. Design Patterns II
   **** Pokedeck 2****
8. Séance projet (1)

Présentation de l'UE – *Plan II*

1. Modèles de programmation réseau et répartie
   ****Serveur de chat****
2. Programmation web
   ****Servlet réactif****
3. Programmation graphique
   ****Système planètes****
4. OpenGL et programmation jeux vidéo 3D
   ****Free game style****
5. Plateformes mobiles (Androïd, iPhone, …)
   ****SMS Bomber****
6. Plateformes mobiles (Androïd, iPhone, …)
   **** Funky SMS Bomber****
7. Concept avancé et Java 8
   ****Projet final****
8. Séance projet (2)

Logiciels de cours

Indépendants des systèmes (Windows, Linux, MacOSX)

Langage Java 1.7

- Pré-installé à l'UPMC (ex. ARI)
- Installer chez soi à partir du site d'Oracle

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Environnements de développement (**choix imposé**)

- Eclipse (**imposé cette année**)

  http://www.eclipse.org

- NetBeans, Visual Studio, JetBrains …

Choix okay mais possibilité de non-réponse aux questions

Présentation de l'UE – *Plan II*

Projet (soutenance en fin de semestre) : 40%

Projet par binôme - Choix parmi les sujets proposés

Soutenances individuelles (50%)

Evaluation du code, qualité, solutions (50%)

Evaluation « mi-parcours » (en classe) : 40%

Contrôle « continu » : 20%

Séances de TME : 70%

Par binôme - Plusieurs rendus intermédiaires

Participation : 30%

Sujets de projets libres à choisir

- Improvisateur musical par modélisation du style
- Prédiction de séries financières et économiques
- Jeu vidéo sur mobile (Androïd/iPhone)
- Sujet libre

Java ?

- Un langage : Orienté objet fortement typé avec classes
- Un environnement d'exécution (JRE) = machine virtuelle et ensemble de bilbiothèques
- Un environnement de développement (JDK) = machine virtuelle et ensemble d'outils
- Une mascotte moche : Duke

Java ?

Java est un des langages de programmation les plus utilisés

Incontournable dans plusieurs domaines :

- **Systèmes dynamiques**: Chargement dynamique de classes
- **Internet**: Web réactif, servlets
- **Systèmes répartis**: RMI, Corba, EJB, etc…

Java n'est pas un langage normalisé, il évolue constamment

Cette évolution se fait

- En ajoutant de **nouvelles API**
- En **modifiant la machine virtuelle**
- Géré par **le JCP** (*Java Community Process* http://www.jcp.org)

Nécessite d'identifier les version du JDK et de la MV … (relou)

Java ?

- Filiation historique:
  - **1983** (AT&T Bell): C++
  - **1991** (Sun Microsystems): Java
- Java est très proche du langage C++
- Toutefois Java simplifie le langage C++, car le but était justement de supprimer les points considérés *critiques* du C++ (ceux qui sont à l'origine des principales erreurs)
- Ainsi Java évite
  - Les pointeurs
  - La surcharge d'opérateurs
  - L'héritage multiple
  - (oui on comprends rien pour l'instant mais ça viendra ☺)

Java vs. C++

D'autres propriétés essentielles de Java

- **Tout est dynamique** (les instances d'une classe sont instanciées dynamiquement).
- **L'utilisation de la mémoire est transparente pour l'utilisateur.** Il n'y a pas besoin de spécifier de mécanisme de destruction, ou de gérer l'espace mémoire. En gros, un langage de glandeur pour la mémoire qui est gérée par le **garbage collector (ramasse-miettes) qui détecte les objets à détruire**

- Gain de fiabilité (et de flemme) car pas de désallocation fausse
- Mais forcément implique une perte de rapidité par rapport à C++

Java vs. C++ : Compilation

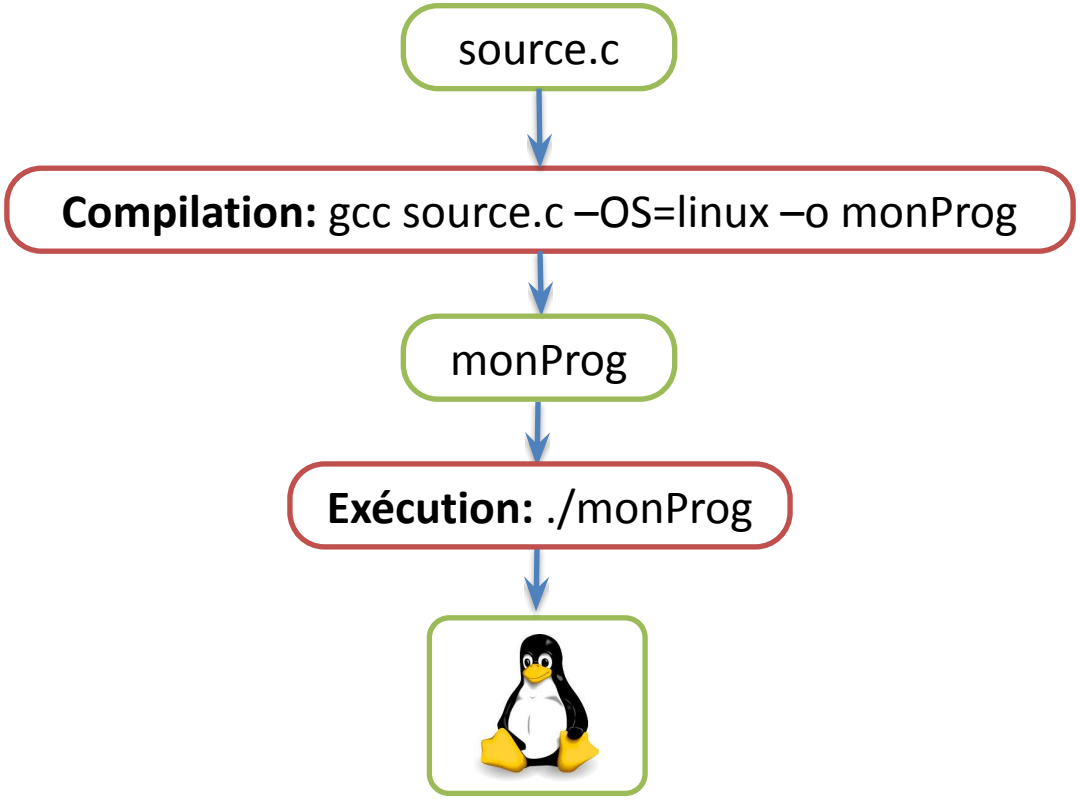
Généralement en C/C++

- Une fois le logiciel codé, on peut fournir les sources ou bien l'exécutable
- En entreprise, on souhaite protéger le code (notre travail)
- … Mais on veut que l'exécutable tourne sur toutes les architectures (processeur, OS, etc..)
- Il faut donc effectuer une compilation du code pour produire un exécutable propre à chaque environnement

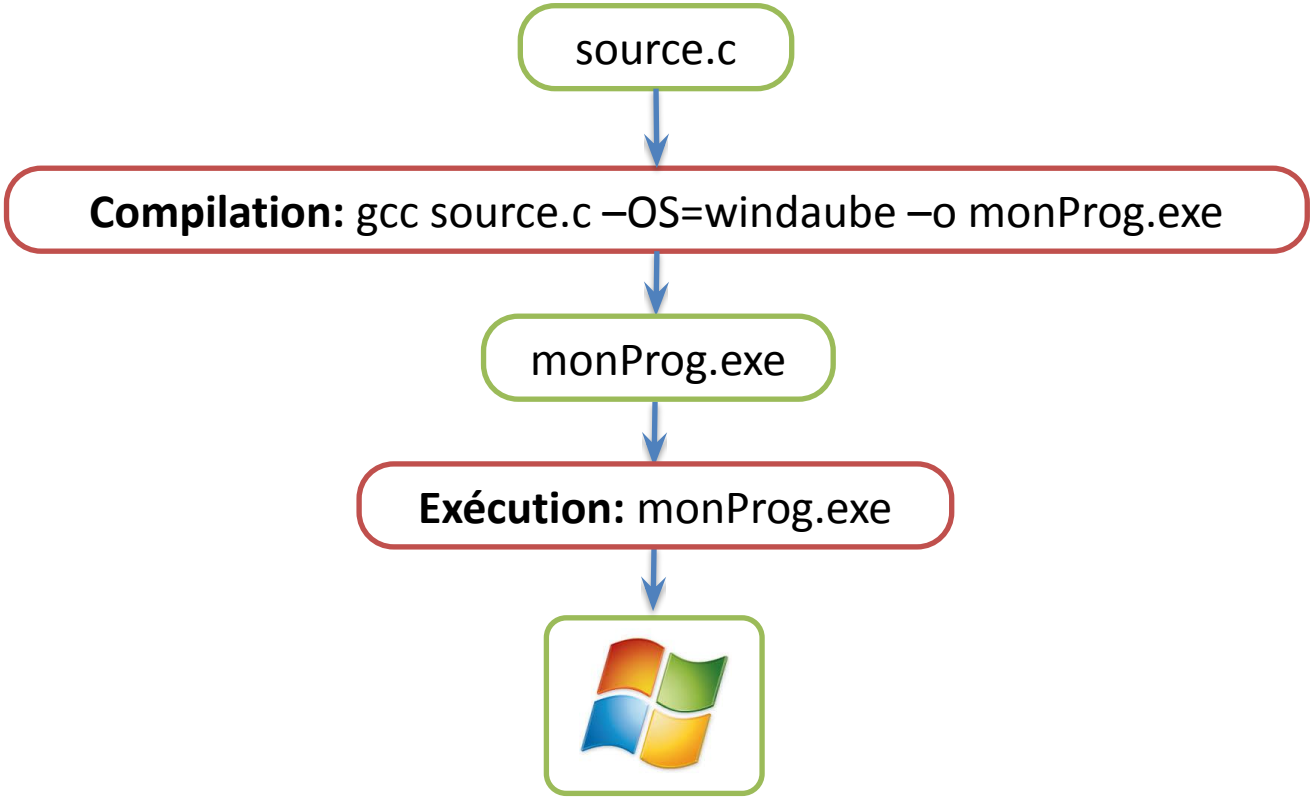
- **On doit donc créer un exécutable pour chaque type d'architecture potentielle …**
- **Et s'assurer que ça marche pour chaque architecture !**

(ie. Fonctions spécifiques aux OS, CPU, etc… relou)

Java vs. C++


source.c
Compilation: gcc source.c –OS=linux –o monProg
monProg
Exécution: ./monProg

Java vs. C++


source.c
Compilation: gcc source.c –OS=windaube –o monProg.exe
monProg.exe
Exécution: monProg.exe

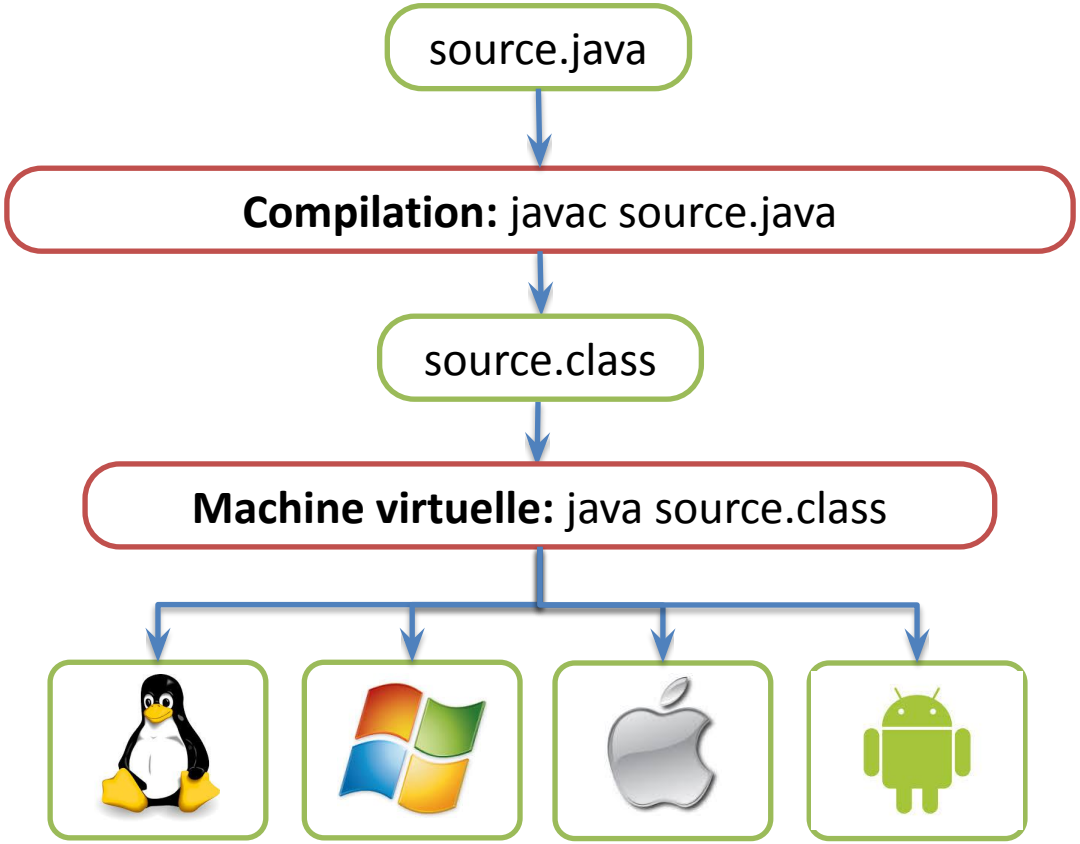
Java vs. C++

Alors qu'en Java …

- Le code n'est **pas traduit directement** dans le langage natif
- Il est d'abord traduit en **bytecode**
- Le bytecode est le **langage d'une machine virtuelle** (JVM)
- Les JVM (Java Virtual Machine) **se chargent d'interpréter le bytecode sur toutes les architectures**

- Le **bytecode** ne dépends pas de l'architecture où il a été compilé
- Les bytecodes d'une machine **pourront s'exécuter sur toutes les autres architectures (#glandeur)**
- Tout cela est possible grâce à l'utilisation de la **machine virtuelle**

Java vs. C++


source.java
Compilation: javac source.java
source.class
Machine virtuelle: java source.class

Java vs. C++

Le bytecode doit être exécuté par une **machine virtuelle**

Cette JVM est simulée par un programme qui:

1. Lit les instructions (bytecode) du programme .class
2. Fait une passe de vérification (type opérande, taille de pile, initialisations, …) pour s'assurer qu'il n'y a aucune action dangereuse
3. Fait plusieurs passes d'optimisation du code
4. Traduit dans le langage natif du CPU de l'ordinateur
5. Lance l'exécution

Java vs. C++

Les vérifications effectuées sur le bytecode et la compilation du bytecode vers le langage natif du CPU ralentissent l'exécution des classes Java

Il existe des techniques de compilation à la volée

Just In Time (JIT) et Hotspot

Qui réduisent ce problème en permettant de ne traduire qu'une seule fois en code natif les instruction qui sont (souvent) exécutées.

Java vs. C++

Le langage Java est

- « C-Like » : Syntaxe de base très similaire au C
- Orienté objet: Tout est objet, sauf les types primitifs (booléens, entiers, flottants, etc..)
- Robuste: Typage fort, pas de pointeurs
- Code intermédiaire: Le compilateur ne produit que du **bytecode** indépendant de l'architecture ou a été compilé le code source.

- Java perd (un peu) en efficacité par rapport à C++
- Mais gagne (beaucoup) en portabilité

Programmation objets, web et mobiles (JAVA)

Cours 1 - Fondamentaux

Programmation Orientée Objet (POO)

UPMC SORBONNE UNIVERSITÉS

ircam Centre Pompidou

Programmation ?

Le schéma simplifié d'un programme informatique peut se résumer par la formule

Programme = Structure de données + Traitements

Programmation ?

Le schéma simplifié d'un programme informatique peut se résumer par la formule

Programme = Structure de données + Traitements

Le cycle de vie d'un programme se décompose en 2 grandes phases

- Une phase de **production** (réaliser le logiciel)
- Une phase de **maintenance** (corriger et faire évoluer le logiciel)

Programmation ?

Le schéma simplifié d'un programme informatique peut se résumer par la formule

Programme = Structure de données + Traitements

Le cycle de vie d'un programme se décompose en 2 grandes phases

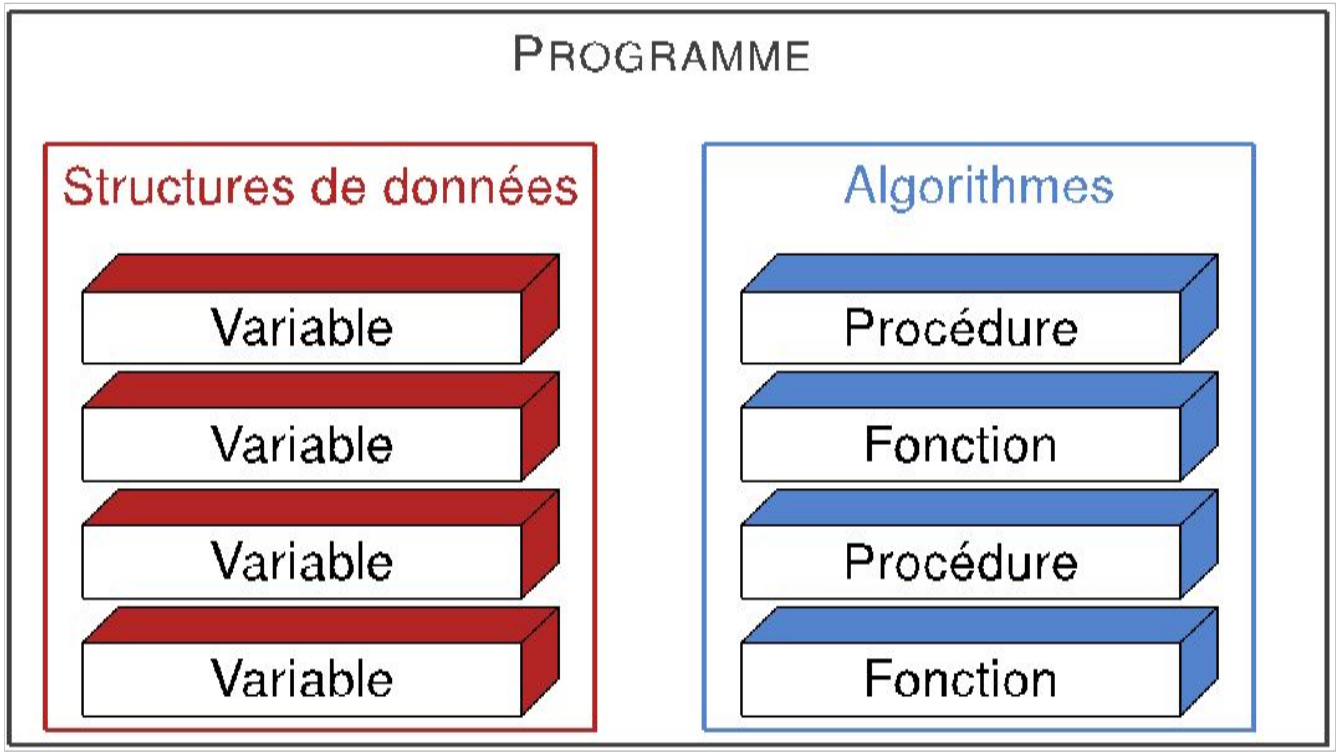
- Une phase de **production** (réaliser le logiciel)
- Une phase de **maintenance** (corriger et faire évoluer le logiciel)

Lors de la production d'un logiciel (au sens industriel), le concepteur à deux grandes options

- Orienter sa conception sur **les traitements**
- Orienter sa conception sur **les données**

Programmation par traitements

Principe habituel (non-objet) : On sépare les données des moyens de traitement de ces données

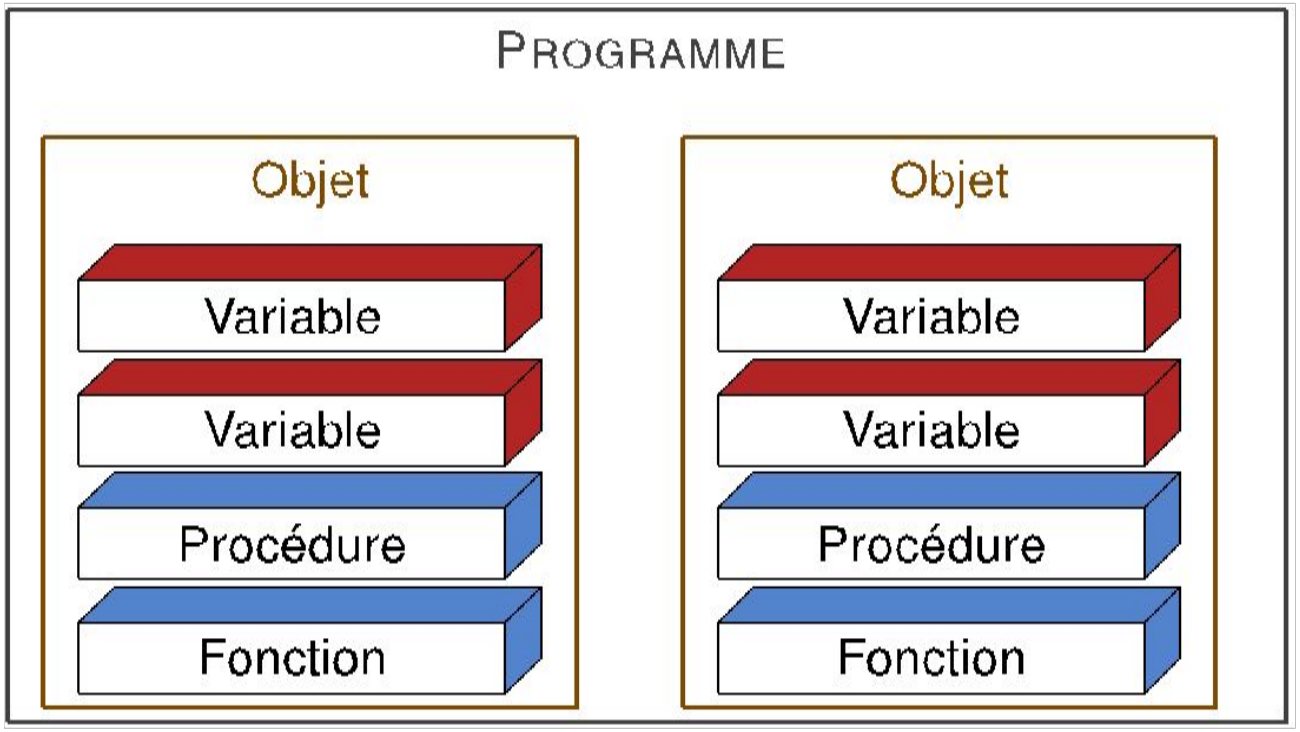


Programmation par traitements

- Premières conceptions de systèmes informatiques ont adopté cette approche (OS, XWindow, gestion, bureautique, calcul, etc…)
- De nombreux systèmes encore développés selon cette approche
- Systèmes *ad-hoc*, adaptés au problème de départ, mais **maintenance très difficile**
- Les traitements sont généralement **beaucoup moins stables que les données** (changement de spécification, ajout de fonctionnalités, etc…)
- Les structures de données sont **choisies en relation étroite** avec les traitements à effectuer

Programmation par objets

Principe: Pour avoir une architecture plus robuste, il semble logique de s'organiser autour des données manipulées.



Programmation par objets

- La construction du système s'axe **d'abord sur la détermination des données** (1er temps) et la réalisation des traitements (haut-niveau) dans un second temps

- Approche permet de bâtir des systèmes plus simples à maintenir et faire évoluer

- On regroupe dans une même entité informatique appelée **objet**, les données et les traitements sur ces données

Programmation par objets

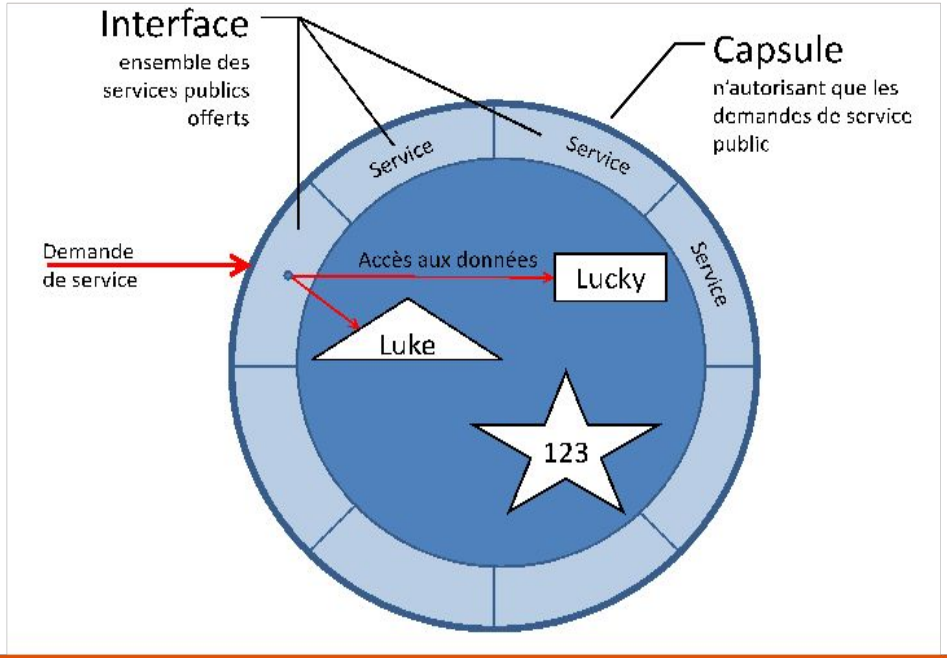
Définition :
Un **objet** est une entité autonome, qui regroupe un ensemble de propriétés (données) et de traitements associés à ces données.

A retenir
Ne commencez pas par vous demander *ce que fait* l'application mais plutôt **ce qu'elle manipule**

Programmation par objets

- Les structures de données définies dans l'objet sont appelés ses **attributs (propriétés)**
- Les procédures et fonctions définies dans l'objet sont appelés ses **méthodes (opérations)**
- Les attributs et méthodes d'un objet sont appelés ses **membres**
- L'ensemble des valeurs des attributs d'un objet à un instant donné est appelé **état interne**

Protection de l'information : **Encapsulation**



Règle

Les données d'un objet (**son état**) peuvent être lues ou modifiées uniquement par les services proposés par l'objet lui-même (**ses méthodes**)

Encapsulation - Définition

Définition :

Le terme **encapsulation** désigne le principe consistant à cacher l'information contenue dans un objet et ne proposer que des méthodes de modification/accès à ces données

- L'objet est vu de l'extérieur comme une **boîte noire** ayant certaines propriétés et ayant un comportement spécifié
- La manière dont le comportement a été implémenté est cachée aux utilisateurs de l'objet

Intérêt

Protéger la structure interne de l'objet contre toute manipulation produisant une erreur (**protection anti-boulets**)

Encapsulation - Définition

L'encapsulation nécessite la spécification de **parties publiques et privées** de l'objet

Elements publics

Correspondent à la partie visible de l'objet depuis l'extérieur. Ensemble de méthodes utilisables par d'autres objets

Elements privés

Partie non visible de l'objet depuis l'extérieur. Constitué d' éléments de l'objet visibles uniquement de l'intérieur de l'objet et de la définition des méthodes

Notion de classe

Pour être intéressante, la notion d'objet doit permettre un **degré d'abstraction = notion de classe**

Notion de classe

Pour être intéressante, la notion d'objet doit permettre un **degré d'abstraction = notion de classe**

Définition - classe

On appelle **classe** la structure d'un objet, i.e., la déclaration de l'ensemble des membres qui composeront un objet

Notion de classe

Pour être intéressante, la notion d'objet doit permettre un **degré d'abstraction = notion de classe**

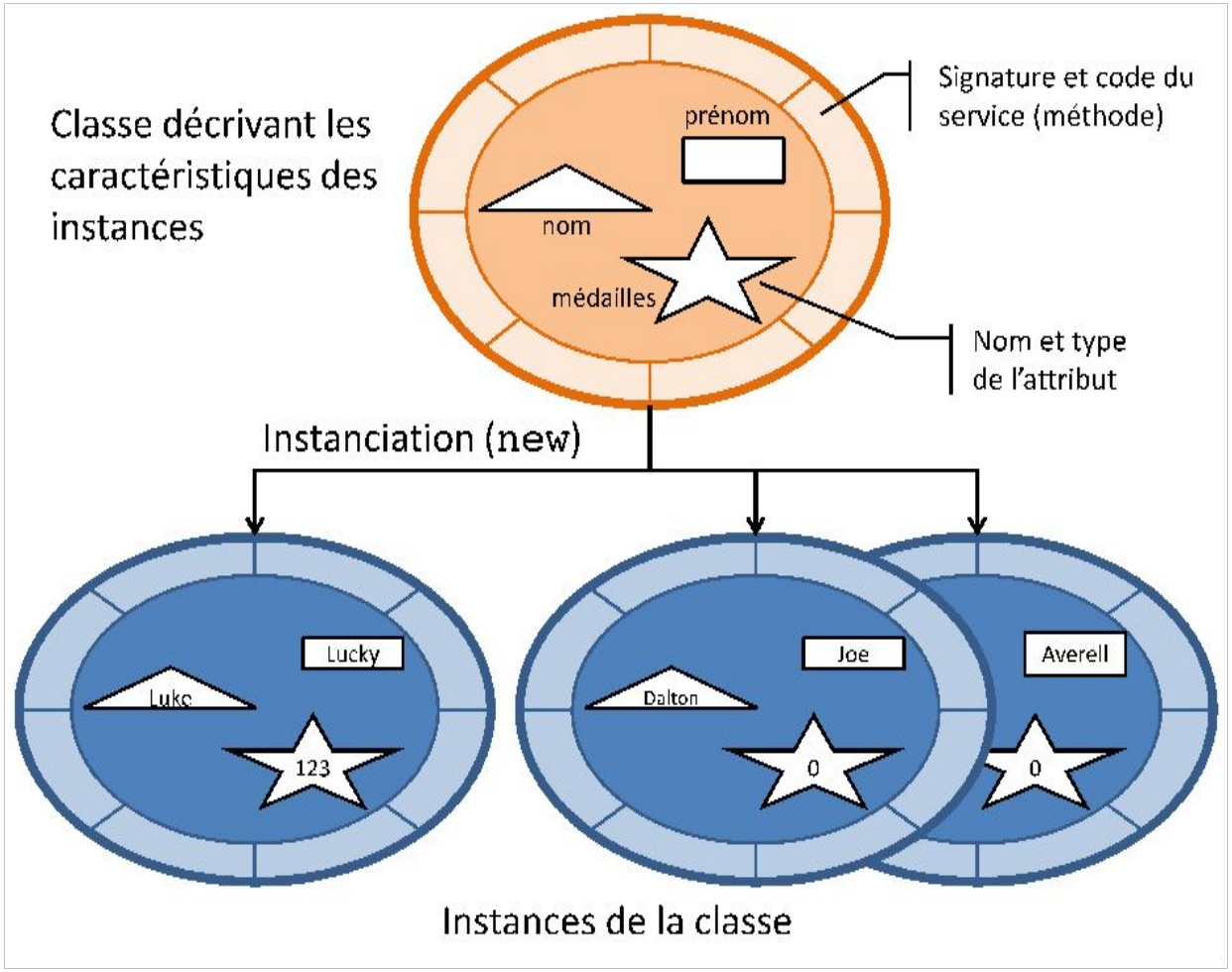
Définition - classe

On appelle **classe** la structure d'un objet, i.e., la déclaration de l'ensemble des membres qui composeront un objet

Définition - instance

La classe peut être vue comme un moule (plan de fabrication) pour la création des objets, qu'on appelle alors des **instances de classe**

Classe = modèle d'objets



Classe = modèle d'objets

Il est important de savoir les différences entre les notions de **classe** et **instance de la classe**

Classe = modèle d'objets

Il est important de savoir les différences entre les notions de **classe** et **instance de la classe**

classe = attributs + méthodes + mécanisme de création (instanciation) + mécanisme de destruction

Classe = modèle d'objets

Il est important de savoir les différences entre les notions de **classe** et **instance de la classe**

classe = attributs + méthodes + mécanisme de création (instanciation) + mécanisme de destruction

instance de la classe = valeur des attributs + accès aux méthodes

Classe = modèle d'objets

Il est important de savoir les différences entre les notions de **classe** et **instance de la classe**

classe = attributs + méthodes + mécanisme de création (instanciation) + mécanisme de destruction

instance de la classe = valeur des attributs + accès aux méthodes

L'**instanciation** est le mécanisme qui permet de créer des instances dont les traits sont décrits par la classe

La **destruction** est le mécanisme permettant de détruire une instance de classe

L'ensemble des instances d'une classe constitue l'**extension de la classe**

Visibilité

En Java, l'encapsulation est assurée par un ensemble de modificateurs d'accès permettant de préciser la visibilité des membres de la classe

Définition

Un membre dont la déclaration est précédée par le mot clé **public** est visible depuis toutes instances de toutes classes

Définition

Un membre dont la déclaration est précédée par le mot clé **private** n'est visible que depuis les instances de la classe

Remarque : Il existe d'autres types de visibilité pour un membre que nous n'évoquerons pas pour l'instant

Visibilité

Le mot clé **public** permet d'indiquer les services qui sont accessibles à l'utilisateur.

```
public class Question {
   ...
   public void repondre(int reponse) {
      ...
   }

   public boolean reponseCorrecte() {
      ...
   }

   public String toString() {
      ...
   }
...
}
```

Visibilité

Le mot clé **private** assure l'encapsulation

```
public class Question {
  // intitulé de la question
  private String question;
  // tableau des choix possibles
  private String[] choix;
  // numéro de la réponse de l'étudiant
  private int reponse;
  // numéro de la bonne réponse
  private int bonneReponse;

  public void repondre(int reponse) {
    ...
  }
  ...
}
```

Accès aux membres

Définition

L'ensemble des méthodes d'un objet accessibles de l'extérieur (depuis un autre objet) est appelé ***interface****. Elle caractérise le comportement de l'objet.*

Définition

*L'accès à un membre d'une classe se fait au moyen de l'****opérateur*** *« . ».*

Définition

L'invocation d'une méthode d'interface est appelé ***appel de méthode****. Il peut être vu comme un envoi de message entre objet.*

Accès aux membres

```
public class Point {
  public double x,y ;
}

public class Rectangle {
  public double longeur,largeur ;
  public Point coin ;
  public Point calculerCentre();
}

public class TestFigure {
  public static void main(String[] args) {
    Rectangle rect ;
    Point coinDuRect = rect.coin ;
    double xDuCoinDuRect = coinDuRect.x ;
    Point centreDuRect = rect.calculerCentre() ;
    double yDuCentreDuRect = centreDuRect.y ;
  }
}
```

Accès aux membres

```
public class Point {
  public double x,y ;
}

public class Rectangle {
  public double longeur,largeur ;
  public Point coin ;
  public Point calculerCentre();
}

public class TestFigure {
  public static void main(String[] args) {
    Rectangle rect ;
    double xDuCoinDuRect = rect.coin.x ;
    double yDuCentreDuRect = rect.calculerCentre().y ;
  }
}
```

Accès aux membres

Quelque soit le niveau de visibilité, on distingue **deux types de membres**

Définition

Un **membre de classe** est un membre **commun à toutes** les instances de la classe et existe dès que la classe est définie et **indépendamment de toute instanciation = mot clé static**

Définition

Un membre qui n'est pas de classe (qui n'est pas précédé du mot-clé static) est dit **membre d'instance**. Chaque instance d'une classe possède **son propre exemplaire** d'un attribut d'instance de la classe

Accès aux membres

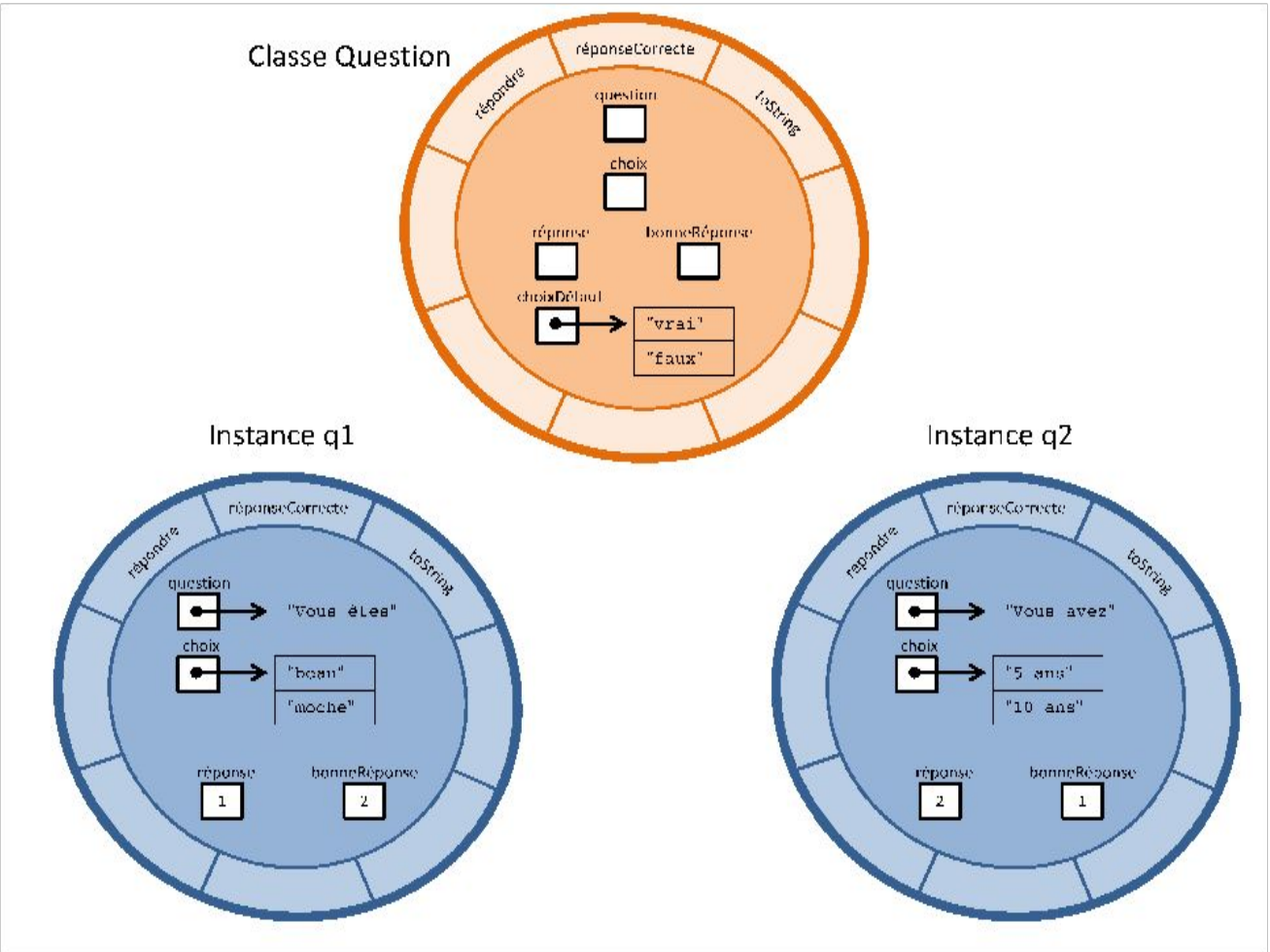
On veut introduire des choix par défaut

```
public class Question {
  ...
  private String[] choixDéfaut = {"vrai", "faux"};
  ...
}
```

Problème : chaque instance dispose de son propre tableau `choixDéfaut` alors qu'il devrait être commun à tous les objets

```
public class Question {
  ...
  private static String[] choixDéfaut
                          = {"vrai", "faux"};
  ...
}
```

Accès aux membres



Accès aux membres

Les membres de classes d'une classe donnée étant communs à toutes les instances de la classe, l'accès à un membre de classe se fait en appliquant l'opérateur « . » sur le nom de la classe.

```
System.out.println(Question.choixDefaut[1]);
```

L'accès aux membres de classe peut aussi se faire avec une instance de la classe suivie de l'opérateur « . ». Mais ceci est peu lisible et à n'utiliser que pour le polymorphisme (voir cours suivant).

```
Question q1 = new Question();
System.out.println(q1.choixDefaut[1]);
```

Accès aux membres

Une classe `java types` contient trois grands types de membres :

```
public class Point {
  // (1) Attributs
  private double x,y;
  // (2) Constructeurs
  public Point(double x, double y) {...}
  // (3) Méthodes
  public double distanceAvec(Point p2) {...}
}
```

Remarque(s)

Les constructeurs sont des méthodes particulières, ils seront donc introduits après celles-ci dans ce cours. Cependant, vous devez **toujours déclarer les constructeurs après les attributs et avant les autres méthodes** (voir ci-dessus).

Mini-projet : Géométrie

Faire un programme :

1. Créer une classe *PointMain* qui contiendra le **main**
2. Créer une classe *Point*
    1. Contient un constructeur pour créer un point
    2. Contient une fonction *moveTo* pour déplacer le point
    3. Contient une fonction r*MoveTo* pour déplacer le point (par addition)
    4. Contient une fonction *distance* qui compare à un autre point
3. Ecrire le programme principal
    1. Crée quatres points.
    2. Effectue des opérations et comparaisons entre ces points
    3. Affiche les résultats (**System.out.println**)
4. (Optionnel) Ajouter la classe *Rectangle*

    (on pourra par exemple créer une méthode pour calculer son périmètre, sa surface, etc.)

45 minutes - Noté

Types de données

Le type d'une variable en Java peut être :

- **Types dits primitifs : int, double, boolean**, etc.
- **nom d'une classe :** par exemple, les chaînes de caractères sont des instances de la classe **String**.

Remarque(s)

Nous laissons de côté pour l'instant le cas des tableaux sur lesquels nous reviendrons plus tard.

Types de données

Type	Taille (en bits)	Exemple
byte	8	1
short	16	345
int	32	-2
long	64	2L
float	32	3.14f, 2.5e+5
double	64	0.2d, 1.567e-5
boolean	1	true ou false
char	16	'a'

Attention

Un attribut de type primitif n'est pas un objet !

Expressions **similaires au C**

```
1  expression :=  variable
2        |  operateur expression
3        |  expression operateur
4        |  expression operateur expression
5        |   ( expression )
6        |   (type de base) expression
7        |  expression ? expression
8                     : expression
9        |  variable = expression
10       |  variable operateur= expression
```

exemple d'expressions :

```
1  z=18                              x = x + 1
2  x += 1                            x++
3  (byte)i                           (z % 3) << 2
4  (x<2) ? 2.7 : 3.14
5  (x != null) || (y >= 3.14)
6  (b = !b) & (( i ^= ~i) == 0)
```

Expressions: **tableaux**

- Déclaration

```
1  instruction ::= type [] variable;
2                | type [] variable  = {value ...} ;
```

- Création :

```
1  expression ::= new type [taille];
```

- Allocation dynamique de l'espace (mot clé new)
- Valeur **null** pour un tableau non alloué
- Vérification des indices - indice à partir de 0
- Pas de dimension multiple : utilisation de tableaux de tableaux
- Déclenchement d'une exception si indice incorrect

Expressions: **instructions**

```
1  instruction ::= { instruction; ... }
2      | if ( bool) {instructions};
3      | if ( bool ) {instructions}
4        else {instructions}
5      | L: while (bool) {instructions}
6      | L: do {instructions} while (bool);
7      | L: for (expressions;
8                bool;
9                expressions)
10             {instructions}
11     | L:switch (expressions)
12           {case constant : intruction;
13             ... default : instruction;}
14
15 continue label;
16 continue;
17 break label;
18 break;
```

Expressions: **instructions**

```
1
2  for (i=1; i < j ; i++) {
3    if  (i*i > j ) break;
4
5  }
```

```
1  i=1;
2  while (i < j) {
3    if  (i*i > j ) break;
4    i++;
5  }
```

```
1  switch (x % 5) {
2    case 0: { ... ; break;}
3    case 1: case 3: { ...; break;}
4    default: {... }
5  }
```

Syntaxe des classes

grammaire de déclaration d'une classe :

```
1  [ modifieur ]* class la_classe [ extends la_superclasse ] {
2    // champs ou attributs
3    [modifieur]* type champ [ = expression] ;
4    // constructeurs
5    [modifieur]* la_classe ( [type variable]*) {
6         instructions // sans return
7    }
8    // me'thodes
9    [modifieur]* TypeRetour methode ([Type variable]*)
10                    [throws [exception]+] {
11              instructions // avec return si type de retour
12   }
13 }
```

modifieurs :

```
1  public protected private abstract static final strictfp
```

Syntaxe des classes

```
3  public class Point {
4    // attributs
5    private double x, y;
6    // constructeurs
7    public Point(double a, double b){x=a;y=b;}
8    public Point(){x=0;y=0;}
9    // accesseurs
10   public double getX(){return x;}
11   public double getY(){return y;}
12   // methodes
13   private void moveto (double a, double b){x=a;y=b;}
14   public void rmoveto (double dx, double dy){x+=dx;y+=dy;}
15   public double distance(){return Math.sqrt(x*x+y*y);}
16   public double distance(Point p2){
17     double dx = p2.getX() - getX();
18     double dy = p2.getY() - getY();
19     return Math.sqrt(dx*dx + dy*dy);
20   }
21   // me'thodes pre'de'finies (standards)
22   public String toString(){return ("("+x+","+y+")");}
23 }
```

Syntaxe des classes

- création d'objets :
  `new Constructeur ( [argument[,argument]*])`
- appel de méthode :
  `expr . methode ([argument[,argument]*])`

```
1  package pobj.cours1;
2
3  public class ExPoint {
4    public static void main(String[] args) {
5      Point p0 = new Point();
6      Point p1 = new Point(3,4);
7      System.out.println(p0 + " - " + p1);
8      p0.rmoveto(7,12); p1.rmoveto(5,6);
9      System.out.println(p0 + " - " + p1);
10     if (p0.distance() == p1.distance())
11       System.out.println("c'est le hasard");
12     else
13       System.out.println("on pouvait parier");
14   }
15 }
```

Exemple de classe prédéfinie

Les chaînes de caractères sont instances de la classe **String**

L'opérateur de concaténation est l'opérateur +

Pour comparer deux chaînes de caractères, on utilise la méthode `equals` (ou `equalsIgnoreCase`) de la classe `String`.

```
String str1 = ....;
String str2 = ....;

if (str1.equals(str2)) {...} else {...}
```

Exemple de classe prédéfinie

La classe `String` offre de nombreuses autres possibilités :

- `length()` renvoie la longueur de la chaîne de caractères.
- `toUpperCase` et `toLowerCase` permettent, respectivement, de mettre la chaîne de caractères en lettres majuscules et minuscules.
- `int indexOf (char ch)` renvoie l'indice de la première occurence du caractère `ch` dans la chaîne de caractères.
- `String substring(int beginIndex,int endIndex)` qui retourne la sous-chaîne constitué des caractères d'indice `beginIndex` à `endIndex - 1`.
- etc ...

Point d'entrée du programme

Une des classe **doit contenir la fonction :**

```
1  public static void main(String[] args) ...
```

Cette fonction sert de point d'entrée du programme (début)

```
1  package pobj.cours1;
2
3  public class ExPoint {
4    public static void main(String[] args) {
5      Point p0 = new Point();
6      Point p1 = new Point(3,4);
7      System.out.println(p0 + " - " + p1);
8      p0.rmoveto(7,12); p1.rmoveto(5,6);
9      System.out.println(p0 + " - " + p1);
10     if (p0.distance() == p1.distance())
11       System.out.println("c'est le hasard");
12     else
13       System.out.println("on pouvait parier");
14   }
15 }
```

Point d'entrée du programme

- compilation :

```
1  > javac pobj/cours1/ExPoint.java
```

- exécution :

```
1  > java pobj/cours1/ExPoint
2  (0.0,0.0) - (3.0,4.0)
3  (7.0,12.0) - (8.0,10.0)
4  on pouvait parier
5  >
```

Mini-projet : Bataille de cartes

Faire un programme :

1. Créer une classe *Bataille* qui contiendra le **main**
2. Créer une classe *Carte*
    1. Contient un constructeur pour créer une carte
    2. Attributs : *couleur* (parmi un tableau statique) et *valeur* (idem)
    3. Méthode : accesseurs, modificateurs et comparateur
3. Créer une classe *Joueur*
    1. Propriétés : tableau de cartes et compteur de points
    2. Méthode : tire une carte et ajoute une carte
4. Ecrire le programme de jeu principal
    1. Crée deux joueurs.
    2. Initialise un paquet de cartes et le mélange aléatoirement (**Math.random**)
    3. Effectue une boucle de jeu en affichant les scores (**System.out.println**)
    4. Affiche le vainqueur de la partie

45 minutes - Noté