

2i002 - Fonctionnement de l'UE

Vincent Guigue
vincent.guigue@lip6.fr



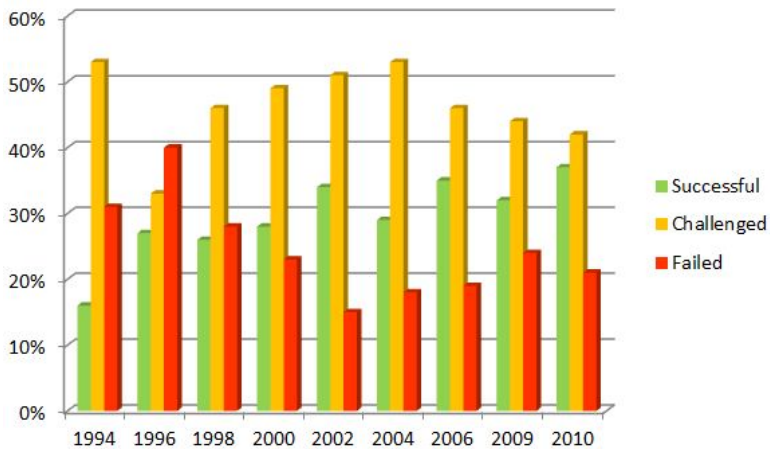
ORGANISATION / EVALUATION

- 1h45 Cours : base de cours + premier tutoriel
- 1h45 TD :
 - Suite tutoriel
 - Reflexion avancée sur les concepts de la semaine
- 1h45 TME : tester ses reflexes sur machine
- Pas de partiel
- Exam = 50 % de la note finale
- CC = 50% dont :
 - TME solo = 20%
 - Interro de TD = 20%
 - **Projet = 10%** (nouveauauté 2017)

2i002 - Introduction

Vincent Guigue





Pourquoi faire de la programmation objet ?

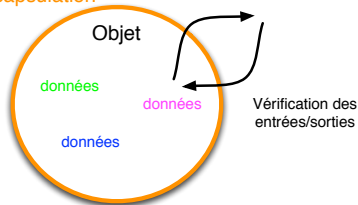
- Pour développer des systèmes complexes... *Sans se planter*

- **diviser** le système complexe en une multitude de systèmes simples : **les objets**
- **sécuriser** l'accès aux données sensibles

Barrière de sécurisation

=

encapsulation



Pourquoi faire de la programmation objet ?

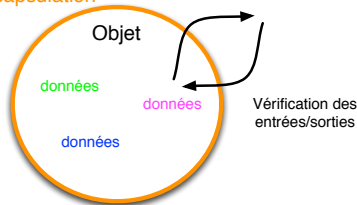
- Pour développer des systèmes complexes... *Sans se planter*

- **diviser** le système complexe en une multitude de systèmes simples : **les objets**
- **sécuriser** l'accès aux données sensibles

Barrière de sécurisation

=

encapsulation



- [corollaire] Travailler à plusieurs... *Sans se planter*
 - toujours penser son programme pour les autres : **sécuriser, simplifier, compartimenter**
 - double vision : client/fournisseur

Liste des fournitures (gratuites)

- **JDK** : Java Development Kit, SE (Standard Edition)

- <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- contient les outils pour compiler et exécuter du JAVA

- **Editeur de texte** : en fonction de goûts/habitudes

- Emacs (performants mais pas simple)

linux gedit (simple et efficace)

windows notepad++ (efficace)

- **IDE** (integrated development environment)

- Netbeans ou Eclipse
- Très performant, agréable, efficace... Même un peu trop au début !
- ⇒ ne pas utiliser avant la séance 5 pour assimiler les bases

DOCUMENTATION

Une documentation très bien faite est accessible en ligne pour toute la base de JAVA :

<https://docs.oracle.com/javase/8/docs/api/>

The screenshot shows the Java Platform, Standard Edition 8 API Specification website. The page is titled "Java™ Platform, Standard Edition 8 API Specification". Below the title, it says "This document is the API specification for the Java™ Platform, Standard Edition." and "See: Description".

The "Profiles" section lists three profiles: compact1, compact2, and compact3.

The "Packages" section is a table with two columns: "Package" and "Description".

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.

Java est un langage moderne qui puise son inspiration de sources diverses :

- une syntaxe très proche du C/C++
- une architecture dynamique avec un compilateur et une JVM

Java évolue régulièrement (nouvelle bibliothèque IHM dans JAVA5, lambda calcul dans JAVA8...)

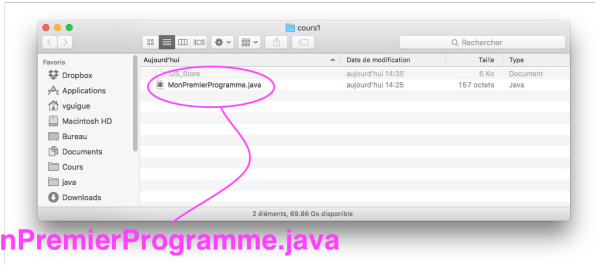
2i002 - Mon premier programme

Vincent Guigue



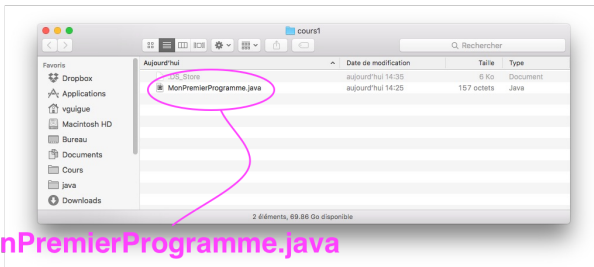
UNE CLASSE, UN MAIN

- ① En JAVA, tout code doit être encapsulé dans une classe :
- **1 classe = 1 fichier** du même nom
 - les noms de classe commencent par une majuscule



UNE CLASSE, UN MAIN

- 1 En JAVA, tout code doit être encapsulé dans une classe :
 - **1 classe = 1 fichier** du même nom
 - les noms de classe commencent par une majuscule



- 2 Un programme principal = un **point d'entrée** dans un système avec de nombreuses classes
 - ce programme est exécutable après compilation

UNE CLASSE, UN MAIN : SYNTAXE

La syntaxe des **signatures de classe** et de la **signature du main** est à apprendre par coeur (explications dans les cours suivants).

① Signature d'une classe (toujours public)

```
1 // dans le fichier MonPremierProgramme.java
2 public class MonPremierProgramme{
```

UNE CLASSE, UN MAIN : SYNTAXE

La syntaxe des **signatures de classe** et de la **signature du main** est à apprendre par coeur (explications dans les cours suivants).

② Signature d'un main

- toujours **public static void**
- toujours **main**
- toujours le même argument **String[] args**

```
1 // dans le fichier MonPremierProgramme.java
2 public class MonPremierProgramme{
3     public static void main(String [] args) {
```

UNE CLASSE, UN MAIN : SYNTAXE

La syntaxe des **signatures de classe** et de la **signature du main** est à apprendre par coeur (explications dans les cours suivants).

③ Instruction d'affichage dans la console

- pour pouvoir vérifier que le programme fonctionne

```
1 // dans le fichier MonPremierProgramme.java
2 public class MonPremierProgramme{
3     public static void main(String[] args) {
4         System.out.println("Bonjour!");
5     }
6 }
```

COMPILATION/EXÉCUTION

- Pré-requis :
 - JDK (Java Dev. Kit) installé sur la machine
 - Etre dans le bon répertoire (!)

① Compilation

- Vérification de la syntaxe, droits d'accès...
- Création d'un exécutable en bytecode :
`MonPremierProgramme.class`

```
» javac MonPremierProgramme.java
```

② Exécution

- Exécution du code dans la console

```
» java MonPremierProgramme (pas d'extension)
```

⇒ Résultat :

```
» Bonjour !
```


2i002 - Mon premier objet

Vincent Guigue

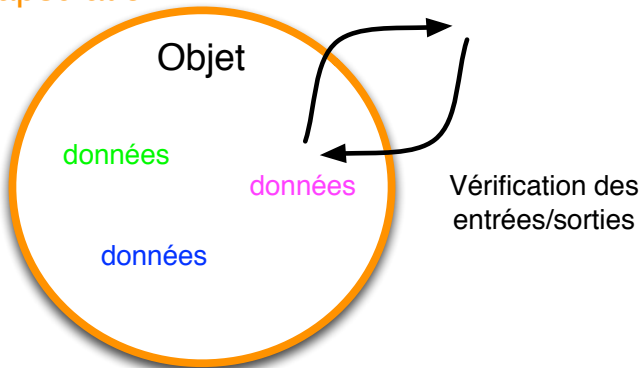


- Diviser un programme complexe en **objets** :
 - objet autonome : **réutilisable** dans plusieurs projets
 - Vecteurs, Personne, DisplaySimu...
 - objet **sécurisé** : garantie de bon usage par d'autre
 - un objet intègre des *données* et des *méthodes* pour les manipuler proprement,
 - les transactions bancaires sont journalisées, les éléments d'une simulation physique ne se téléportent pas...
 - objet **simple & intuitif** : le client ne voit que ce qui est nécessaire
- Enjeux :
 - **Réfléchir en amont** au découpage & à la sécurisation
 - **Documenter** le code (en premier lieu, respecter les conventions pour faciliter la compréhension)

Barrière de sécurisation

=

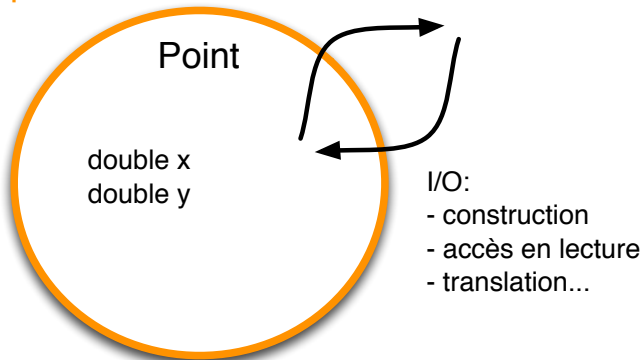
encapsulation



Barrière de sécurisation

=

encapsulation

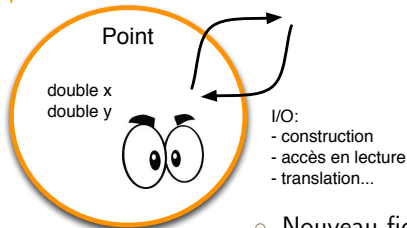


Point de vue fournisseur :

Barrière de sécurisation

=

encapsulation



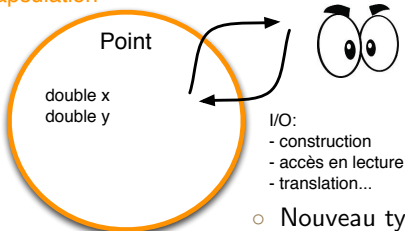
- Nouveau fichier, nouvelle classe
 - Point.java
- Comment construire un objet ?
 - e.g. J'attends 2 valeurs réelles
 - je mets à jour x et y
- Comment établir un dialogue (IO) ?
 - je définis une méthode pour observer x

Point de vue client :

Barrière de sécurisation

=

encapsulation



- Nouveau type de variable
 - `int i; Point p;`
- Comment construire un objet ?
 - Je donne 2 valeurs réelles + syntaxe
 - `p = new Point(1.2, 3.1);`
- Comment dialoguer ?
 - J'utilise le `.` pour accéder à l'interface de l'objet :
 - `p.getX();`

SYNTAXE : SIGNATURE ET ATTRIBUTS

① Fichier : 1 classe = 1 fichier

- nom de la classe + .java = nom du fichier
- marquer l'encapsulation, faciliter la ré-utilisation
- le nom de classe commence par une majuscule

```
1 // Création du fichier Point.java
```

SYNTAXE : SIGNATURE ET ATTRIBUTS

- 1 Fichier : 1 classe = 1 fichier
 - nom de la classe + .java = nom du fichier
 - marquer l'encapsulation, faciliter la ré-utilisation
 - le nom de classe commence par une majuscule
- 2 Signature : une classe est toujours public (en 2i002)

```
1 // Création du fichier Point.java
2 public class Point{ // classe publique
```


SYNTAXE : SIGNATURE ET ATTRIBUTS

- ① Fichier : 1 classe = 1 fichier
 - nom de la classe + .java = nom du fichier
 - marquer l'encapsulation, faciliter la ré-utilisation
 - le nom de classe commence par une majuscule
- ② Signature : une classe est toujours public (en 2i002)
- ③ Déclaration des attributs :
 - Répondre à : *De quoi est composé notre objet ?*
 - Les attributs sont presque toujours private (cf plus loin)
 - nom des attributs en minuscules

```
1 // Création du fichier Point.java
2 public class Point{ // classe publique
3     private double x,y; // attributs privés
```

SYNTAXE : SIGNATURE ET ATTRIBUTS

① Fichier : 1 classe = 1 fichier

- nom de la classe + .java = nom du fichier
- marquer l'encapsulation, faciliter la ré-utilisation
- le nom de classe commence par une majuscule

② Signature : une classe est toujours public (en 2i002)

③ Déclaration des attributs :


- Répondre à : *De quoi est composé notre objet ?*
- Les attributs sont presque toujours private (cf plus loin)
- nom des attributs en minuscules

④ Définir des méthodes

- Comment construire un objet ?
- Quelles opérations effectuer sur l'objet ?
- nom des méthode en minuscules

```
1 // Création du fichier Point.java
2 public class Point{ // classe publique
3     private double x,y; // attributs privés
```

Comment construire un Point ?

 Attention à considérer le problème des 2 points de vues


Fournisseur

- *Besoin* : 2 coordonnées fournies en argument
- *Action* : affectation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

SYNTAXE : CONSTRUCTEUR

Comment construire un Point ?

 Attention à considérer le problème des 2 points de vues

Fournisseur

- *Besoin* : 2 coordonnées fournies en argument
- *Action* : affectation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

Client


ie : utilisateur d'une classe Point existante...

- *Besoin* : créer une instance dans la mémoire avec les bonnes valeurs

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String [] args){
```

SYNTAXE : CONSTRUCTEUR

Comment construire un Point ?

 Attention à considérer le problème des 2 points de vues

Fournisseur

- *Besoin* : 2 coordonnées fournies en argument
- *Action* : affectation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

Client

ie : utilisateur d'une classe Point existante...

- *Besoin* : créer une instance dans la mémoire avec les bonnes valeurs

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5         // appel du constructeur
6         // avec des valeurs choisies
7         Point p = new Point(2., 3.1);
8     }
9 }
```

AUTORISATIONS D'ACCÈS

- **public** : accessible / visible depuis l'extérieur de l'objet (eg : un main, un autre objet...)
- **private** : protégé / invisible depuis l'extérieur de l'objet
- Les constructeurs sont en général **public**, ils ont vocation à être appelés depuis l'extérieur
- Les attributs sont en général **private**, ils sont protégés et non accessibles depuis l'extérieur

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // opération autorisée
7         Point p = new Point(2., 3.1);
8
9         // opération impossible :
10        // ERREUR DE COMPILATION
11        double d = p.x;
12    }
13 }
```

SYNTAXE : MÉTHODES

Comment manipuler un Point ?

① définir des méthodes

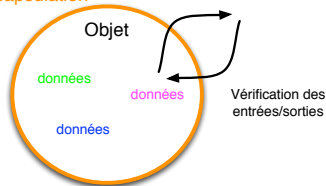
eg : accéder aux attributs (en lecture)

② les invoquer depuis l'extérieur

Barrière de sécurisation

=

encapsulation



Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     public Point(double x2, double y2){
5         x = x2;
6         y = y2;
7     }
8
9     public double getX(){
10         return x;
11     }
12 }
13 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String [] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8
9         double px = p.getX();
10
11         System.out.println(
12             "coord_x de p: "+px);
13     }
14 }
```

SYNTAXE : SURCHARGE DU CONSTRUCTEUR

Comment construire un Point ? ... de plusieurs manières !

Ex :

- 2 valeurs à fournir : le plus classique
- 0 valeur : génération aléatoire de x et y
- 1 valeur : affectation de la même valeur pour x et y

SYNTAXE : SURCHARGE DU CONSTRUCTEUR

Comment construire un Point ? ... de plusieurs manières !

Ex :

- 2 valeurs à fournir : le plus classique
- 0 valeur : génération aléatoire de x et y
- 1 valeur : affectation de la même valeur pour x et y

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
```

SYNTAXE : SURCHARGE DU CONSTRUCTEUR

Comment construire un Point ? ... de plusieurs manières !

Ex :

- 2 valeurs à fournir : le plus classique
- 0 valeur : génération aléatoire de x et y
- 1 valeur : affectation de la même valeur pour x et y

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs

CONTRAINTE : signatures des constructeurs différentes

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9     public Point(double d){ // surcharge
10        x = d;
11        y = d;
12    }
13    public Point(){ // autre surcharge
14        // aléatoire entre 0 et 10
15        x = Math.random()*10;
16        y = Math.random()*10;
17    }
18 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8         // construction d'un autre point:
9         Point p2 = new Point();
10        // construction d'un 3e point:
11        Point p3 = new Point(4.2);
12
13    }
14 }
```

SYNTAXE : MÉTHODES STANDARDS

- Les méthodes standards existent sur tous les objets (cf cours héritage)... Mais le comportement n'est pas satisfaisant
- Ex : conversion d'un objet en chaîne de caractères `public String toString()`

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     ...
4
5 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8
9         String str = p.toString();
10
11         System.out.println("p:" + str);
12     }
13 }
```

```
» p: Point@8764152
```

SYNTAXE : MÉTHODES STANDARDS

- Les méthodes standards existent sur tous les objets (cf cours héritage)... Mais le comportement n'est pas satisfaisant
- Ex : conversion d'un objet en chaîne de caractères `public String toString()`

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     ...
4     public String toString(){
5         return "["+ x +", "+ y +"]";
6     }
7 }
```

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String [] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8
9         String str = p.toString();
10
11         System.out.println("p: " + str);
12     }
13 }
```

» p: [2, 3.1]

REFLEXION SUR LA SYNTAXE OBJET

Exemple type : addition entre 2 Point

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

REFLEXION SUR LA SYNTAXE OBJET

Exemple type : addition entre 2 Point

Réfléchir à la signature d'une méthode add permettant d'additionner 2 instances de Point (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des x et y des attributs des opérandes)

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String [] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8         Point p2 = new Point(0.5, 1);
9
10        Point p3 = p.add(p2);
11    }
12 }
```

REFLEXION SUR LA SYNTAXE OBJET

Exemple type : addition entre 2 Point

Réfléchir à la signature d'une méthode add permettant d'additionner 2 instances de Point (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des x et y des attributs des opérandes)

Client

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5
6         // construction d'un point:
7         Point p = new Point(2., 3.1);
8         Point p2 = new Point(0.5, 1);
9
10        Point p3 = p.add(p2);
11    }
12 }
```

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     public Point(double x2, double y2){
5         x = x2;
6         y = y2;
7     }
8
9     public Point add(Point p){
10        return new Point(x+p.x, y+p.y);
11    }
```

Syntaxe objet = un truc à prendre... Pas évident au début !

Définition

- Même nom de fonction, arguments différents
- Le type de retour ne compte pas

```
1 public class Point {  
2     ...  
3     public void move(double dx, double dy){  
4         x+=dx; y+=dy;  
5     }  
6     public void move(double dx, double dy, double scale){  
7         x+=dx*scale; y+=dy*scale;  
8     }  
9     public void move(int dx, int dy){  
10        x+=dx; y+=dy;  
11    }  
12    public void move(Point p){  
13        x+=p.x; y+=p.y;  
14    }
```

Rappel : dans la classe Point, vous avez accès aux attributs privés des autres instances de Point

Nous avons vu précédemment comment compiler et exécuter *UNE* classe... **Comment faire maintenant qu'il y en a plusieurs ?**

```
» javac Point.java  
» javac TestPoint.java  
» java TestPoint
```

Syntaxe réduite :

```
» javac Point.java TestPoint.java OU javac *.java  
» java TestPoint
```

Remarque : il peut y avoir **plusieurs main**
(mais pas plus de 1 par classe)

- Compilation de tous les main d'un coup
- Execution d'un seul (appel à la classe correspondante)

CAS AMBIGUS : this.

Un argument de méthode et un attribut portent le même nom : il faut les distinguer !

Exemple le plus classique : le constructeur

```
1 public class Point{
2     private double x,y; // attributs
3     public Point(double x, double y){ // arguments du constructeur
4         // distinguer l'attribut et l'argument
5         //pour faire l'affectation dans le bon sens
```

CAS AMBIGUS : this.

Un argument de méthode et un attribut portent le même nom : il faut les distinguer !

Exemple le plus classique : le constructeur

```
1 public class Point{
2     private double x,y; // attributs
3     public Point(double x, double y){ // arguments du constructeur
4         // distinguer l'attribut et l'argument
5         //pour faire l'affectation dans le bon sens
6
7         this.x = x; // utiliser l'argument pour init. l'attribut
8         this.y = y;
9     }
10 }
```

Vous avez toujours le droit d'utiliser la syntaxe `this.attr` pour désigner l'attribut `attr` dans la classe (même dans les cas non ambigus)

Vous pouvez utiliser `this.maMethode()` pour invoquer `maMethode` lorsque vous êtes dans une autre méthode de l'objet.

CONSTRUCTEUR MULTIPLE : USAGE DU this()

Approche standard pour écrire plusieurs constructeurs dans une classe :

- 1 Ecrire le constructeur général, prenant le plus d'arguments
 - 2 Appeler le constructeur 1 avec des arguments spécifiques
- ⇒ éviter les copier-coller, fiabiliser le code

```
1 public class Point{
2     private double x,y; // attributs
3
4     public Point(double x, double y){ // constructeur 1
5         this.x = x;
6         this.y = y;
7     }
8
9     public Point(){ // constructeur 2
10        // ATTENTION : aucun code avant this()
11        this(Math.random()*10, Math.random()*10); // invocation
12                                                    // du constructeur 1
13    }
14 }
```

2i002 - Types de base, variables, boucles, conditionnelles

Guide de survie en JAVA

Vincent Guigue



TYPES DES VARIABLES DE BASE EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en JAVA avec les opérateurs les plus courants.

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`

TYPES DES VARIABLES DE BASE EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en JAVA avec les opérateurs les plus courants.

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`



La plupart des types et syntaxes associées sont comparables au C/C++... **Sauf le booléen.**

Le booléen vaut `true`/`false` et n'est pas convertible en entier

TYPES DES VARIABLES DE BASE EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en JAVA avec les opérateurs les plus courants.

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`



La plupart des types et syntaxes associées sont comparables au C/C++... **Sauf le booléen.**

Le booléen vaut `true`/`false` et n'est pas convertible en entier

- Déclaration

```
1 int i; // déclaration de i
2 System.out.println(i); // => 0
3 double d = 2.6;
4 boolean b = true; // ou false
5 char c = 'a';
```


TYPES DES VARIABLES DE BASE EN JAVA

- Entier, réel, booléen, caractère : ces types sont disponibles de base en JAVA avec les opérateurs les plus courants.

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`



La plupart des types et syntaxes associées sont comparables au C/C++... **Sauf le booléen.**

Le booléen vaut `true`/`false` et n'est pas convertible en entier

- Déclaration

```
1 int i; // déclaration de i
2 System.out.println(i); // => 0
3 double d = 2.6;
4 boolean b = true; // ou false
5 char c = 'a';
```

```
1 // opérations de base: + - / * ...
2 int j = i+2;
3 int k = 1/2; // = 0 Attention a la division entiere
```

Gestion des chaînes de caractères

String n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à JAVA et son caractère immuable la rapproche très nettement d'un type de base.

```
1 String s = "toto"; // création d'une chaîne de caractères
2 s = s + "vaàlafac";
3 System.out.println(s); // affichage de s dans la console
```

Gestion des chaines de caractères

String n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à JAVA et son caractère immuable la rapproche très nettement d'un type de base.

```
1 String s = "toto"; // création d'une chaine de caracteres
2 s = s + "vaàla fac";
3 System.out.println(s); // affichage de s dans la console
```




Ne pas confondre l'**objet** String et l'affichage dans la console.

Gestion des chaines de caractères

String n'est pas un type de base, c'est un objet qui se comporte différemment des types de base... Mais c'est une classe complètement intégrée à JAVA et son caractère immuable la rapproche très nettement d'un type de base.

```
1 String s = "toto"; // création d'une chaine de caracteres
2 s = s + "vaàla fac";
3 System.out.println(s); // affichage de s dans la console
```

 Ne pas confondre l'**objet** String et l'affichage dans la console.

Les **possibilités sont nombreuses** : extraction de sous-chaines (substring), division en plusieurs chaines (split), recherche de caractères, construction de nouvelles chaines à partir d'expressions régulières (replace)... Toute la documentation sur : <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

2 choses à retenir sur les String

- 1 **Les chaînes sont immutables** : modifier une chaîne existante est impossible, il faut créer une nouvelle chaîne qui est une modification de l'ancienne. Cela rend la classe peu efficace dans certains cas... Et il faut alors se tourner vers des objets plus évolués (StringBuffer notamment)
- 2 **Ne pas utiliser == avec les String** mais toujours la méthode `.equals`. Les deux versions compilent mais la première donnera régulièrement des résultats faux (que nous expliquerons plus tard).

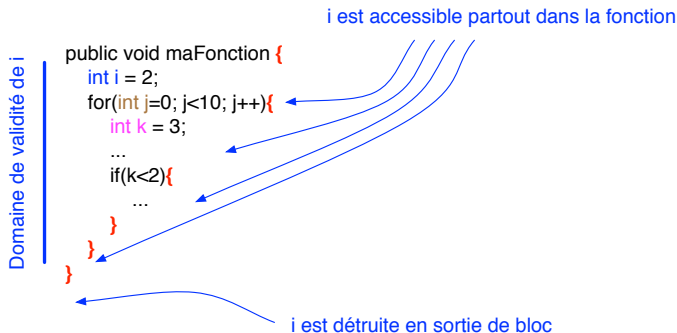
```
1 String s1 = "toto";
2 String s2 = "titi";
3 if( s1.equals(s2) )
4     System.out.println("les chaînes sont identiques");
5 else
6     System.out.println("les chaînes sont différentes");
```

DURÉE DE VIE

Logique de bloc

- une fonction est un bloc,
- une boucle ou une conditionnelle forme également un bloc,
- les blocs sont repérés par des accolades : `{...}`

Les variables **déclarée** dans un bloc sont détruites en sortant du bloc.



Logique de bloc

- une fonction est un bloc,
- une boucle ou une conditionnelle forme également un bloc,
- les blocs sont repérés par des accolades : `{...}`

Les variables **déclarée** dans un bloc sont détruites en sortant du bloc.

```
public void maFonction {  
    int i = 2;  
    for(int j=0; j<10; j++){  
        int k = 3;  
        ...  
        if(k<2){  
            ...  
        }  
    }  
}
```

CONVERSIONS ENTRE TYPES

JAVA, un langage typé

Les types sont très importants en JAVA : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont **implicites** :

```
1 double d = 1; double d2 = i; // avec i un int existant
```

Il est possible de transformer n'importe quel type de base en String (l'affichage est donc facile)

```
1 String s = "mon_message_"+1.5+"_"+d;
```


CONVERSIONS ENTRE TYPES

JAVA, un langage typé

Les types sont très importants en JAVA : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont **implicites** :

```
1 double d = 1; double d2 = i; // avec i un int existant
```

Il est possible de transformer n'importe quel type de base en String (l'affichage est donc facile)

```
1 String s = "mon_message"+1.5+" "+d;
```

- Certaines conversions doivent être **explicites**

```
1 int i = (int) 2.4;
```

il y a une perte d'information liée à la conversion ; JAVA ne tolère pas la conversion implicitement, il faut que le programmeur la demande explicitement (pour être sûr que la perte d'information est souhaitée).

CONVERSIONS ENTRE TYPES

JAVA, un langage typé

Les types sont très importants en JAVA : le compilateur vérifie toujours les types des différentes variables

- Certaines conversions sont **implicites** :

```
1 double d = 1; double d2 = i; // avec i un int existant
```

Il est possible de transformer n'importe quel type de base en String (l'affichage est donc facile)

```
1 String s = "mon_message_"+1.5+"_"+d;
```

- Certaines conversions doivent être **explicites**

```
1 int i = (int) 2.4;
```

- Conversions **impossibles**

```
1 int i = (int) true; // conversion impossible des booléens
2                      // vers le domaine numérique
3                      // => ERREUR de compilation
```

SYNTAXE DES FONCTIONS/MÉTHODES

- Syntaxe directement inspirée du C/C++

- Déclaration :

visibilité	type de retour	nom de la fonction	arguments
public	double	monCalcul	(int arg1, String arg2)

```
1 public double monCalcul(int i, double d){
```

SYNTAXE DES FONCTIONS/MÉTHODES

- Syntaxe directement inspirée du C/C++

- Déclaration :

visibilité	type de retour	nom de la fonction	arguments
public	double	monCalcul	(int arg1, String arg2)

- Calculs divers

```
1 public double monCalcul(int i, double d){  
2     double resultat = 100. + i * d;
```

SYNTAXE DES FONCTIONS/MÉTHODES

- Syntaxe directement inspirée du C/C++

- Déclaration :

visibilité type de retour nom de la fonction arguments
public double monCalcul (int arg1, String arg2)

- Calculs divers
- Retour (obligatoire si autre que void)

```
1 public double monCalcul(int i, double d){  
2     double resultat = 100. + i * d;  
3     return resultat;  
4 }
```

OPÉRATEURS CLASSIQUES (PAR ORDRE DE PRIORITÉ)

opérateurs postfixés	[] . expr++ expr--
opérateurs unaires	++expr --expr +expr -expr ~ !
création ou cast	new (type) expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
décalages	<< >> >>>
opérateurs relationnels	< > <= >=
opérateurs d'égalité	== !=
et bit à bit	&
ou exclusif bit à bit	^
ou (inclusif) bit à bit	
et logique	&&
ou logique	
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ^= = <<= >>= >>>=

CONDITIONNELLES

- Syntaxe du *Si, ... Alors* :

```
1 int i=8;
2 if(i > 5){
3     // code à effectuer dans ce cas
4 }
5 else{ // le else est facultatif
6     // Code à effectuer sinon
7 }
```

CONDITIONNELLES

- Syntaxe du *Si, ... Alors* :

```
1 int i=8;
2 if(i > 5){
3     // code à effectuer dans ce cas
4 }
5 else{ // le else est facultatif
6     // Code à effectuer sinon
7 }
```

- En cas de clauses multiples :

```
1 switch(i){
2 case 1:
3     // Code à effectuer si i == 1
4     break; // sinon le reste du code est AUSSI effectué
5 case 2: //
6     // Code à effectuer si i == 2
7     break;
8 default : // Si on n'est passé nulle part ailleurs
9 }
```


BOUCLES

La définition des boucles est identiques au C/C++

- Syntaxes : 2 options (principales)

Pour i allant de 0 à 9, faire...

```
1 int i;  
2 for(i=0; i<10; i++){// i prend les valeurs 0 à 9 =  
3                      // 10 itérations  
4 // code a effectuer 10 fois  
5 }
```

BOUCLES

La définition des boucles est identiques au C/C++

- Syntaxes : 2 options (principales)

Pour i allant de 0 à 9, faire...

```
1 int i;  
2 for(i=0; i<10; i++){// i prend les valeurs 0 à 9 =  
3                     // 10 itérations  
4 // code a effectuer 10 fois  
5 }
```

Tant que i inférieur à 10, faire...

```
1 int i = 0;  
2 while(i<10){// i prend les valeurs 0 à 9 =  
3           // 10 itérations  
4 // code a effectuer 10 fois  
5 i++; // ne pas oublier, sinon boucle infinie !  
6 }
```

- D'autres syntaxes sont possibles : *do...while* etc...

3 types d'interruptions de boucles

- **return** : l'interruption la **plus forte**. Coupe l'exécution de la méthode (**sort de la fonction**, pas seulement la boucle).

```
1 // le modulo par 5 peut-il retrouver un entier >=5?
2 public void maFonction(){
3     for(int i=0; i<10; i++){
4         if(i%5>4){
5             System.out.println("C'est très étrange");
6             return;
7         }
8     }
9     System.out.println("L'opération modulo 5 retourne "+
10         "toujours un entier inférieur à 5");
11 }
```

3 types d'interruptions de boucles

- `return`
- `break` : l'interruption de boucle

```
1 // 6 fait-il parti des multiples de 2?
2 public void maFonction(){
3     boolean found = true;
4     for(int i=0; i<10; i++){
5         if(i * 2 == 6){
6             found = true;
7             break; // pas besoin d'aller plus loin
8         }
9     }
10    if(found)
11        System.out.println("6 fait parti des multiples de 2")
12 }
```

3 types d'interruptions de boucles

- `return`
- `break`
- `continue` : sauter une itération de boucle

```
1 // afficher 3./i pour i variant de -10 à 10
2 // il faut penser à sauter le cas 0 qui provoque un problème
3 public void maFonction(){
4     for(int i=-10; i<=10; i++){// -10 et 10 inclus
5         if(i == 0)
6             continue;
7         System.out.println("3./"+i+"="+(3./i) );
8     }
9 }
```

Ces instructions rendent le code plus lisible en limitant notamment le nombre de blocs imbriqués.

2i002 - Exercices d'application

Vincent Guigue



Programmation **sans objet**, boucles, types de base.
Donner les instructions de compilation exécution

- Calculer la somme des entiers de 1 à 63.

$1+2+3+\dots+63$

- Calculer la somme des entiers pairs jusqu'à 60.

$2+4+6+\dots+60$

- En changeant les pas de boucle
- Avec un `for` / avec un `while`
- En utilisant le modulo

QUELQUES FONCTIONS MATHÉMATIQUES

- Générer un nombre aléatoire entre 0 et 1
- Ecrire une fonction qui écrit bonjour une fois sur 3 et au revoir 2 fois sur 3 aléatoirement
- Calculer le cosinus de π et vérifier le résultat

BOOLÉEN

En JAVA, le booléen est un type à part entière

Que pensez-vous des opérations suivantes :

```
1 int i = 3;  
2 if(2<i<10) ...  
3 if(i) ...  
4 if(i < 5 && i == 2) ...  
5 if(i < 5 || i == 2) ...
```

Extrait de documentation

- Dans la classe String : `char charAt(int)`, `int length()`, `String replace(char, char)`
- Déclarer une chaîne de caractères
- Si elle termine par un "e", afficher : `mot féminin : «mot»`
sinon, afficher `mot masculin : «mot»`
- Vérifier si le mot contient une lettre particulière
- Afficher : `pi = 3.14...` (tous les types se *transforment en chaînes de caractères*)
- Afficher π avec un nombre de décimales fixé (affichage formaté)

Construction, méthode, réflexion sur les instances

- Classe Vecteur (en 2D)
 - Constructeur,
 - Constructeur random,
 - Affichage et méthode standard `toString`
 - Norme,
 - Produit scalaire,
- Classe de test
- Instruction de compilation / exécution