Applications Réparties

Université Paris VIII / Parcours SRM / Master 2 - 2008
Pierre Châtel - Laboratoire d'Informatique de Paris VI, Thales - chatelp@gmail.com

Organisation

Module de 21h réparties entre cours et TPs selon les besoins.

7 séances du 10 Octobre au 21 Novembre, 3h par semaine.

Evaluation:

- ... des notions de cours :
  Un examen final sous forme de questions de rédactions et QCM.

- ... des connaissances technologiques :
  Un projet final par groupe de 3 évalué dans une soutenance en Janvier prochain.

Objectifs

Comprendre les concepts (Cours)

- Les technologies évoluent très vite mais les concepts sous-jacent restent le plus souvent les mêmes.
- Un bon ingénieur est capable de s'adapter très vite aux nouvelles technologies en transposant ses connaissances.

Comprendre les technologies actuelles (Cours + TP)

- Plusieurs solutions industrielles seront indiquées, les principales testées en TP.

Acquérir des connaissances utiles dans le milieu du travail

- Les solutions réparties sont devenues incontournables pour répondre à certains besoins industriels (contraintes de taille, de répartition, d'accès, de tolérance aux pannes,...)

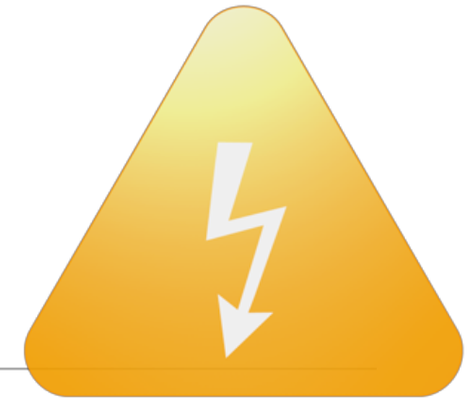
Informations

- Page du cours à l'adresse suivante :

  http://www.chatelp.org/?page_id=49

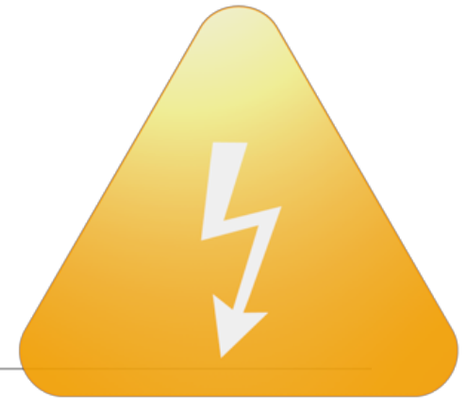
- Les corrections de TP fournies seront disponibles sur cette page.

Chapitres du cours, par concepts



1. Le langage Java
    1. Concepts objets, de base, Eclipse
    2. POO, concepts avancés
2. Rappels
    1. Les applications réparties
    2. Architectures réparties
    3. Client-Serveur et Java
3. Objets répartis : CORBA
4. Annuaires pour applications réparties
    1. JNDI
    2. LDAP
5. JMS
6. Services Web

Chapitres du cours, par concepts



1. Le langage Java

Séance 1, TP1

1. Concepts objets, de base, Eclipse

2. POO, concepts avancés

Séance 2, TP2

2. Rappels

1. Les applications réparties
2. Architectures réparties
3. Client-Serveur et Java

Séance 3, Projet

3. Objets répartis : CORBA

Séance 4, TP3

4. Annuaires pour applications réparties

1. JNDI
2. LDAP

Séance 5, TP4

5. JMS

Séance 6, TP5

6. Services Web

Séance 7, TP6

Le langage Java

Chap #2

Introduction

Omniprésence de Java dans le monde du réparti grand public (ex. téléphones mobiles) et industriel (ex. serveurs d'entreprise).

Historiquement : Applications Réparties = CORBA ("Common Object Request Broker Architecture") développées en C++.

Problème : complexité des AR CORBA C++.

Une solution :

- Java intègre des technologies réparties directement dans le langage et les kits de développement (ex. **RMI**, SOAP, XML-RPC, JMS, JXTA, **EJB** et même ... CORBA !).

- Grâce à ces technologies : gestion de la sécurité, de l'activation automatique des composants, des transactions, ...

Introduction

Avantages du langage Java pour les ARs :

- Facilités de communication entre machines (téléchargement de code, gestions des flots d'exécution concurrents, ...).

- Gestion automatique et protection de la mémoire : “Garbage Collector”, mécanisme d’instanciation des objets (instruction *new*, mais pas de *malloc*).

- Simplicité de la syntaxe : Java à été conçu du départ comme un langage objet, contrairement à C (puis C++ à partir de C).

- Des environnements de développements (IDE) évolués : Eclipse, NetBeans, ...

Concepts objets

Les langages *Java*, *C++*, *Objective-C*, *C#*, ... suivent tous un **paradigme** de programmation appelé **Programmation Orientée Objets** (“Object Oriented Programming” ou OOP).

Dans ce paradigme, un objet logiciel est un ensemble logique ...

- **d’états.**
- **de comportements.**

Les objets logiciels sont souvent utilisés pour modéliser les objets du “monde réel”.

Nous allons essayer de comprendre comment l’état et le comportement d’un objet sont représentés en Java, introduire le concept d’encapsulation des données et expliquer les bénéfices de cette programmation orientée objets.

Concepts Objets

Objets

Des objets du monde réel : un chien, une table, un vélo, une télé, ...

- Ils possèdent **tous** un état et un comportement.

- Ex. du vélo : état = {pédalier courant, cadence des pédales, vitesse actuelle}, comportement = {changer de pédalier, changer de cadence, utiliser les freins}.

Les objets logiciels sont conceptuellement identiques à ceux du monde réel. Ils possèdent eux aussi des états et comportements.

- Etats = *champs* (*"fields"*) de l'objet Java. Certains langages utilisent aussi le terme de *variable*.

- Comportements = *méthodes* (*"methods"*) de l'objet Java. Certains langages utilisent aussi le terme de *fonction*.

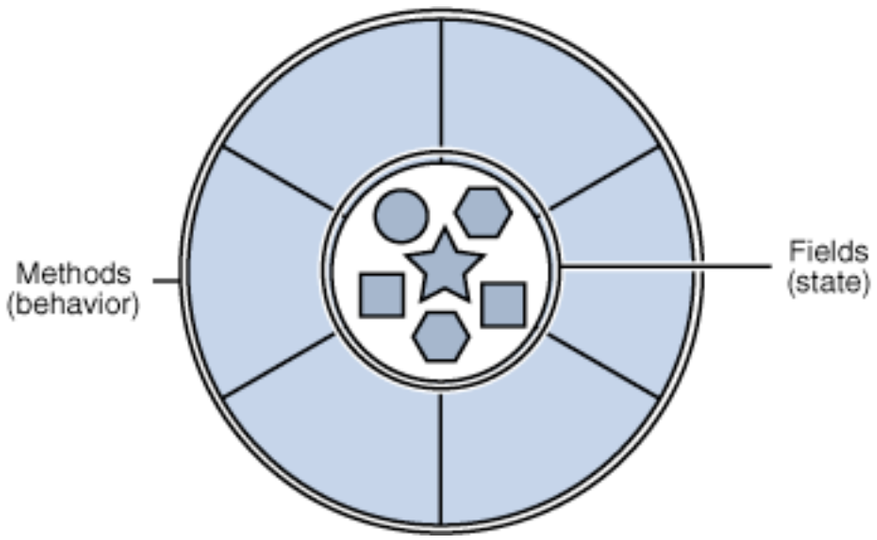
Concepts Objets

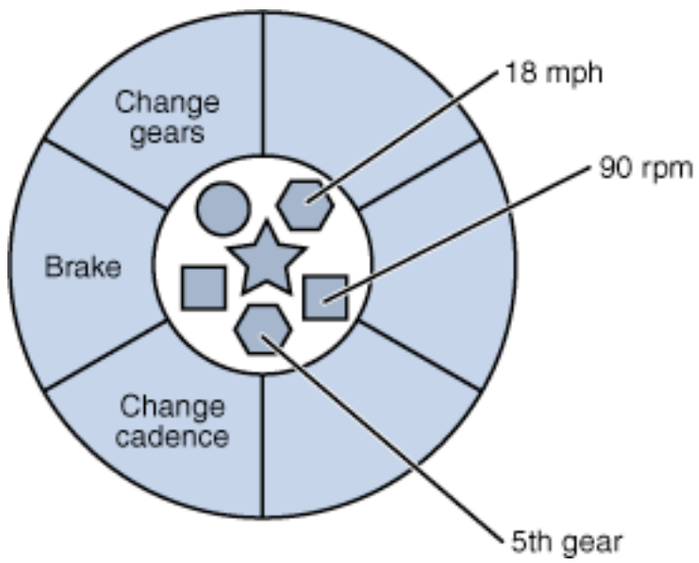
Objets : encapsulation des données

Les méthodes **agissent** sur (modifient) l'état des objets.

Les méthodes **permettent d'accéder** à l'état (interne) des objets.

Encapsulation : quand le développeur cache l'état de ses objets et oblige l'utilisateur à interagir avec l'objet uniquement via ses méthodes publiques.





➡Concept fondamental de l'OOP qui permet à l'objet de garder le contrôle sur son état.

➡Ex. vélo permet de changer 6 vitesses uniquement, contrôlé par une méthode qui vérifie que *0 <= gear < 6*.

Concepts Objets

Objets : bénéfices

Concevoir votre code sous la forme d'objets logiciels individuels apporte un certain nombre de bénéfices par rapport à une conception linéaire traditionnelle :

- **Modularité** : le code source d'un objet (une classe) peut être développé et maintenu indépendamment du code source des autres types d'objets. Une fois créé un objet peut être véhiculé aisément au sein d'une application.
- **Masquage de l'information** : grâce à l'encapsulation, pas besoin de connaître les détails d'implémentation d'un objet, juste les méthodes exposées.
- **Réutilisabilité** : si vous disposez d'un objet pré-existant (mis au point par un autre développeur), vous pouvez le réutiliser dans votre propre programme.
- **Pluggabilité et facilité pour debugger** : si un objet particulier semble poser problème, il est possible de le remplacer dynamiquement en phase de debug.

Concepts Objets
Classes

Une classe peut être vue comme un plan, ou **une fabrique d'objets**.

Dans le monde réel : de nombreux objets du même type (Ex. des milliers de vélos du même modèle, construits à partir du même plan).

En Java : on dit que ces vélos sont des *instances* de la classe `Bicyle`.

```
class Bicycle {

        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeCadence(int newValue) { cadence = newValue; }
        void changeGear(int newValue) { gear = newValue; }
        void speedUp(int increment) { speed = speed + increment; }
        void applyBrakes(int decrement) { peed = speed - decrement; }

        void printStates() {
             System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
        }
}
```

Concepts Objets

Classes

Une classe peut être vue comme un plan, ou **une fabrique d'objets**.

Dans le monde réel : de nombreux objets du même type (Ex. des milliers de vélos du même modèle, construits à partir du même plan).

En Java : on dit que ces vélos sont des *instances* de la classe `Bicyle`.

```
class Bicycle {

        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeCadence(int newValue) { cadence = newValue; }
        void changeGear(int newValue) { gear = newValue; }
        void speedUp(int increment) { speed = speed + increment; }
        void applyBrakes(int decrement) { peed = speed - decrement; }

        void printStates() {
             System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
        }
}
```

Concepts Objets

Classes

Une classe peut être vue comme un plan, ou **une fabrique d'objets**.

Dans le monde réel : de nombreux objets du même type (Ex. des milliers de vélos du même modèle, construits à partir du même plan).

En Java : on dit que ces vélos sont des *instances* de la classe `Bicyle`.

```
class Bicycle {

        int cadence = 0;
        int speed = 0;
        int gear = 1;

        void changeCadence(int newValue) { cadence = newValue; }
        void changeGear(int newValue) { gear = newValue; }
        void speedUp(int increment) { speed = speed + increment; }
        void applyBrakes(int decrement) { peed = speed - decrement; }

        void printStates() {
             System.out.println("cadence:"+cadence+" speed:"+speed+" gear:"+gear);
        }
}
```

Un champs (état)

Une méthode (comportement)

Concepts Objets

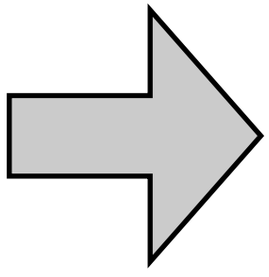
Classes

Dans l'exemple précédent **la classe** ne contiens pas de méthode "main", il ne s'agit pas d'une application complète, seulement d'**une fabrique d'instance** qui peut **être utilisée** par une autre portion de code.

La responsabilité de créer et d'utiliser des instances de `Bicycle` appartient ici à une autre classe de l'application :

```
class BicycleDemo {
     public static void main(String[] args) {
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.printStates();
     }
}
```

Affiche



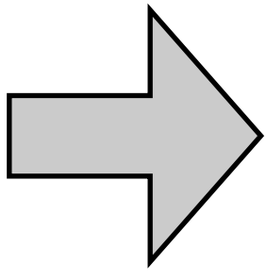
Concepts Objets
Classes

Dans l'exemple précédent **la classe** ne contiens pas de méthode "main", il ne s'agit pas d'une application complète, seulement d'**une fabrique d'instance** qui peut **être utilisée** par une autre portion de code.

La responsabilité de créer et d'utiliser des instances de `Bicycle` appartient ici à une autre classe de l'application :

```
class BicycleDemo {
     public static void main(String[] args) {
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();

          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.printStates();
     }
}
```

Affiche



```
cadence:50 speed:10 gear:2
cadence:40 speed:10 gear:2
```

Concepts Objets
Héritage

Certains types d'objets on des caractéristiques communes.

- Ex. Les VTT, vélos de ville et tandems partagent tous les caractéristiques propres aux vélos (pédalier courant, cadence des pédales, vitesse actuelle) mais avec des subtilités liées à leurs usages :

  - VTT : plus de vitesses disponibles et plus solide.
  - Vélos de ville : plus légers que les autres catégories.
  - Tandems : 2 sièges et 2 guidons.

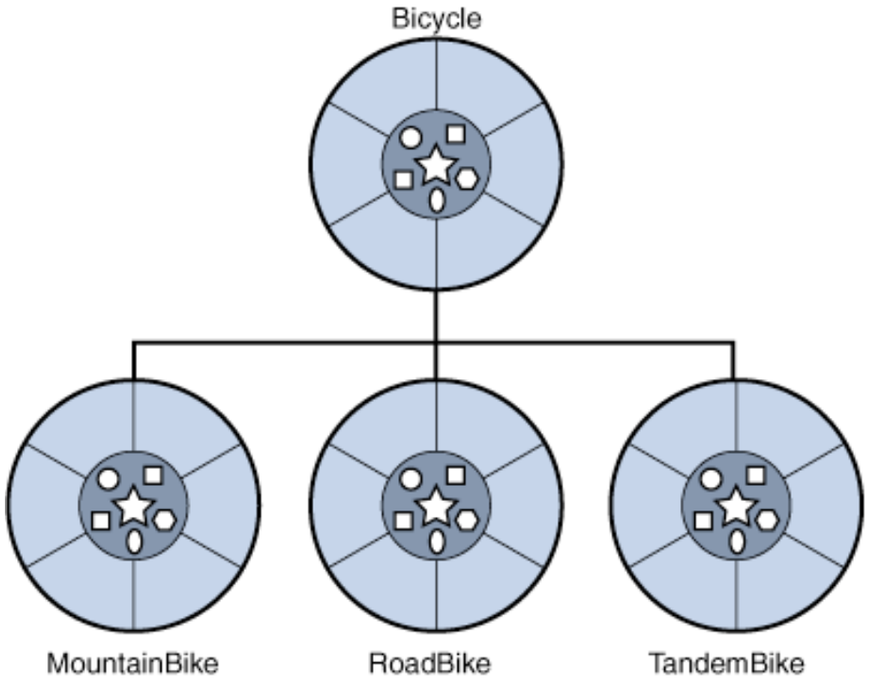
L'OOP permet à certaines classes (nommées *classes filles*) d'hériter des états et comportements usuels d'autres classes (nommées *classes mères*).

En Java, une classe fille ne peut être liée qu'à **une seule et unique** classe mère directe, une classe mère à une nombre illimité de classes filles.

Concepts Objets

Héritage

Dans notre exemple, les VTT, vélos de ville et tandems deviennent des classes filles (ou sous-classes, “subclasses”) de la classe mère (“superclass”) vélo.



La syntaxe pour créer une classe fille repose sur l’utilisation du mot clé `extends`. Elle hérite alors de tous les champs et méthodes de la classe mère.

```
class MountainBike extends Bicycle {

     // new fields and methods defining a mountain bike would go here

}
```

Concepts Objets

Interfaces

Une interface est un regroupement déclaratif **de méthodes (pas de champs)** définissant le comportement visible, **depuis l'extérieur**, d'un objet.

Une interface est assimilable à un **contrat** formel passé entre une classe et le monde extérieur, il est vérifié à la compilation.

Dans sa version la plus simpliste, une interface ne contient qu'un ensemble de **déclarations** de méthodes.

Ex. Interface d'un Vélo et son implémentation basique :

```
interface Bicycle {

       void changeCadence(int newValue);

       void changeGear(int newValue);

       void speedUp(int increment);

       void applyBrakes(int decrement);
}
```

```
class BasicBicycle implements Bicycle {

   // Content of previous Bicycle *class*
      + public keyword on methods

}
```

Concepts Objets

Interfaces

Une interface est un regroupement déclaratif **de méthodes (pas de champs)** définissant le comportement visible, **depuis l'extérieur**, d'un objet.

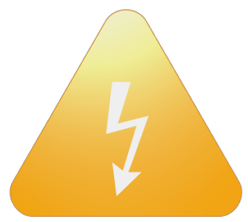
Une interface est assimilable à un **contrat** formel passé entre une classe et le monde extérieur, il est vérifié à la compilation.

Dans sa version la plus simpliste, une interface ne contient qu'un ensemble de **déclarations** de méthodes.

Ex. Interface d'un Vélo et son implémentation basique :

```
interface Bicycle {

        void changeCadence(int newValue);

        void changeGear(int newValue);

        void speedUp(int increment);

        void applyBrakes(int decrement);
}
```

```
class BasicBicycle implements Bicycle {

   // Content of previous Bicycle *class*
      + public keyword on methods

}
```



Classe et interface ne peuvent pas porter le même nom.

Concepts Objets

Paquetages

Un paquetage, ou “package” est un espace de nommage qui permet l’organisation logique d’un ensemble de classes et interfaces.

Conceptuellement, les paquetages sont proches des dossiers et sous-dossiers présents sur votre système de fichier.

La plateforme Java fournis une bibliothèque de classes très importante : il s’agit de la “Java Application Programming Interface” ou “Java API”.

- Ses classes représentent les briques de base et les tâches usuelles de vos programmes.

- Elles sont organisées sous forme de paquetages.
  Ex. Classe chaîne de caractères String : `java.lang.String`

- Documentation de l’API : http://java.sun.com/javase/6/docs/api/index.html

Concepts de base

Les transparents suivants présentes un ensemble de concepts de bases essentiels à la compréhension du langage Java et son utilisation.

Contrairement au chapitre précédent, ces concepts ne sont pas spécifiquement liés à l'aspect "Objet" de la programmation en Java.

Seront notamment abordés :

- Les variables.
- Les opérateurs.
- Les expressions, instructions, blocs.
- La gestion du flot de contrôle.
- Les chaînes de caractères.

Concepts de base

Variables

Comme vous l'avez vu précédemment, un objet Java préserve son état dans des champs ("fields") :

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

En Java, les termes *champs* et *variables* sont tous deux utilisés, ce qui est une source de confusion puisqu'ils semblent se référer à la même chose. **En fait, on distingue plusieurs types de variables :**

- Les variables d'instance (champs non statiques).
- Les variables de classes (champs statiques).
- Les variables locales.
- Les paramètres.

Concepts de base
Variables

Les variables d'instance (champs non statiques) :

- Pour être précis, un objet préserve son état individuel dans des champs *non-statiques* (ils n'ont pas été déclarés avec le mot clé `static`).
- **La valeur de ces champs est unique à chaque instance de la classe considérée** (autrement dit à chacun de ses objets).
- Ex. La vitesse courant d'un vélo est indépendante de celle d'un autre vélo.

Les variables de classes (champs statiques) :

- Ces champs sont déclarés avec le mot clé `static` qui indique au compilateur que seule une copie de cette variable doit exister.
- Une variable de classe est partagée par toutes les instances de la classes.
- Ex. Le nombre total de vitesse d'un type particulié de vélo.

```
static int numGears = 6;
```

Concepts de base
Variables

Les variables locales :

- Permet notamment de préserver l'état d'une méthode, d'effectuer des calculs...
- Syntaxe de déclaration similaire aux variables d'instances (pas de mot clé `static`), la seule différence vient de l'emplacement (dans une méthode).
- Une variable d'instance est accessible uniquement au sein de la méthode qui la déclare, pas dans le reste de la classe.

Les paramètres :

- Déjà vu dans les exemples de méthodes précédents :

```
void changeCadence(int newValue) (...)
public static void main(String[] args) (...)
```

- Attention : les paramètres sont toujours désignés comme étant des variables, jamais comme des champs.

Concepts de base

Variables : nommage

Comme dans tout langage de programmation, il faut respecter un ensemble de règles et conventions lors du nommage des variables :

- Le nom des variables est sensible à la “case” (majuscule/minuscule) et est constitué d’une chaîne de lettres et chiffres Unicodes de taille illimitée.

- Par convention il est conseillé de toujours commencer le nom d’une variable par une lettre et d’éviter l’usage de ‘$’ et ‘_’.

- Evidemment, les espaces blancs sont interdits.

- Il est conseillé, et de bon sens, d’utiliser des noms complets sans abbréviations et de commencer le nom de la variable par une minuscule :

  ✓ Nommage correct : `cadenceMaximum`, `vitesseVelo`, `palier`, ...

  X Nommage incorrect : `c`, `VitesseVelo`, `$p`, ...

Concepts de base

Variables : types primitifs

Le langage Java est fortement typé : toutes les variables doivent avoir un type déclaré avant de pouvoir être utilisées.

- Ex. `int gear = 1;`

Le type de la variable détermine les valeurs qu'elle peut contenir ainsi que les opérations qui peuvent lui être appliquées.

Le langage Java supporte au total 8 types primitifs, ils sont prédéfinis et nommés par des mots-clé réservés :

- `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` et `char`.

Le support des chaînes de caractères passe par la classe `java.lang.String`.

- Entourer un texte de guillemets (“ “ ou ‘Quotes’) permet la création automatique d'une instance immutable (non-modifiable) de cette classe.
- Il est possible de concaténer des chaînes avec l'opérateur `+`.
- Ex. `String s = "this is ” + “a string";`

Concepts de base
Variables : types primitifs

`byte` : sur 8 bits, signé → valeur possible ∈ [-128, 127].

- Utile pour sauver de la mémoire dans de très grands tableaux, si optimiser l'usage mémoire est important.
- Rarement mis en oeuvre dans une utilisation courante.

`short` : sur 16 bits, signé → valeur possible ∈ [-32 768, 32 767].

- Mêmes remarques que pour `byte`.

`int` : sur 32 bits, signé → valeur possible ∈ [-2 147 483 648, 2 147 483 647].

- **Type à utiliser par défaut pour les valeurs entières**, sauf cas particulier d'optimisation (`byte` ou `short` dans ce cas).

`long` : sur 64 bits, signé → valeur possible ∈ [-9 223 372 036 854 775 808, 9 223 372 036 854 775 807].

- A utiliser à la place de `int` pour stocker une très grande (ou très petite...) valeur entière.

Concepts de base

Variables : types primitifs

`float` : sur 32 bits, valeurs flottantes négatives et positives à précision simple (consulter spec. Java chapitre 4.2.3 *)

- Utile pour sauver de la mémoire dans de très grands tableaux de valeurs flottantes, si optimiser l'usage mémoire est important.

`double` : sur 64 bits, valeurs flottantes négatives et positives à précisision double (consulter spec. Java chapitre 4.2.3 *)

- **Type à utiliser par défaut pour les valeurs flottantes**, sauf cas particulier d'optimisation (`float` dans ce cas).

`boolean` : 1 bit d'information, mais taille pas explicitement définie en Java !

- uniquement 2 valeurs possibles : `true` ou `false`.

`char` : un charactère Unicode sur 16 bits → valeur possible ∈ ['\u0000' (ou 0), '\uffff' (ou 65 535)]

* http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3

Concepts de base

Variables : types primitifs

Type primitif	Valeur par défaut
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000'
String (or any object)	null
boolean	FALSE

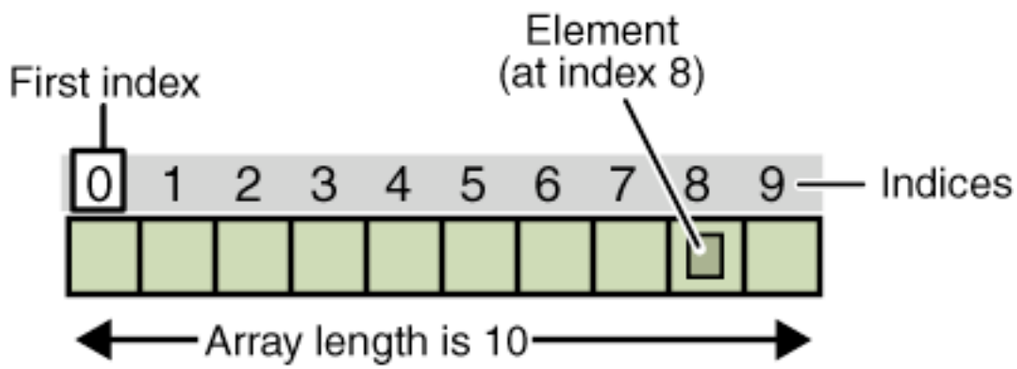
Concepts de base

Tables

Une table est un conteneur d'objets qui possède un nombre fixe de valeurs d'un type unique.

La longueur d'une table est établie lors de sa création. Après création, cette taille est fixe.

Chaque entrée de la table est appelée un *élément*. Et chaque élément est accessible par un index commençant à 0.

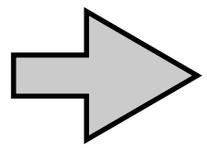


Concepts de base

Tables : ex. de code


```
class ArrayDemo {
     public static void main(String[] args) {
          int[] anArray;              // declares an array of integers

          anArray = new int[10];      // allocates memory for 10 integers

          anArray[0] = 100;           // initialize first element
          anArray[1] = 200;           // initialize second element
          anArray[2] = 300;           // etc.
          anArray[3] = 400;
          anArray[4] = 500;
          anArray[5] = 600;
          anArray[6] = 700;
          anArray[7] = 800;
          anArray[8] = 900;
          anArray[9] = 1000;

          System.out.println("Element at index 0: " + anArray[0]);
          System.out.println("Element at index 1: " + anArray[1]);
          System.out.println("Element at index 2: " + anArray[2]);
          System.out.println("Element at index 3: " + anArray[3]);
          System.out.println("Element at index 4: " + anArray[4]);
          System.out.println("Element at index 5: " + anArray[5]);
          System.out.println("Element at index 6: " + anArray[6]);
          System.out.println("Element at index 7: " + anArray[7]);
          System.out.println("Element at index 8: " + anArray[8]);
          System.out.println("Element at index 9: " + anArray[9]);
     }
}
```

Affiche

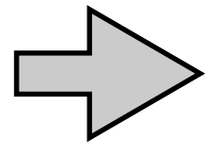


Concepts de base

Tables : ex. de code


```
class ArrayDemo {
     public static void main(String[] args) {
          int[] anArray;              // declares an array of integers

          anArray = new int[10];      // allocates memory for 10 integers

          anArray[0] = 100;           // initialize first element
          anArray[1] = 200;           // initialize second element
          anArray[2] = 300;           // etc.
          anArray[3] = 400;
          anArray[4] = 500;
          anArray[5] = 600;
          anArray[6] = 700;
          anArray[7] = 800;
          anArray[8] = 900;
          anArray[9] = 1000;

          System.out.println("Element at index 0: " + anArray[0]);
          System.out.println("Element at index 1: " + anArray[1]);
          System.out.println("Element at index 2: " + anArray[2]);
          System.out.println("Element at index 3: " + anArray[3]);
          System.out.println("Element at index 4: " + anArray[4]);
          System.out.println("Element at index 5: " + anArray[5]);
          System.out.println("Element at index 6: " + anArray[6]);
          System.out.println("Element at index 7: " + anArray[7]);
          System.out.println("Element at index 8: " + anArray[8]);
          System.out.println("Element at index 9: " + anArray[9]);
     }
}
```

Affiche



```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

Concepts de base

Tables : opérations

Déclaration : comme pour une déclaration de variable, elle est en 2 parties : son type et son nom. On n'indique pas la taille lors de la déclaration.

```
byte[] bytes;
short[] shorts;
long[] longs;
```

```
float[] floats;
double[] doubles;
boolean[] booleans;
```

```
char[] chars;
String[] strings;
```

Création, initialisation :

- en utilisant l'opération `new` :

```
anArray = new int[10];
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

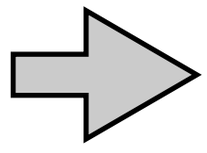
- en utilisant ce raccourcis :

```
int[] anArray = {100, 200, 300, 400, 500, 600,
700, 800, 900, 1000};
```

Copie : utilisation de la méthode `arrayCopy` de la classe `System`.

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

Affiche



Concepts de base

Tables : opérations

Déclaration : comme pour une déclaration de variable, elle est en 2 parties : son type et son nom. On n'indique pas la taille lors de la déclaration.

```
byte[] bytes;
short[] shorts;
long[] longs;
```

```
float[] floats;
double[] doubles;
boolean[] booleans;
```

```
char[] chars;
String[] strings;
```

Création, initialisation :

- en utilisant l'opération `new` :

```
anArray = new int[10];
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

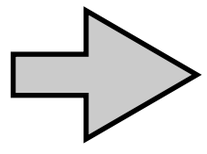
- en utilisant ce raccourcis :

```
int[] anArray = {100, 200, 300, 400, 500, 600,
700, 800, 900, 1000};
```

Copie : utilisation de la méthode `arrayCopy` de la classe `System`.

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

Affiche



```
caffein
```

Concepts de base

Tables : tables multidimensionnelles

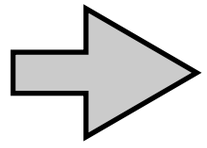
Il est possible de déclarer des “tables de tables”, connues aussi sous le nom de tables multidimensionnelles :

- En utilisant 2 ou plusieurs ensembles de [ ]. Ex. `String[][] names.`
- Chaque élément peut être accédé par un ensemble de valeurs d’index correspondantes.

En fait, en Java, une table multidimensionnelle est une table dans les composants sont eux même des tables. Une conséquence directe et que les lignes peuvent avoir des tailles variables au sein d’une même table !

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]);
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

Affiche



Concepts de base

Tables : tables multidimensionnelles

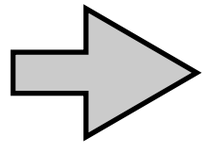
Il est possible de déclarer des “tables de tables”, connues aussi sous le nom de tables multidimensionnelles :

- En utilisant 2 ou plusieurs ensembles de [ ]. Ex. `String[][] names.`
- Chaque élément peut être accédé par un ensemble de valeurs d’index correspondantes.

En fait, en Java, une table multidimensionnelle est une table dans les composants sont eux même des tables. Une conséquence directe et que les lignes peuvent avoir des tailles variables au sein d’une même table !

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]);
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

Affiche



```
Mr. Smith
Ms. Jones
```

Concepts de base
Opérateurs

Les opérateurs en Java sont des symboles spéciaux qui effectuent des opérations spécifiques sur un, deux ou trois *opérandes* puis retournent un résultat.

Il est possible de classer les opérateurs par leur priorité. Dans la table suivante, plus les opérateurs les plus haut, plus leur priorité est élevée.

- Un opérateur *a* de plus haute priorité qu'un opérateur *b* sera évalué avant ce dernier.
- Si plusieurs opérateurs de même priorité se suivent dans une expression alors :
  - les opérateurs binaires sont évalués de gauche à droite.
  - les opérateurs d'affectation sont évalués de droite à gauche.

Concepts de base

Opérateurs

Nom	Opérateurs
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Concepts de base
Opérateurs : affectation

Il s'agit certainement de l'opérateur que l'on rencontre le plus fréquemment : "=".

Nous l'avons précédemment rencontrés dans la classe Vélo, il affecte la valeur à sa droite à l'opérande sur sa gauche (une variable).

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Cet opérateur peut aussi être utilisé pour affecter des références d'objets à des variables. Nous reviendrons sur ce point dans la suite des transparents.

Concepts de base

Opérateurs : op. arithmétiques

Le langage Java fournis des opérateurs qui effectuent l'addition, la soustraction, la multiplication et la division.

Il sont similaires à leurs alter-ego mathématiques que vous connaissez déjà. A l'exception peut-être de "%" qui divise son premier opérande par son second et retourne le *reste* comme résultat :

- `+` opérateur additif (aussi utilisé pour la concaténation de Strings).
- `-` opérateur de soustraction.
- `*` opérateur de multiplication.
- `/` opérateur de division.
- `%` opérateur de reste.

Il est possible de combiner les opérateurs arithmétiques avec l'affectation :

```
x = x + 1;  ⇒  x += 1;
```

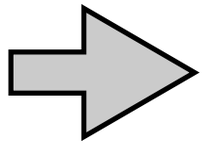
Concepts de base

Opérateurs : op. arithmétiques


```
class ArithmeticDemo {
     public static void main (String[] args){
          int result = 1 + 2;                    // result is now 3
          System.out.println(result);

          result = result - 1;                   // result is now 2
          System.out.println(result);

          result = result * 2;                   // result is now 4
          System.out.println(result);

          result = result / 2;                   // result is now 2
          System.out.println(result);

          result = result + 8;                   // result is now 10
          result = result % 7;                   // result is now 3
          System.out.println(result);
     }
}
```

```
class ConcatDemo {
     public static void main(String[] args){
          String firstString = "This is";
          String secondString = " a concatenated string.";
          String thirdString = firstString+secondString;
          System.out.println(thirdString);
     }
}
```

Affiche



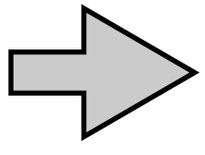
Concepts de base

Opérateurs : op. arithmétiques

```
class ArithmeticDemo {
     public static void main (String[] args){
          int result = 1 + 2;                    // result is now 3
          System.out.println(result);

          result = result - 1;                   // result is now 2
          System.out.println(result);

          result = result * 2;                   // result is now 4
          System.out.println(result);

          result = result / 2;                   // result is now 2
          System.out.println(result);

          result = result + 8;                   // result is now 10
          result = result % 7;                   // result is now 3
          System.out.println(result);
     }
}
```

```
class ConcatDemo {
     public static void main(String[] args){
          String firstString = "This is";
          String secondString = " a concatenated string.";
          String thirdString = firstString+secondString;
          System.out.println(thirdString);
     }
}
```

Affiche



```
"This is a concatenated string."
```

Concepts de base
Opérateurs : op. unaires

Les opérateurs unaires ne requièrent qu'un seul opérande :

- `+` Opérateur positif unaire; indique une valeur positive (optionnel).
- `-` Opérateur négatif unaire; indique une valeur négative.
- `++` Opérateur d'incrementation; augmente une valeur de 1.
- `--` Opérateur de décrementation, diminue une valeur de 1.
- `!` Opérateur de complément logique, inverse une valeur booléenne.

`++` et `--` peuvent être appliqués avant (préfix) ou après (postfix) l'opérande.

- Dans le cas `result++;` → `result` incrémenté de 1, la valeur renvoyée par l'expression est la valeur initiale de `result`.
- Dans le cas `++result;` → `result` incrémenté de 1, la valeur renvoyée par l'expression est la valeur incrémentée de `result`.

Concepts de base

Opérateurs : op. unaires


```
class UnaryDemo {
     public static void main(String[] args){
          int result = +1;                // result is now 1
          System.out.println(result);
          result--;                       // result is now 0
          System.out.println(result);
          result++;                       // result is now 1
          System.out.println(result);
          result = -result;               // result is now -1
          System.out.println(result);
          boolean success = false;
          System.out.println(success);    // false
          System.out.println(!success);   // true
     }
}
```

```
class PrePostDemo {
     public static void main(String[] args){
       int i = 3;
       i++;
       System.out.println(i);     // "4"
       ++i;
       System.out.println(i);     // "5"
       System.out.println(++i);   // "6"
       System.out.println(i++);   // "6" Attention!
       System.out.println(i);     // "7"
     }
}
```

Concepts de base

Opérateurs : égalité, inégalités

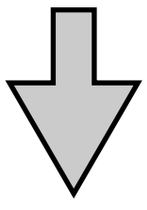
De même que pour les opérateurs arithmétiques, ceux d'égalité et d'inégalité vont probablement vous sembler familiers. Gardez surtout en mémoire que vous devez utiliser "==" et non "=" pour tester une égalité entre deux valeurs primitives :

- == égal à.
- != non égal à.
- > plus grand que.
- >= plus grand ou égal à.
- < inférieur à.
- <= inférieur ou égal à.

Concepts de base

Opérateurs : égalité, inégalités

```
class ComparisonDemo {

     public static void main(String[] args){
          int value1 = 1;
          int value2 = 2;
          if(value1 == value2) System.out.println("value1 == value2");
          if(value1 != value2) System.out.println("value1 != value2");
          if(value1 > value2) System.out.println("value1 > value2");
          if(value1 < value2) System.out.println("value1 < value2");
          if(value1 <= value2) System.out.println("value1 <= value2");
     }
}
```

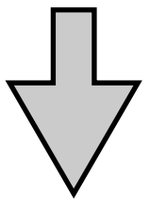


Affiche

Concepts de base

Opérateurs : égalité, inégalités

```
class ComparisonDemo {

     public static void main(String[] args){
          int value1 = 1;
          int value2 = 2;
          if(value1 == value2) System.out.println("value1 == value2");
          if(value1 != value2) System.out.println("value1 != value2");
          if(value1 > value2) System.out.println("value1 > value2");
          if(value1 < value2) System.out.println("value1 < value2");
          if(value1 <= value2) System.out.println("value1 <= value2");
     }
}
```



Affiche

```
value1 != value2
value1 < value2
value1 <= value2
```

Concepts de base

Opérateurs : op. conditionnels

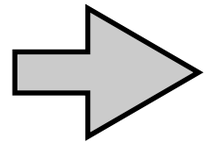
Les opérateurs conditionnels suivants s'appliquent uniquement à des opérandes booléens. Le second opérande n'est évalué que si nécessaire :

- && et-conditionnel booléen.
- || ou-conditionnel booléen.

L'opérateur `?:` peut être vu comme un raccourcis pour l'instruction `if-then-else`. On l'appel opérateur ternaire car c'est le seul à utiliser 3 opérandes.

```
class ConditionalDemo {
     public static void main(String[] args){
          int value1 = 1;
          int value2 = 2;
          int result;
          boolean someCondition = true;
          result = someCondition ? value1 : value2;

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

Affiche



Concepts de base

Opérateurs : op. conditionnels

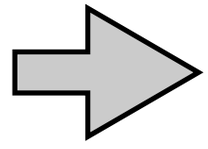
Les opérateurs conditionnels suivants s'appliquent uniquement à des opérandes booléens. Le second opérande n'est évalué que si nécessaire :

- && et-conditionnel booléen.
- || ou-conditionnel booléen.

L'opérateur `?:` peut être vu comme un raccourcis pour l'instruction `if-then-else`. On l'appel opérateur ternaire car c'est le seul à utiliser 3 opérandes.

```
class ConditionalDemo {
     public static void main(String[] args){
          int value1 = 1;
          int value2 = 2;
          int result;
          boolean someCondition = true;
          result = someCondition ? value1 : value2;

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

Affiche



1

Concepts de base

Opérateurs : comparateur de types

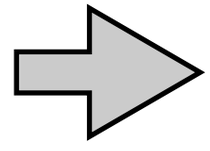
L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche



Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
```

Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
```

Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
```

Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
```

Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
```

Concepts de base

Opérateurs : comparateur de types

L'opérateur `instanceof` compare un objet à un type spécifié.

Vous pouvez l'utiliser pour tester si un objet est une instance :

- d'une classe,
- d'une sous-classe,
- ou d'une classe qui implémente une interface donnée.

```
class InstanceofDemo {
  public static void main(String[] args) {

    Parent obj1 = new Parent();
    Parent obj2 = new Child();

    System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
    System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
    System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
    System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
    System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
    System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
  }
}

class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Affiche

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

Concepts de base

Expressions, instructions, blocs

Maintenant que nous avons étudié les variables et opérateurs, il est temps de s'intéresser aux expressions, instructions et blocs :

- Les opérateurs peuvent être utilisés pour construire des **expressions**, qui calculent des valeurs.

- Les expressions constituent le coeur des **instructions**.

- Les instructions peuvent être regroupées au sein de **blocs**.

Concepts de base

Expressions

Une expression est construite par des variables, des opérateurs et des invocations de méthodes :

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if(value1 == value2) System.out.println("value1 == value2");
```

Le type d'une valeur retournée par une expression depend des éléments utilisés dans l'expression. (ex. cadence = 0 retourne un int).

En Java, il est possible de composer les expressions. (ex. 1 * 2 * 3).

- L'ordre d'évaluation d'une expression composée peut influencer son résultat. Par défaut il suit l'ordre de priorité des opérateurs vu précédemment.

- Il est possible de parenthèser une expression pour forcer localement l'ordre d'évaluation.

  Ex. `x + y / 100` équivalent à `x + (y / 100)`
  Mais on peut forcer l'addition : `(x + y) / 100`

Concepts de base
Instructions

Les instructions peuvent être assimilées à des phrases du langage courant.

Une instruction forme une unité d'exécution complète et autonome.

Les expressions suivantes peuvent être transformées en instructions en leur ajoutant seulement un “`;`” de terminaison :

- Expressions d’affectation.
- Tout usage de `++` ou `--`.
- Invocations de méthodes.
- Expressions de création d’objets (d’instanciation).

S’ajoute à ces instructions simples deux autres types courants :

- Les instructions de déclaration (ex. `double aValue = 8933.234;`)
- Les instructions de gestion du flot de contrôle. (cf. chapitre dédié).

Concepts de base

Instructions

Quelques exemple d'instructions simples :

```
aValue = 8933.234;                          // assignment statement

aValue++;                                   // increment statement

System.out.println("Hello World!");         // method invocation statement

Bicycle myBike = new Bicycle();             // object creation statement
```

Concepts de base

Blocs

Un bloc est un regroupement d'une ou plusieurs instructions entre accolades "{ }".

Un bloc peut être utilisés partout où une instruction est autorisée.

```
class BlockDemo {
      public static void main(String[] args) {
            boolean condition = true;
            if (condition) { // begin block 1
                  System.out.println("Condition is true.");
            } // end block one
            else { // begin block 2
                  System.out.println("Condition is false.");
            } // end block 2
      }
}
```

Concepts de base
Gestion du flot de contrôle

Les instructions dans votre code source sont généralement exécutées **de haut en bas, dans l'ordre dans lequel elles apparaissent**.

A contrario, les instructions de gestion du flot de contrôle cassent cet ordre d'exécution en effectuant de la prise de décision, du bouclage, du branchement...

- ... de générale elles vous permettent d'**exécuter conditionnellement certaines portions de votre code**.

Ce chapitre décrit les instructions de prise de décision (`if-then`, `if-then-else`, `switch`), les instructions de boucle (`for`, `while`, `do-while`) et de branchement (`break`, `continue`, `return`) supportées en Java.

Concepts de base

Gestion du flot de contrôle : `if-then`

C’est l’instruction de flot la plus basique. Elle indique à votre programme de n'exécuter une portion de code seulement si la condition indiquée dans le `if` s’évalue à `true`.

Ex. Un vélo n’autorise les freins à s’enclencher que s’il est déjà en mouvement.

```
void applyBrakes(){
     if (isMoving){  // the "if" clause: bicycle must moving
          currentSpeed--; // the "then" clause: decrease current speed
     }
}
```

Si la condition est évaluée à `false`, alors le contrôle saute directement à la fin de l’instruction `if-then`.

Nb : il est possible d'omettre les `{}` si la clause `then` ne contiens qu’une seule instruction.

Concepts de base

Gestion du flot de contrôle : `if-then-else`

L'instruction `if-then-else`, contrairement à la précédente, fournis un chemin alternatif que le flot de contrôle va emprunter si la condition indiquée dans le `if` est évaluée à `false`.

Ex. Utilisé ici pour afficher un message d'erreur si les freins sont utilisés alors que le vélo n'est pas en mouvement.

```
void applyBrakes(){
     if (isMoving) {
          currentSpeed--;
     } else {
          System.err.println("The bicycle has already stopped!");
     }
}
```

Il est possible d'enchaîner les `if-then-else` avec la construction `else-if` :

```
if (testscore >= 90) {
           grade = 'A';
} else if (testscore >= 80) {
           grade = 'B';
} else if (testscore >= 70) {
           grade = 'C';
} (...)
```

Concepts de base

Gestion du flot de contrôle : `switch`

L'instruction **`switch`**, contrairement aux deux précédentes, permet de spécifier de manière simplifiée un nombre potentiellement infini de chemins pour le flot de contrôle.

La sélecteur (~ condition du if) utilisé dans le **`switch`** fonctionne avec les types primitifs suivants : **`byte`**, **`short`**, **`char`** et **`int`**.

Le **`switch`** évalue son sélecteur et dirige le flot de contrôle vers la ligne **`case`** correspondante, si elle existe. (attention à ne pas oublier le **`break`** à la fin !).

```
class SwitchDemo {
    public static void main(String[] args) {
        int month = 8;
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
               (...)
            default: System.out.println("Invalid month.");break;
        }
    }
}
```

Concepts de base

Gestion du flot de contrôle : `while`

L'instruction `while` exécute continuellement un bloc d'instructions **tant qu'une condition particulière s'évalue à `true`**. Sa syntaxe est la suivante :

```
while (expression) {
     statement(s)
}
```

`expression` doit retourner un booléen :

- `true` → le bloc `statement(s)` est exécuté, l'expression est re-testée pour déterminer si on doit boucler.
- `false` → le bloc `statement(s)` n'est pas/plus exécuté, le contrôle saute directement à la fin de l'instruction `while`.

```
//compteur de 1 à 10
class WhileDemo {
     public static void main(String[] args){
          int count = 1;
          while (count < 11) {
               System.out.println("Count is: " + count);
               count++;
          }
     }
}
```

```
//boucle infinie
while (true){
    // your code goes here
}
```

Concepts de base

Gestion du flot de contrôle : `do-while`

La langage Java fournis également une instruction `do-while` qui peut être définie de la façon suivante :

```
do {
      statement(s)
} while (expression);
```

La différence entre `do-while` et `while` tient dans le fait que `do-while` évalue son expression de test à la fin de la boucle au lieu du début.

- Ce qui signifie que le contenu de `statement(s)` est évalué au moins une première fois (même si expression est évalue à `false`).

```
//compteur de 1 à 10
class DoWhileDemo {
     public static void main(String[] args){
          int count = 1;
          do {
               System.out.println("Count is: " + count);
               count++;
          } while (count <= 11);
     }
}
```

Concepts de base

Gestion du flot de contrôle : `for`

L'instruction `for` fournis un moyen compact d'itérer sur un espace de valeurs. Elle peut être définie de la façon suivante :

```
for (initialization; termination; increment) {
    statement(s)
}
```

Attention aux points suivants :

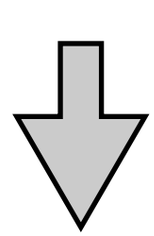
- L'expression `initialization` initialise la boucle `for`, elle est exécutée une seule fois, lorsque la boucle est lancée.
- Quand l'expression `termination` est évaluée à `false`, la boucle se termine.
- L'expression `increment` est évaluée après chaque itération de la boucle, elle peut tout aussi bien décrémenter.

Les expressions de la boucle for sont optionnelles (`for ( ; ; ) { (...) }`).

Concepts de base

Gestion du flot de contrôle : `for`

```
class ForDemo {
     public static void main(String[] args){
          for(int i=1; i<11; i++){
               System.out.println("Count is: " + i);
          }
     }
}
```

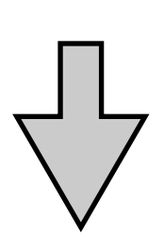


Affiche

Concepts de base

Gestion du flot de contrôle : `for`

```
class ForDemo {
     public static void main(String[] args){
          for(int i=1; i<11; i++){
               System.out.println("Count is: " + i);
          }
     }
}
```



Affiche

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Concepts de base

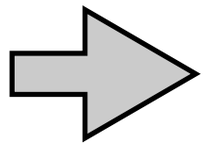
Gestion du flot de contrôle : `for(each)`

Depuis Java v1.5, on dispose d'une nouvelle forme de `for` pour itérer de manière compacte et efficace sur une `Collection` ou table.

- Il s'agit en fait d'un **for-each** bien que cette syntaxe n'existe pas en Java.

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};

class EnhancedForDemo {
     public static void main(String[] args){
          int[] numbers = {1,2,3,4,5,6,7,8,9,10};
          for (int item : numbers) {
            System.out.println("Count is: " + item);
          }
     }
}
```

Affiche



Il est recommandé d'utiliser cette nouvelle forme dès que possible dans votre code car elle en augmente considérablement la lisibilité et évite de déclarer inutilement des variables temporaires (`i`, `j`, `k`, ...).

Concepts de base

Gestion du flot de contrôle : `for(each)`

Depuis Java v1.5, on dispose d'une nouvelle forme de `for` pour itérer de manière compacte et efficace sur une `Collection` ou table.

- Il s'agit en fait d'un **for-each** bien que cette syntaxe n'existe pas en Java.

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};

class EnhancedForDemo {
     public static void main(String[] args){
          int[] numbers = {1,2,3,4,5,6,7,8,9,10};
          for (int item : numbers) {
            System.out.println("Count is: " + item);
          }
     }
}
```

Affiche

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Il est recommandé d'utiliser cette nouvelle forme dès que possible dans votre code car elle en augmente considérablement la lisibilité et évite de déclarer inutilement des variables temporaires (`i`, `j`, `k`, ...).

Concepts de base

Gestion du flot de contrôle : `break`

Cette instruction possède 2 formes distinctes : avec un *label*, ou sans *label*.

- La forme sans label à été utilisée précédemment dans le `switch` pour interrompre le flot de contrôle. Il est en fait possible de faire la même chose avec `for`, `while` et `do-while`.

```
int searchfor = 12;
 for (i = 0; i < arrayOfInts.length; i++) {
   if (arrayOfInts[i] == searchfor) {
       foundIt = true;
       break; // interrompt le for puisque l'élément voulu à été trouvé
   }
}
```

- La forme avec label permet non plus d'arrêter seulement la boucle englobante mais plusieurs boucles imbriquées.

```
search:
      for (i = 0; i < arrayOfInts.length; i++) {
          for (j = 0; j < arrayOfInts[i].length; j++) {
              if (arrayOfInts[i][j] == searchfor) {
                  foundIt = true;
                  break search;
              }
          }
      }
//-> attention : le flot de contrôle est amené à la 1ère ligne après l'instruction labelisée (pas un go-to) !
```

Concepts de base

Gestion du flot de contrôle : `continue`

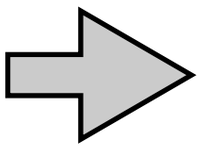
L'instruction **`continue`** permet de sauter l'itération courante d'un **`for`**, **`while`** ou **`do-while`**.

Comme pour le **`break`** il est possible d'utiliser des labels pour sauter l'itération d'une boucle imbriquée.

L'exemple suivant parcours une chaîne de caractère à la recherche des lettres '`p`' pour les compter :

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe = "peter piper picked a peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            //interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            //process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Affiche



Concepts de base

Gestion du flot de contrôle : `continue`

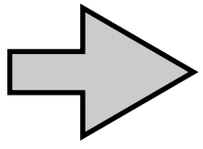
L'instruction **`continue`** permet de sauter l'itération courante d'un **`for`**, **`while`** ou **`do-while`**.

Comme pour le **`break`** il est possible d'utiliser des labels pour sauter l'itération d'une boucle imbriquée.

L'exemple suivant parcours une chaîne de caractère à la recherche des lettres '`p`' pour les compter :

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe = "peter piper picked a peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            //interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            //process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Affiche



```
Found 9 p's in the
string.
```

Concepts de base

Gestion du flot de contrôle : `return`

L'instruction return permet de sortir de la méthode en cours d'exécution. Le flot de contrôle retourne alors au point où la méthode avait été invoquée.

Elle se présente sous 2 formes :

- Retour avec valeur qui doit être d'un type compatible avec le type de retour déclaré par la méthode : ex. `return ++count;`

- Retour sans valeur : `return;`

Concepts de base
Méthode ‘main’

Le point d’entrée pour l'exécution d’un programme Java est une classe qui définit une méthode statique ‘main’ dont la signature est donnée dans l’exemple ci dessous.

Un même programme peut disposer de plusieurs méthodes main qui sont autant de points d’entrées possible pour l'exécution.

Cette méthode prend en argument un tableau qui contient la liste des arguments qui ont été passés au programme (en ligne de commande par exemple).

```
class BicycleDemo {
     public static void main(String[] args) {
          Bicycle bike1 = new Bicycle();
          Bicycle bike2 = new Bicycle();
          (...)
     }
}
```

L'IDE Eclipse

La majeure partie des TPs de ce cours seront réalisés sous Eclipse.

Eclipse est un environnement de développement ou IDE ("Integrated Development Environment".

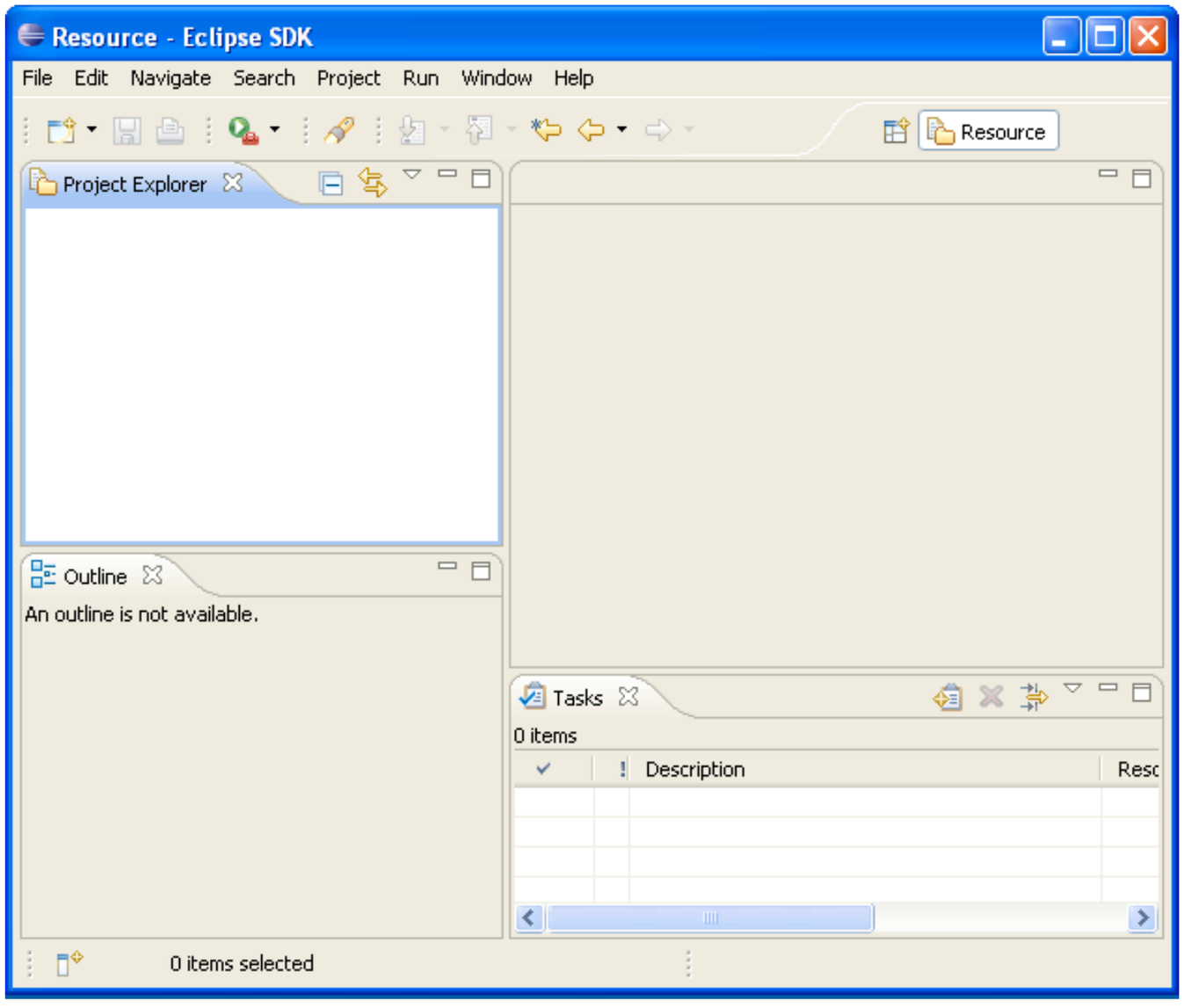
Il permet la programmation en Java ou dans d'autres langages, il intègre notamment :

- Des outils d'aide à la programmation (completion, refactoring).
- Des outils d'aide à la détection de bugs (informations directement dans le code).
- Le cycle de développement en Java (développement, compilation, exécution) et masque l'étape de compilation du code source au programmeur.

L'IDE Eclipse

Le Workbench

Le **Workbench** est la fenêtre principale d'Eclipse, elle est liée à un **workspace** qui contient un ou plusieurs projets de développement.

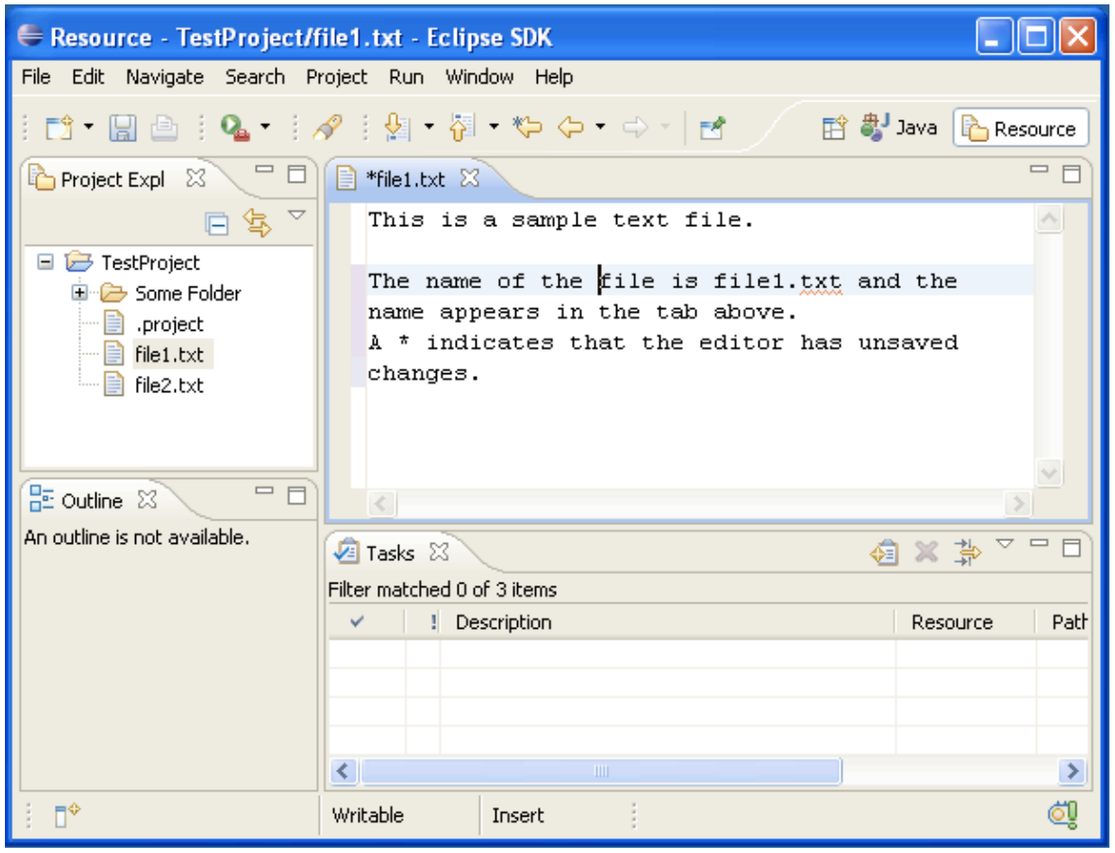


L'IDE Eclipse

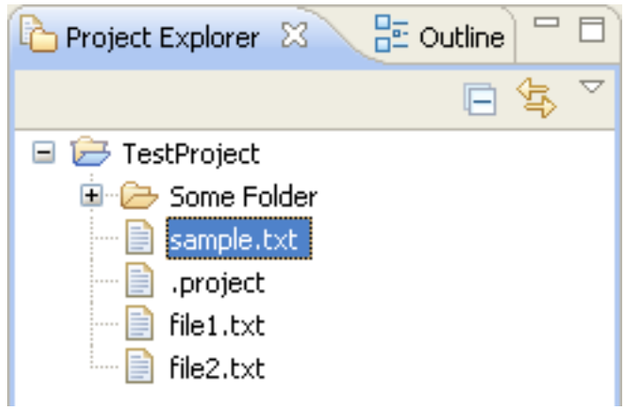
Perspectives, éditeurs et vues

Un workbench affiche une **perspective** : plusieurs perspectives pré-configurées sont fournies avec Eclipse.

Une perspective est un ensemble de fenêtres qui découpent le workbench. Chacune de ces fenêtre sont des **éditeurs** ou des **vues**.



Un éditeur (de texte)



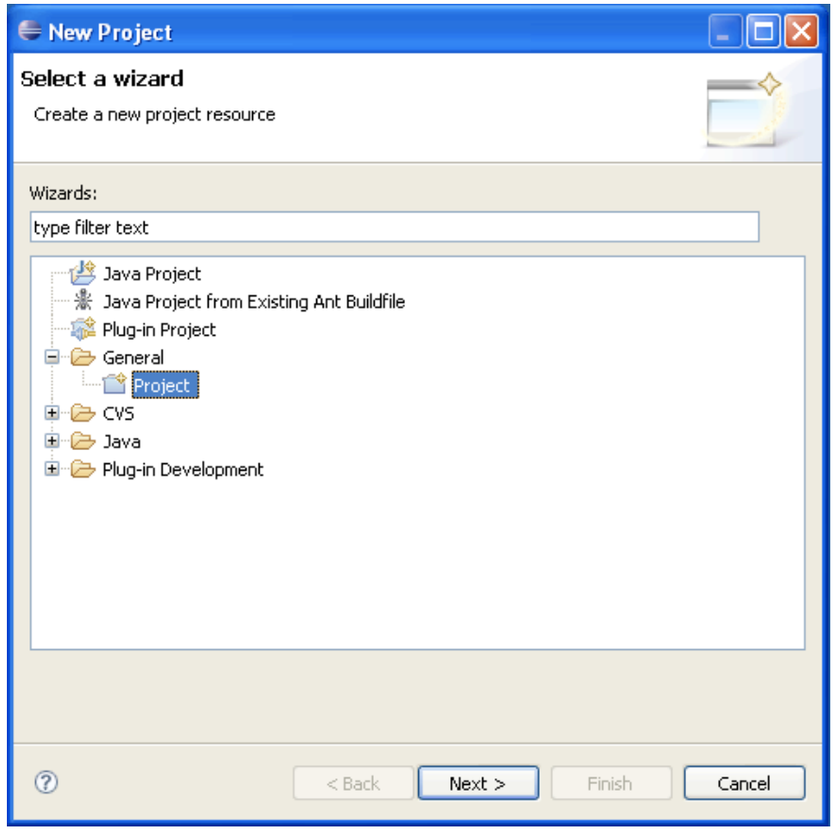
Une vue
L'explorateur de projets

L’IDE Eclipse

Projets

Il est possible de créer un nouveau projet de plusieurs manières : via le menu File, via le menu contextuel de l’explorateur de projets ou via un bouton de la barre d’outils.

Il existe différents types de projet : dans les TPs nous utiliserons uniquement les projets Java.



“Wizard” de création de projets obtenu via le menu File.

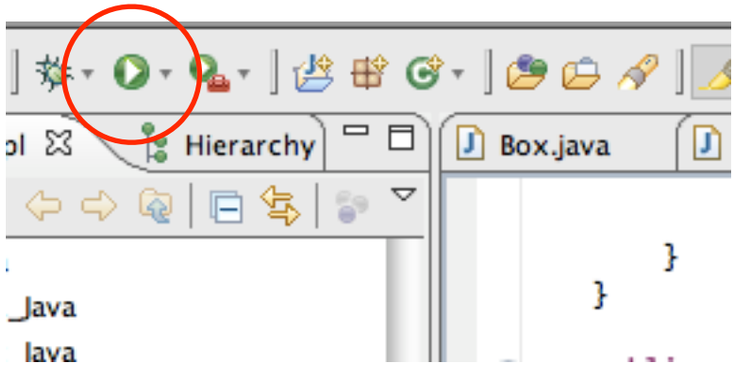
L'IDE Eclipse

Remarques

Utilisez l'aide contextuelle et les suggestions d'Eclipse pour corriger vos erreurs de programmation en cours de développement : n'attendez pas la compilation !

Pour créer une nouvelle *configuration* d'exécution, cliquez-droit sur une classe Java contenant une méthode main valide et choisissez Run As → Java Application.

Rappel : dans Eclipse, vous ne gérez pas la compilation de vos sources Java : utilisez uniquement le bouton *Run* pour lancer vos exécutables.



Fin du cours #1