Programmation Orientée Objet avec Java

5 septembre 2013

Chapitre 1: Introduction
Chapitre 2: Bases de Java
Chapitre 3: Les tableaux simples
Chapitre 4: Classes, attributs, méthodes
Chapitre 5: Héritage
Chapitre 6: Documenter un projet
Chapitre 7: Exceptions
Chapitre 8: Interfaces – Classes abstraites
Chapitre 9: Les collections
Chapitre 10: Mini projet

mini-projet

Chapitre I – Introduction

- I. Les notions de la programmation orientée objet
- II. Un premier exemple

Chapitre I – Introduction

- I. Les notions de la programmation orientée objet
- II. Un premier exemple

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

un exemple de programme

Sous linux, on écrit avec un éditeur (gedit ou emacs) le fichier `MaClasse.java`.
(**attention** : Java distingue majuscules et minuscules)

```
public class MaClasse {
public static void main (String[] args) {
System.out.println("affichez ce que vous voulez");}
}// ceci est un commentaire
```

programme

- `class` mot réservé qui indique que l'on commence la description d'une classe, ce mot est nécessairement suivi du nom de la classe : c'est l'entête de la classe (ligne 1)
- { marque le début d'un bloc ; en ligne 2 c'est le début du corps de la classe MaClasse
- une classe contient des attributs et des fonctions ou des méthodes ; ici il n'y a qu'une méthode nommée `main` (ligne 2)
- `public static` sont des *modificateurs*
- `void` indique que la méthode `main` ne renvoie aucune valeur

programme

- l'exécution d'un programme en Java se fait toujours dans une méthode principale qui se nomme toujours `main` ; l'entête de cette méthode est toujours comme dans la ligne 2, on ne peut changer que l'identificateur arg
- `String` est une classe du paquetage `java.lang` ;
- `String [ ]` arg est le paramètre de la méthode `main`, et c'est un tableau de chaînes de caratères
- `System.out.println`("affichez ce que vous voulez") ; est une instruction qui écrit à l'écran *affichez ce que vous voulez* puis passe à la ligne ; `println` est une méthode de l'objet `out` qui appartient à la classe `System`, cet objet sert à écrire dans le fenêtre d'exécution
- toute instruction se termine par un point-virgule

compilation

Pour compiler ce fichier, dans une fenêtre de commande, on exécute la commande

javac MaClasse.java

On peut alors voir dans le répertoire courant un fichier *MaClasse.class*, ce fichier contient le bytecode de la classe *MaClasse*. Ce code est indépendant de la machine (et de son système d'exploitation).
Puis on fait appel à la machine virtuelle Java qui interprète le bytecode au fur et à mesure de l'exécution du programme : on utilise la commande

java MaClasse

qui exécute la méthode `main` contenue dans la classe *MaClasse*.

On prendra l'habitude suivante :
le nom du fichier sera MaClasse.java pour la déclaration `class MaClasse` { ....
D'autre part, on a tout intérêt à séparer les fichiers source des fichiers bytecode .class.
Pour cela on créera deux répertoires jumeaux Source et Class ; on écrira les fichiers source .java dans Source.
Toujours dans le répertoire Source, on compilera par la commande

javac -d ../Class/ MaClasse.java

puis toujours dans le répertoire Source, on exécutera par la commande

java -cp ../Class/ MaClasse

Librairies

Java fournit de nombreuses librairies de classes remplissant des fonctionnalités très diverses : c'est l'API Java (**A**pplication and **P**rogramming **I**nterface /Interface pour la programmation d'applications).
Ces classes sont regroupées par catégories en paquetages (ou «packages»).
Les principaux paquetages :

- java.util : structures de données classiques
- java.io : entrées / sorties
- java.lang : chaînes de caractères, interaction avec l'OS, threads
- java.awt : interfaces graphiques, images et dessins
- java.applet : les applets sur le web
- javax.swing : package récent proposant des composants « légers » pour la création graphiques

JDK

L'environnement de développement fourni par Sun est le JDK (**J**ava **D**evelopment **K**it / Kit de développement Java).
Il contient :

- les classes de base de l'API java (plusieurs centaines),
- la documentation au format HTML (dans le répertoire où est installé le JDK - /jdk1.../docs/api/index.html - ou bien à l'adresse suivante http ://java.sun.com/docs/index.html)
- le compilateur : javac
- la JVM (machine virtuelle) : java
- le visualiseur d'applets : appletviewer
- le générateur de documentation : javadoc

Deux ouvrages :

- en anglais : The Java Programming Language, K. Arnold, J. Gosling et H. David, Addison Wesley, 2000
- en français : Le Langage Java, Irène Charron, Hermès Sciences

Environnement intégré

Il existe de nombreux IDE (Integrated Development Environment) parmi lesquels

- Eclipse
- Netbeans
- JCreator (Windows uniquement)
- ...

Bien sûr il faut apprendre à s'en servir car ils offrent de nombreuses possibilités.
Pour l'instant on fera simple, comme dans l'exemple précédent (sous Linux).

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Structure d'un programme Java

Un programme Java est constitué d'un ou plusieurs fichiers dont le nom est terminé par .java.
La structure générale d'un programma Java est

- Bibliothèques utilisées
- classes
  - attributs (ou variables)
  - méthodes (ou fonctions ou procédures)

Le point d'entrée est toujours la méthode `main` dont l'en-tête est toujours
`public static void main ( String [ ] arg ) {... }`
C'est la méthode automatiquement appelée par Java.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Tout ce qui est délimité par de accolades respectivement ouvrante et fermante sera appelé un *bloc*.
Un programme doit toujours être commenté pour des raisons de lisibilité, d'évolution potentielle . . .
Les commentaires sont

- /∗ sur plusieurs lignes ∗/
- // sur une seule ligne
- / ∗ ∗ sont utilisés pour créer une documentation automatique en html ∗/

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Les identificateurs sont composés de suite de lettre ou de chiffre, commencent par une lettre et doivent être différents des mots réservés et des mots-clés.
Par exemple, MaClasse, maClasse, ma_classe, maClasse1,...
On respectera cette convention d'écriture :

- tout nom de classe commence avec une majuscule
- tout nom de méthode commence avec une minuscule
- tout nom de variable ou attribut commence avec une minuscule ; par exemple entier
- si le nom de la méthode ou de la variable est composé de plusieurs mots alors le premier caractère des mots suivants le premier mot sont en majuscule ; par exemple changerDePlace() , monEntierAMoiEtAPersonneDAutre.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Les types en Java

Toutes les données manipulées en Java doivent être **typées**.
En Java il y a deux catégories de type, les types **primitifs** et les autres.
Les types primitifs comprennent les types : booléen, caractère, entier ou réel.
Les autres types sont les types tableau ou les types objet.
Par exemple, `String` est un type objet, `int[ ]` est un type de tableaux d'entiers.
Pour créer de nouveaux types on peut construire des tableaux ou définir des *classes*.

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Par exemple, on définit une classe `Personne` dans le but de définir une classe `Carnet` qui contiendra des `Personne` dans un tableau.

```
class Personne {
String nom;
integer age;
....}
class Carnet {
Personne[] liste;
...
```

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

En Java la manipulation des données diffère selon leur type.
Règle : LES DONNÉES DE TYPE PRIMITIF SONT MANIPULÉES PAR VALEUR ET LES DONNÉES DES AUTRES TYPES SONT MANIPULÉES PAR RÉFÉRENCE
La référence d'une donnée est l'adresse en mémoire de cette donnée.
La référence est typée par le type de la donnée dont elle est l'adresse mémoire.
Selon la règle énoncée une variable de type primitif contient une valeur de ce type et une variable de type non primitif contient l'adresse mémoire où est stockée la valeur de type.

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Exemple

```
public class Essai {
public static void main(String[] args) {
byte[] t={1,2,3};
byte[] s;
short x=1;
short y;
s=t;
System.out.println("tableau 1 "+ t);
// tableau 1 [B@6e1408
//              adresse en héxadécimal (base 16)
System.out.println("tableau 2 "+ s);
// tableau 2 [B@6e1408
t[0]=-1;
System.out.println("tableau 1, 1ère valeur "+ t[0]);
// tableau 1 [B@6e1408 -1
System.out.println("tableau 2, 1ère valeur "+ s[0]);
// tableau 2 [@6e1408 -1
System.out.println("x="+x);// x=1
y=x; x=2;
System.out.println("x="+x);// x=2
System.out.println("y="+y);// y= 1
}
}
```

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Affectation des types non primitifs

La manipulation des données de type non primitif est limitée à la *comparaison* et l'*affectation*.
Dans ce dernier cas c'est l'adresse mémoire qui est dupliquée et non pas les données référencées.

Remarques :

- la syntaxe de l'affection se fait est le symbole $=$
- la variable à gauche reçoit la valeur de l'expression à droite

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Les types primitifs

Les variables de type primitif

- `boolean`
- `int`
- `float`
- `char`

sont manipulées par **valeur**.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

le type booléen

Le type `boolean` a deux valeurs `true` et `false`.
Il est codée sur 1 bit.
On verra plus tard les opérateurs logiques.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

le type caractère

Le type `char` est codé sur 2 octets selon le code UNICODE 2.1.
Les valeurs de type `char` sont écrites entre apostrophes soit directement soit avec le code unicode sous la forme \uhhhh où hhhh est un chiffre héxadécimal.
Par exemple \u03C6 correspond à φ.
(voir `http://www.unicode.org` pour les caractères en Unicode)

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

les types entier

Selon la dimension du codage, on distingue les types

- `byte` : 1 octet, intervalle $[-128, 127]$
- `short` : 2 octets, intervalle $[-32768, 32767]$
- `int` : 4 octets, intervalle $[-2147483648, 2147483647]$
- `long` : 8 octets, intervalle $[-9.10^{18}, 9.10^{18}]$ environ

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

écriture des entiers

Les valeurs entières peuvent s'écrire

- en base 10 : par exemple 12
- en base 8 (octal) indiquée par un 0 en préfixe : par exemple 014
- en base 16 (héxadécimal) indiquée par 0x en préfixe : par exemple 0xc
- les valeurs de type `long` sont suffixées par l ou L.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérateurs de comparaison et opérateurs logiques

- $==$ teste l'égalité
- $!=$ teste la différence
- $<=, <, >, =>$ testent l'ordre
- && est l'opérateur booléen ET,
- || est l'opérateur booléen OU,
- ! est l'opérateur booléen de négation.

L'évaluation d'une expression booléenne se fait de gauche à droite et est stoppée dès que le résultat est connu.
En effet `false` && ... est faux quelque soit la valeur qui suit `false`.
De même, `true` || ... est vrai quelque soit la valeur qui suit `true`.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Les types réels

Selon la dimension du codage, on distingue les types

- `float` : simple précision sur 4 octets
- `double` : double précision sur 8 octets

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

écriture des flottants

un réel comporte toujours un point (notation anglo-saxonne de la virgule) même s'il a une valeur entière.
Par exemple 1.f vaut 1 ; c'est un flottant et sera affiché 1.0.
Une valeur constante de type `float` est toujours suffixée par f ; par exemple, 10.2f est de type `float`.
Une valeur constante de type `double` n'a pas de rajout ; par exemple, 10.2 est de type `double`.
On peut utiliser une notation scientifique ; par exemple, 1.02e1f de type `float` ou .102e2 de type `double`.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Opérations sur les réels

- $+, -$: signe (arité 1)
- $+, -, *, /$: addition, soustraction, multiplication, division

D'autres opérations sont disponibles dans la classe `java.lang.Math` qu'il faut importer si on veut l'utiliser.
On y trouve par exemple

- les fonctions trigonométriques (argument en radian) : `sin, cos, tan, asin, acos, atan`
- les fonctions logarithmiques et exponentielles : `log, exp, sqrt, pow`
- arrondi : `round, ceil, floor`
- valeur absolue : `abs`

Pour toutes ces fonctions l'argument est de type `double`.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Exemple

```
import java.lang.Math;
public class Essai2 {
public static void main(String[] args) {
System.out.println(Math.sqrt(4.0)) ;//2.0
System.out.println(Math.sin(Math.PI/2)) ;//1.0
}}
```

voir l'API pour la classe Math.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Conversions implicites entre types

- `byte` $\rightsquigarrow$ `short`
- `short` $\rightsquigarrow$ `int`
- `char` $\rightsquigarrow$ `int`
- `int` $\rightsquigarrow$ `float`
- `float` $\rightsquigarrow$ `double`

Par exemple si x est de type `short` et y de type `int`, l'affectation y=x transforme automatiquement la valeur de x en une valeur de type `int` pour l'affecter à y.
Pour le type `char`, une valeur de ce type a un code unicode \uhhhh ; lorsque cette valeur est affectée à une variable i de type int hhhh est calculé en décimal et affecté à i.
Par exemple,

```
i = '2'; System.out.println(i);
```

affiche 50 car le caractère '2' a le code \u0032.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Conversions explicites entre types

On peut explicitement faire les conversions suivantes

- `byte` $\rightsquigarrow$ `char`
- `short` $\rightsquigarrow$ `char`
- `short` $\rightsquigarrow$ `byte`
- `int` $\rightsquigarrow$ `char`
- `int` $\rightsquigarrow$ `short`
- `char` $\rightsquigarrow$ `byte`
- `char` $\rightsquigarrow$ `short`
- `double` $\rightsquigarrow$ `float`
- `float` $\rightsquigarrow$ `int`

La conversion se fait de la façon suivante : (nouveau type) (expression), où expression est du type d'origine.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Conversions explicites entre types

Attention : la conversion s'effectue après évaluation de l'expression dans son type d'origine.
Par exemple
`short s= -140;`
`byte b=(byte) s;//b` vaut 116 car $-140 + 256 = 116$

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs
1. le type booléen
2. le type caractère
3. les types entier
4. Les types réels
5. Conversions
6. Constantes

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Constantes

On peut nommer des constantes de la façon suivante :

```
static final int max = 100;
```

`final` signifie que l'on ne pourra pas changer sa valeur.
L'ordre entre `final` et `static` n'a pas d'importance.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision
1. l'instruction if
2. l'instruction switch

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

La décision

On va voir deux instructions de décision `if` et `switch`

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
1. l'instruction if
2. l'instruction switch
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

syntaxe de if

`if (expression) {SuiteDInstructions1;} else {SuiteDIinstructions2;}`
dans laquelle `expression` est une expression booléenne.
Remarques :

- `else` n'est pas obligatoire,
- Si `SuiteDInstructions` ne contient qu'une unique instruction alors les accolades ne sont pas obligatoires.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision
1. l'instruction if
2. l'instruction switch

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

syntaxe de if

règle : SANS ACCOLADES, `ELSE` SE RAPPORTE TOUJOURS AU `IF` PRÉCÉDENT LE PLUS PROCHE.

Selon cette règle, dans l'instruction suivante

`if (expression1) if (expression2) instruction1; else instruction2;`

`instruction2` s'exécute lorsque `expression1` est vraie et `expression2` fausse. Rien n'est prévue si `expression1` est fausse.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision
1. l'instruction if
2. l'instruction switch

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

syntaxe de if

Si `SuiteDInstructions1` et `SuiteDInstructions2` sont chacune une affectation à **la même variable** x alors on peut écrire

```
x = expression1? expression2 : expression3;
```

avec `expression1` de type booléen, `expression2` et `expression3` de même type que x.

`expression2` est affecté à x si `expression1` est évaluée à `true`
`expression3` est affecté à x si `expression1` est évaluée à `false`

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision
1. l'instruction if
2. l'instruction switch

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

syntaxe de switch

L'instruction `switch` permet de faire plusieurs tests sur la valeur d'une même variable (uniquement de type `byte, short, int, char`).
La syntaxe est la suivante :

```
switch (variable){
case valeur1 : SuiteDinstructions1;break;
...
case valeurk : SuiteDinstructionsk;break;
default : SuiteDinstructions;break;
}
```

Remarque : on peut écrire plusieurs `case` séparés par : pour une seule suite d'instructions
`case valeur1 : case valeur2 : case valeur3 : SuiteDinstructions;break;`

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision
1. l'instruction if
2. l'instruction switch

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

sémantique de switch

Pour chaque `case` la valeur de variable est évaluée puis comparée à la valeur du `case` ; si elles sont égales la suite d'instructions correspondante est exécutée.
Le mot clef `default` indique la suite d'instructions à exécuter lorsqu'aucune des valeurs des `case` ne sont égales à la valeur de la variable.
Attention : si on omet `break` à la fin des instructions, alors toutes lesinstructions suivant la première exécutée seront à leur tour exécutées sans tenir compte des valeurs.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération
1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

L'itération

On va voir les instructions

- `for`
- `while`
- `do while`

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

l'instruction for

La syntaxe est la suivante

```
for (compteur; test; modification){
  SuiteDInstructions; }
```

- `compteur` est la variable qui sert de contrôle à la boucle (avec l'initialisation du compteur et/ou sa déclaration le cas échéant)
- `test` est la condition que doit vérifier le compteur pour **continuer** la boucle `for`,
- `modification` est une instruction qui modifie la valeur du compteur de façon à ce que test devienne faux (sinon boucle infinie),
- si `SuiteDInstructions` ne contient qu'une instruction les accolades ne sont pas indispensables.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

1. l'instruction for

2. l'instruction while

3. l'instruction do while

4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

for

`for`(i=0 ;i<10 ; i++) effectue 10 itérations pour i variant de 0 à 9.
`for`(i=1 ;i<10 ; i++) effectue 9 itérations pour i variant de 1 à 9.
`for`(i=1 ;i<=10 ; i++) effectue 10 itérations pour i variant de 1 à 10.
`for`(`int` j=10 ;j>=1 ; j−−) effectue 10 itérations pour j variant de 10 à 1.
Remarques :

- les boucles `for` peuvent être imbriquées
- le compteur peut être utilisé dans la suite d'instructions
- si compteur est déclaré dans la boucle `for` sa portée est limitée à la boucle (il sera invisible en dehors).

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

l'instruction while

La syntaxe est la suivante

```
while (expressionBooléenne) {
SuiteDInstructions; }
```

`expressionBooléenne` est évaluée et tant qu'elle est vraie alors `SuiteDInstructions` est exécutée.
Il faut bien sûr que cette suite d'instructions fasse évoluer une ou des variables de l'expression booléenne de façon à ce qu'elle devienne fausse sinon on a un risque de boucle infinie.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération
1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

l'instruction do while

La syntaxe est la suivante

```
do { SuiteDInstructions; }
while (expressionBooléenne);
```

`SuiteDInstructions` est exécutée et tant que `expressionBooléenne` est vraie on répète cette exécution. De même on doit s'assurer que l'expression booléenne devienne fausse au cours de l'exécution de la suite d'instructions.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération
1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

différence en while et do while

L'expression booléenne n'est pas évaluée au même moment
Avec `while` l'expression booléenne est évaluée avant l'exécution des instructions.
Avec `do while` l'expression booléenne est évaluée après l'exécution des instructions.
Donc , la suite d'instructions est toujours exécutée au moins une fois avec `do while` .

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération
1. l'instruction for
2. l'instruction while
3. l'instruction do while
4. Dérouter une itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Dérouter une itération

Il est possible

- d'interrompre une boucle par l'instruction `break`
- de sauter une partie des instructions de la boucle par l'instruction `continue`.

`break` l'itération est stoppée et les instructions suivant cette itération sont exécutées

`continue` une boucle `while (expression booléenne){}` ou `do {}while (expression booléenne)` reprend au test de l'expression booléenne

`continue` une boucle `for` passe à l'itération suivante

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Classes enveloppantes

Il est possible de considérer les types primitifs comme des types objet par l'intermédiaire des ≪ classes enveloppantes ≫ qui existent pour chaque type primitif :

- java.lang.Byte $\rightsquigarrow$ byte
- ...
- java.lang.Float $\rightsquigarrow$ float
- java.lang.Boolean $\rightsquigarrow$ boolean
- java.lang.Character $\rightsquigarrow$ character

Ces classes possèdent des méthodes permettant le traitement relatif au type primitif associé.

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

exemple

```
public class TestEnveloppante {
public static void main( String [] args ) {
int n = Integer.parseInt("213");
System.out.println(n+3);// affiche 216
}
```

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties
1. Sorties
2. Entrées

VIII. Construire ses propres fonctions

Sorties

On peut afficher la valeur d'une variable x avec

```
System.out.println(x);
```

qui dans ce cas va à la ligne après affichage.
La commande

```
System.out.print(x);
```

affiche x et laisse le curseur à coté de la valeur de x.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties
1. Sorties
2. Entrées

VIII. Construire ses propres fonctions

Sorties

Pour afficher un texte il faut le mettre entre guillemets.
On peut afficher les contenus de plusieurs variables et du texte en les séparant par un $+$.
On dit que le symbole $+$ est un symbole de *concaténation*.

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

1. Sorties
2. Entrées

VIII. Construire ses propres fonctions

En ligne de commande

Une première possibilité est de profiter de la structure de la méthode `main`.
Elle a comme argument un tableau de chaînes de caractères. On peut alors fournir des valeurs pour ces paramètres en ligne de commande.
Par exemple

```
public class Argument {
public static void main(String[] args) {
int x;
x =  Integer.parseInt(arg[0]);
System.out.println("L'entier vaut " + x);
}
```

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
1. Sorties
2. Entrées
VIII. Construire ses propres fonctions

Entrées

Sinon dans un environnement non graphique on utilisera la classe `Scanner` du package `java.util` (à partir de Java 1.5 seulement) pour réaliser des saisies au clavier.
Grâce à cette classe on peut saisir des valeurs numériques `byte`, `short`, `int`, `long`, `float`, `double` et des valeurs caractères ou chaînes de caractères.
On commence tout d'abord avant la définition de la classe par importer la classe Scanner

```
import java.util.Scanner;
```

Puis on crée un objet de type Scanner de la façon suivante

```
Scanner saisieClavier = new Scanner(System.in);
```

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
1. Sorties
2. Entrées
VIII. Construire ses propres fonctions

la classe Scanner

On utilise une méthode de la classe `Scanner` qui permet de lire un entier, un réel ou une chaîne de caractères.

```
int i;
i=saisieClavier.nextInt();
```

Pour les valeurs numériques les méthodes sont `nextByte`, `nextShort`, `nextInt`, `nextLong`, `nextFloat`, `nextDouble`.
Pour une chaîne de caractères String on utilise la méthode saisieClavier.`next`() qui saisit la suite de caractères jusqu'au caractère espace qui marque la fin de la saisie.
Pour saisir une phrase composée de plusieurs mots on utilise la méthode saisieClavier.`nextLine`().
Pour saisir un caractère on utilise la méthode saisieClavier.`next`().`charAt`(0) qui saisit le premier caractère entré.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
1. Sorties
2. Entrées
VIII. Construire ses propres fonctions

Exemple

```
import java.util.Scanner;
public class EssaiPgcd {
 // fct qui renvoie le pgcd de a et b
static int pgcd(int a,int b){
while (a != b) {
if (a<b) b=b-a ; else a-=b ;}
return a ;}
// méthode main
public static void main ( String [ ] arg ) {
int x ; int y ;
Scanner saisieClavier = new Scanner(System.in) ;
System.out.println("entrez 2 entiers positifs");
System.out.println("sinon entrez (0 pour arreter)");
x=saisieClavier.nextInt() ;
while (x > 0) {
y=saisieClavier.nextInt() ;
if (y <= 0) break; else {
System.out.println("le pgcd de "+x+" et "+y+
                          " est " +pgcd(x,y));
System.out.println("entrer deux entiers positifs
                            (0 pour arreter)");
x=saisieClavier.nextInt() ; }
} } }
```

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Fonctions

On distinguera les sous-programmes s'appliquant à des données de types primitifs et les autres. Pour le premier cas on parlera toujours de fonctions mais de méthodes pour les types non primitifs.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Syntaxe

Pour déclarer une fonction on écrira pour l'instant si la fonction renvoie un résultat

```
public static typeResultat nomDeLaFonction (listeDeParamètres){
  SuiteDInstructions;
  }
```

ou bien si la fonction ne renvoie pas de résultat

```
public static void nomDeLaFonction (listeDeParamètres){
  SuiteDInstructions;
  }
```

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Syntaxe

- `typeResultat` est le type de la valeur qui est retournée par la fonction,
- nomDeLaFonction est le nom de la fonction lequel sera utilisé pour l'appel de la fonction,
- listeDeParamètres est la liste des paramètres avec lesquels la fonction sera appelée,
- (listeDeParamètres) est de la forme (`type1` identPara1, `type2` identPara2, ...),
- ces paramètres sont dits *formels*,
- listeDeParamètres peut être vide,
- si la fonction retourne une valeur SuiteDInstructions doit contenir une instruction `return` valeur ;
- une telle fonction ne renvoie qu'**une seule valeur**,
- l'instruction `return` valeur ; marque la fin de la fin fonction, la méthode appelante reprend le contrôle,

Programmation Orientée Objet avec Java

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

void

`void` indique que la fonction ne retourne aucune valeur

Remarque : une instruction

```
return;
```

peut être utilisée lorsque la fonction ne retourne aucune valeur pour, par exemple, sortir d'une instruction conditionnelle.

Visibilité

Les instructions figurant dans la fonction peuvent utiliser

- les paramètres
- des variables propres à la fonction que l'on dira locales à la fonction.

Règle : TOUTE VARIABLE DÉCLARÉE À L'INTÉRIEUR D'UNE FONCTION N'EST VISIBLE QUE DANS CETTE FONCTION.
Cette règle s'applique aussi aux méthodes et donc à la méthode principale `main`.
Pour une fonction, ses variables locales n'existent que le temps de l'exécution de la fonction et ne peuvent pas être utilisées par d'autres fonctions parce que non visibles.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

appel

Une fois la fonction définie elle peut être utilisée par un autre fonction ou méthode.
La syntaxe à respecter est le nom de la fonction suivie entre parenthèses d'une liste de paramètres en même nombre et même type respectif que dans la définition de la fonction.
Les variables sur lesquelles est *appliquée* la fonction sont dits paramètres réels ou effectifs.
Si la fonction renvoie un résultat elle sera utilisée comme une variable du type renvoyé : affectation, test, dans une autre fonction ou méthode . . ..
Par exemple,

```
z=pgcd(x,y) ;
```

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

passage de paramètres

Règle : en Java les arguments d'une fonction sont passés par valeur, c'est-à-dire les valeurs des paramètres réels sont copiés au moment de l'appel.

I. Structure d'un programme en Java
II. Les types en Java
III. Les types primitifs
IV. La décision
V. L'itération
VI. Classes enveloppantes
VII. Entrées - sorties
VIII. Construire ses propres fonctions

Fonction récursive

Une fonction peut appeler une autre fonction comme `estPremier` utilisant `pgcd`.
Elle peut s'appeler elle-même : on parle alors de fonction récursive.
Par exemple, le calcul de la fonction factorielle s'écrira :

```
import java.util.Scanner ;
public class EssaiFact {
static int Fact(int n){
if (n<=1)
return 1 ;
else return n*Fact(n-1) ;
}
public static void main ( String [ ] arg ) {
int x ;
Scanner saisieClavier = new Scanner(System.in) ;
System.out.print("entrer un entier: x=");
x=saisieClavier.nextInt() ;
System.out.println("x! = "+ Fact(x));
} }
```

I. Structure d'un programme en Java

II. Les types en Java

III. Les types primitifs

IV. La décision

V. L'itération

VI. Classes enveloppantes

VII. Entrées - sorties

VIII. Construire ses propres fonctions

Exercices

1. Ecrire un programme qui saisit un entier n et qui affiche le terme de rang n de la suite de Fibonacci.
2. Ecrire un programme qui saisit un entier n et qui affiche le nombre d'itérations de la suite de Syracuse pour atteindre 1.
3. Ecrire un programme qui saisit un entier seuil et qui affiche le rang k du premier terme de la suite de Fibonacci qui dépasse seuil.