Développement Orienté Objet
R2.01

Jean-Philippe Prost
Jean-Philippe.Prost@univ-amu.fr

IUT d'Aix - Aix-Marseille Université

Principe (Support principal)

Yet Another Insignificant Programming notes (YAIP), de Chua Hock-Chuan :

`https://tinyurl.com/mtz6w63a`

Attention !

- Vous devez **lire** et **travailler** ce support. C'est lui qui constitue le cours.
- Le CM et les transparents qui vont avec ne sont qu'un *résumé*

Introduction

Travail personnel


Voir § xx et § yy : travail personnel

Principe (Autres ressources)

- *Ametice : pour les instructions*
- **Documentation Javadoc de l'API (17)**
  https://docs.oracle.com/en/java/javase/17/docs/api/
- *Ces transparents (sur ametice)*
- *Pour apprendre la frappe dactylo (à 10 doigts, en regardant l'écran et non le clavier) : multiples didactitiels en ligne*

Principe (Encore d'autres ressources...)

- *Tutoriel Oracle (pour les bases seulement : s'appuient sur Java 8) :* `https://docs.oracle.com/javase/tutorial/index.html`
- *Développons en Java. Ensemble très complet de tutoriels Java, de Jean-Michel Doudoux. En français.*

Principe (Outils)

- *Linux*
- *éditeur de texte*
- *JDK 17, ou $\geqslant$ 8 (cf. YAIP pour installation)*
- *JRE même version que le JDK (cf. YAIP)*
- *Eclipse (cf. YAIP)*
- *StarUML*
- *(git, avec GitHub et GitHub classroom)*
  Pour apprendre seul :
  - *Tutoriel Git-it*
  - *Manuel de référence Git, et tutoriel*

Environnement de travail

Les séances de TDP ne sont PAS destinées à l'installation des outils nécessaires sur vos machines personnelles ! !

Pour installer le nécessaire **avant de commencer**, voir sur YAIP (et ailleurs) :

- Le guide de survie du programmeur sur macOS & Ubuntu
- Éditeurs de code source et IDEs
- (Semaine 1) Installer le JDK
- (Semaine 3) Installer Eclipse

Sinon, utiliser **les machines de l'IUT**, ou **la VDI**.

Attention !

Choisissez OBLIGATOIREMENT un login de type `prenomNom` (valable pour tout : git, discord, etc. et n'importe quel contexte où votre login est communiqué)

Pas de `ninja13` ou de `wonderWoman456`...

- Note globale de Travaux Dirigés Pratiques (TDP) (= TP non dédoublés) : qualité et quantité
- Note d'examen final, sur machine

Introduction

Objectifs du Programme National BUT (PN), R2.01

- Compétences ciblées
  - Developper — c'est-a-dire concevoir, coder, tester et integrer — une solution informatique pour un client.
  - Proposer des applications informatiques optimisées en fonction de critères spécifiques : temps d'exécution, précision, consommation de ressources..
- Apprentissages critiques
  - AC11.01 : Implementer des conceptions simples
  - AC11.02 : Elaborer des conceptions simples
  - AC11.03 : Faire des essais et évaluer leurs résultats en regard des spécifications
  - AC12.01 : Analyser un probleme avec methode (decoupage en elements algorithmiques simples, structure de donnees...)
- SAÉ concernées
  - SAE2.01 : Développement d'une application
  - SAE2.02 : Exploration algorithmique d'un problème
- Mots clés : Programmation Orientée Objet, Analyse, Conception Orientée Objet

Introduction

Principe (Un seul secret pour apprendre à programmer)

PROGRAMMER, PROGRAMMER, et encore PROGRAMMER.

Nous utiliserons **Java**, et **UML** (Unified Modelling Language)

Référence

Ch. 2, Java Basics
`https://www3.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html`

Syntaxe Java

Pour maîtriser un langage de prog., il faut maîtriser :

- la syntaxe
- l'Interface de Programmation d'Application (API), pour ne pas réinventer la roue

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons

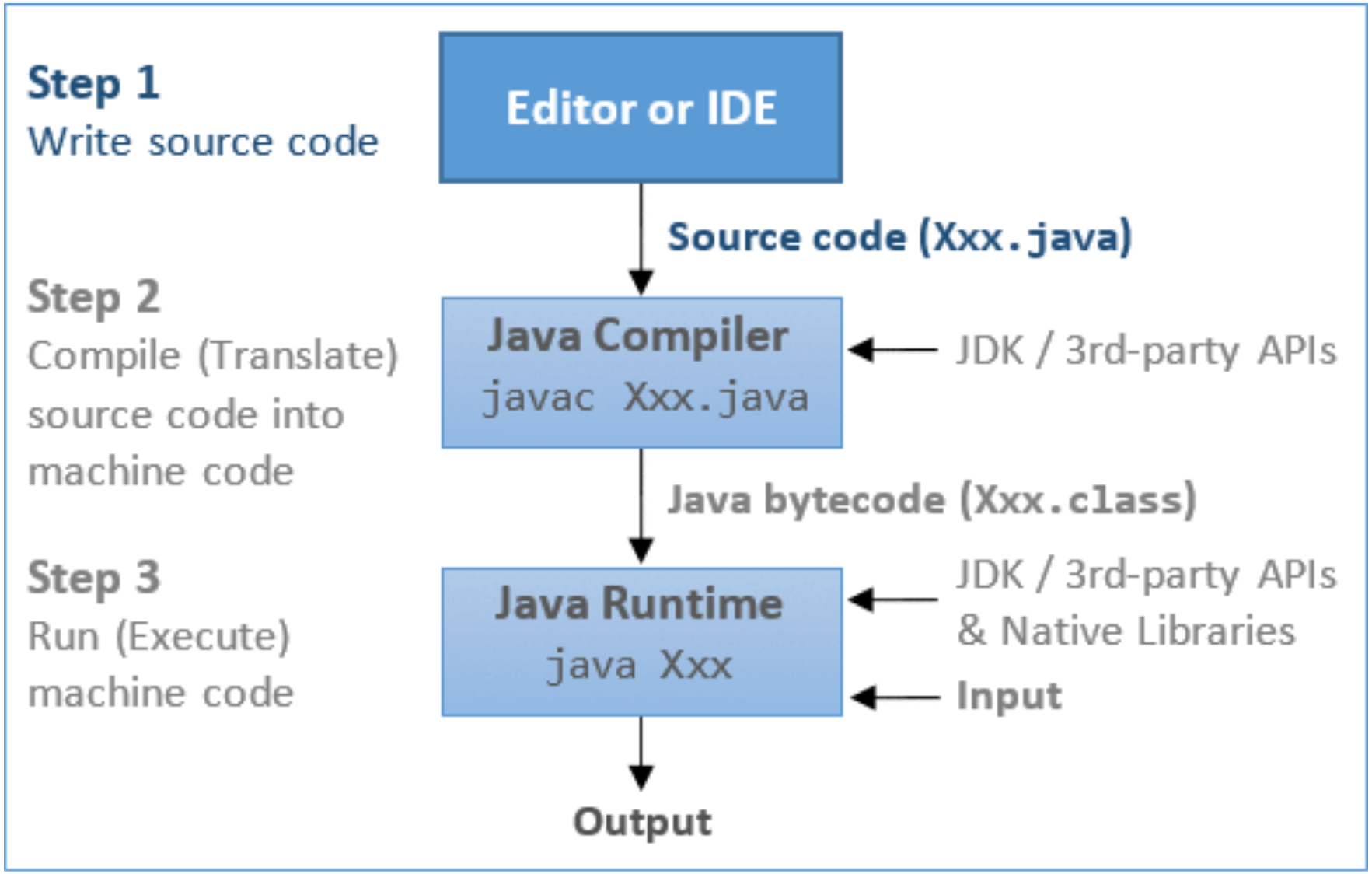

Step 1
Write source code
Editor or IDE
Source code (Xxx.java)
Step 2
Compile (Translate)
source code into
machine code
Java Compiler
javac Xxx.java
JDK / 3rd-party APIs
Java bytecode (Xxx.class)
Step 3
Run (Execute)
machine code
Java Runtime
java Xxx
JDK / 3rd-party APIs
& Native Libraries
Input
Output

Image : YAIP

Syntaxe Java

Commentaires, instructions, blocs, espaces, formatage, etc.


§ 1.2 à § 1.6 : travail personnel et en TDP

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons

Définition (Variable)

Une variable est

- *un espace de stockage* nommé
- *qui stocke une* valeur
- *d'un* type *de donnée particulier.*

Exemple

```
int sum;           // Declare an "int" variable named "sum"
double average;    // Declare a "double" variable named "average"
String message;    // Declare a "String" variable named "message"
char grade;        // Declare a "char" variable named "grade"
```

Java est un langage dit *fortement typé* : on ne mélange pas les torchons et les serviettes

- Une variable de type `int` peut stocker une valeur entière, comme par ex. 12
- Une variable de type `int` ne peut PAS stocker une valeur réelle à virgule flottante, comme par ex. 12,34

Définition (Type)

Un type *de donnée définit :*

- *un domaine de valeurs (associé aux données de ce type)*
- *un ensemble d'opérations possibles (applicables aux données de ce type)*


Java impose des contraintes sur la variable : **apprenez ces contraintes (§ 2.2)**

Voir aussi. . .

[Nommer une variable] Par convention, un nom de variable

- **est un *nom*, ou un *groupe nominal***
- **a son premier mot en minuscule, et les suivants ont leur 1er caractère en majuscule**

Question

Cette convention de nommage a un nom : lequel ?

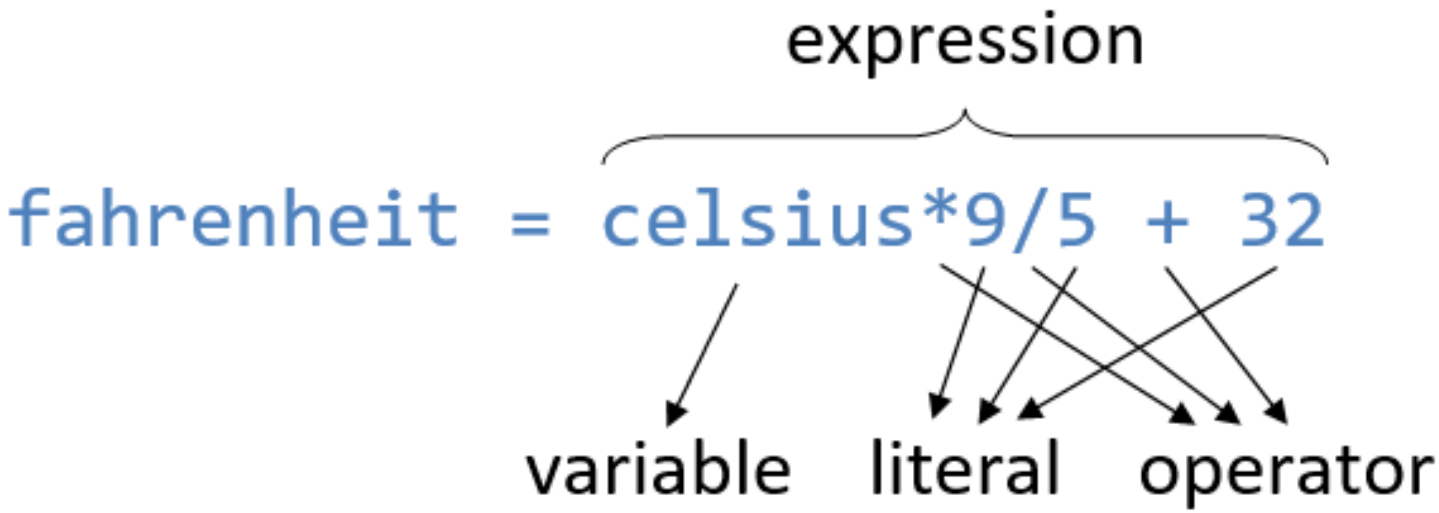
Recommandations

- choisir un nom *auto-descriptif*, qui reflète la sémantique de la variable. Par ex., `numeroEtudiant`, `couleurCarosserie`
- éviter les noms à une seule lettre, par ex. `x`, `y`, `a`, `i3`
- attention au *singulier* et *pluriel* : `etudiant` représente un seul individu, tandis que `etudiants` en représente plusieurs (stockés en général dans une structure, comme une liste ou un tableau).

- Une variable doit être déclarée avant d'être utilisée
- elle doit être déclarée avec un type
- elle peut être déclarée n'importe où dans un programme
- Java est un langage *typé statiquement* : le type est résolu à la compilation, et ne peut pas changer
- une constante est une variable non-modifiable, déclarée par le mot clé `final`
- par convention, une constante se nomme par des mots en majuscule séparés par des '_'. Par ex., `MIN_VALUE`, `MAX_SIZE`, etc.

Définition (Expression)

Une expression *est une combinaison d'*opérateurs *et d'*opérandes *(variables, litéraux), qui peut être* évaluée *et ramenée à* une valeur unique d'un type particulier.



Syntaxe Java

Affectation


Voir § 2.6 sur YAIP

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons

En Java, 2 grandes familles de types de données :

- les types *primitifs*, par ex. `int`, `double`
- les types *références*, par ex. les objets, les tableaux

Syntaxe Java

§ 3.1 Types primitifs

TYPE	DESCRIPTION	
`byte`	**Integer**	8-bit signed integer The range is `[-2^7, 2^7-1] = [-128, 127]`
`short`		16-bit signed integer The range is `[-2^15, 2^15-1] = [-32768, 32767]`
`int`		32-bit signed integer The range is `[-2^31, 2^31-1] = [-2147483648, 2147483647]` (≈9 digits, ±2G)
`long`		64-bit signed integer The range is `[-2^63, 2^63-1] = [-9223372036854775808, 9223372036854775807]` (≈19 digits)
`float`	**Floating-Point Number** F x 2^E	32-bit single precision floating-point number (≈6-7 significant decimal digits, in the range of `±[1.4x10^-45, 3.4028235x10^38]`)
`double`		64-bit double precision floating-point number (≈14-15 significant decimal digits, in the range of `±[4.9x10^-324, 1.7976931348623157x10^308]`)
`char`	**Character** Represented in 16-bit Unicode `'\u0000'` to `'\uFFFF'`. Can be treated as integer in the range of `[0, 65535]` in arithmetic operations. (Unlike C/C++, which uses 8-bit ASCII code.)	
`boolean`	**Binary** Takes a literal value of either `true` or `false`. The size of `boolean` is not defined in the Java specification, but requires at least one bit. `booleans` are used in *test* in decision and loop, not applicable for arithmetic operations. (Unlike C/C++, which uses integer `0` for false, and non-zero for true.)	


(avancé, facultatif) voir §§ 3.2 à 3.4 sur YAIP


YAIP § 3.5 à 3.7

Faire les exercices.

Remarque : ignorer § 3.8

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons


YAIP §§ 4.1 à 4.6

- 4.1 Arithmetic Operators
- 4.2 Arithmetic Expressions
- 4.3 Type Conversion in Arithmetic Operations
- 4.4 More on Arithmetic Operators
- 4.5 Overflow/Underflow
- 4.6 More on Integer vs. Floating-Point Numbers

Exemple

```
// Assign a "double" value to an "int" variable
double d = 3.5;
int i = d;          //Compilation error

// Assign a "float" value to an "int" variable
int sum = 55.66f; //Compilation error

// Assign a "long" value to an "int" variable
long lg = 123;

int count = lg;   //Compilation error
```

Syntaxe Java

§ 4.8 Opérateurs d'affectation composée

Operation	Mode	Usage	Description	Example
`=`	Binary	*var* `=` *expr*	Assignment Assign the LHS value to the RHS variable	`x = 5;`
`+=`	Binary	*var* `+=` *expr* same as: *var* `=` *var* `+` *expr*	Compound addition and assignment	`x += 5;` same as: `x = x + 5`
`-=`	Binary	*var* `-=` *expr* same as: *var* `=` *var* `-` *expr*	Compound subtraction and assignment	`x -= 5;` same as: `x = x - 5`
`*=`	Binary	*var* `*=` *expr* same as: *var* `=` *var* `*` *expr*	Compound multiplication and assignment	`x *= 5;` same as: `x = x * 5`
`/=`	Binary	*var* `/=` *expr* same as: *var* `=` *var* `/` *expr*	Compound division and assignment	`x /= 5;` same as: `x = x / 5`
`%=`	Binary	*var* `%=` *expr* same as: *var* `=` *var* `%` *expr*	Compound modulus (remainder) and assignment	`x %= 5;` same as: `x = x % 5`

Exemple

```
byte b1 = 5, b2 = 8, b3;
b3 = (byte)(b1 + b2);     // byte + byte -> int + int -> int, need to
                          // explicitly cast back to "byte"
b3 = b1 + b2;             // error: RHS is int, cannot assign to byte
b1 += b2;                 // implicitly casted back to "byte"

char c1 = '0', c2;
c2 = (char)(c1 + 2);      // char + int -> int + int -> int, need to
                          // explicitly cast back to "char"
c2 = c1 + 2;              // error: RHS is int, cannot assign to char
c1 += 2;                  // implicitly casted back to "char"
```

Syntaxe Java

§ 4.9 Incrementer/décrementer

Operator	Mode	Usage	Description	Example

++ (Increment)	Unary Prefix
	Unary Postfix
		++x
		x++
			Increment the value of the operand by 1.
			x++ or ++x is the same as x += 1 or x = x + 1
				int x = 5;
				x++; // x is 6
				++x; // x is 7

-- (Decrement)	Unary Prefix
	Unary Postfix
		--x
		x--
			Decrement the value of the operand by 1.
			x-- or --x is the same as x -= 1 or x = x - 1
				int y = 6;
				y--; // y is 5
				--y; // y is 4

Syntaxe Java

§ 4.10 Opérateur relationnels et logiques

Operator	Mode	Usage	Description	Example (x=5, y=8)
==	Binary	`x == y`	Equal to	`(x == y)` ⇒ `false`
!=	Binary	`x != y`	Not Equal to	`(x != y)` ⇒ `true`
>	Binary	`x > y`	Greater than	`(x > y)` ⇒ `false`
>=	Binary	`x >= y`	Greater than or equal to	`(x >= 5)` ⇒ `true`
<	Binary	`x < y`	Less than	`(y < 8)` ⇒ `false`
<=	Binary	`x <= y`	Less than or equal to	`(y <= 8)` ⇒ `true`

Operator	Mode	Usage	Description	Example
!	Unary	`!x`	Logical NOT	
&&	Binary	`x && y`	Logical AND	
||	Binary	`x || y`	Logical OR	
^	Binary	`x ^ y`	Logical Exclusive-OR (XOR)	

Exemple

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)
// wrong to use 0 <= x <= 100

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)
// or
!((x >= 0) && (x <= 100))

// Return true if year is a leap year
// A year is a leap year if it is divisible by 4 but not by 100,
// or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```


F aire les exercices du § 4.10

L'opérateur '+' est *surchargé*, c'est-à-dire qu'il dénote différentes opérations selon le type des opérandes.

Exemple

```
1 + 2 => 3   // int + int => int
1.2 + 2.2 => 3.4   // double + double => double
1 + 2.2 => 1.0 + 2.2 => 3.2  // int + double => double + double => double
'0' + 2 => 48 + 2 => 50   // char + int => int + int => int
                          // (need to cast back to char '2')
```

Exemple

```
"Hello" + "world" => "Helloworld"
"Hi" + ", " + "world" + "!" => "Hi, world!"
```

Exemple

```
"The number is " + 5 => "The number is " + "5" => "The number is 5"
"The average is " + average + "!" (suppose average=5.5) =>
    "The average is " + "5.5" + "!" => "The average is 5.5!"
"How about " + a + b (suppose a=1, b=1) => "How about 1" + b =>
    "How about 11" (left-associative)
"How about " + (a + b) (suppose a=1, b=1) => "How about " + 2 => "How about 2"
```

Exemple

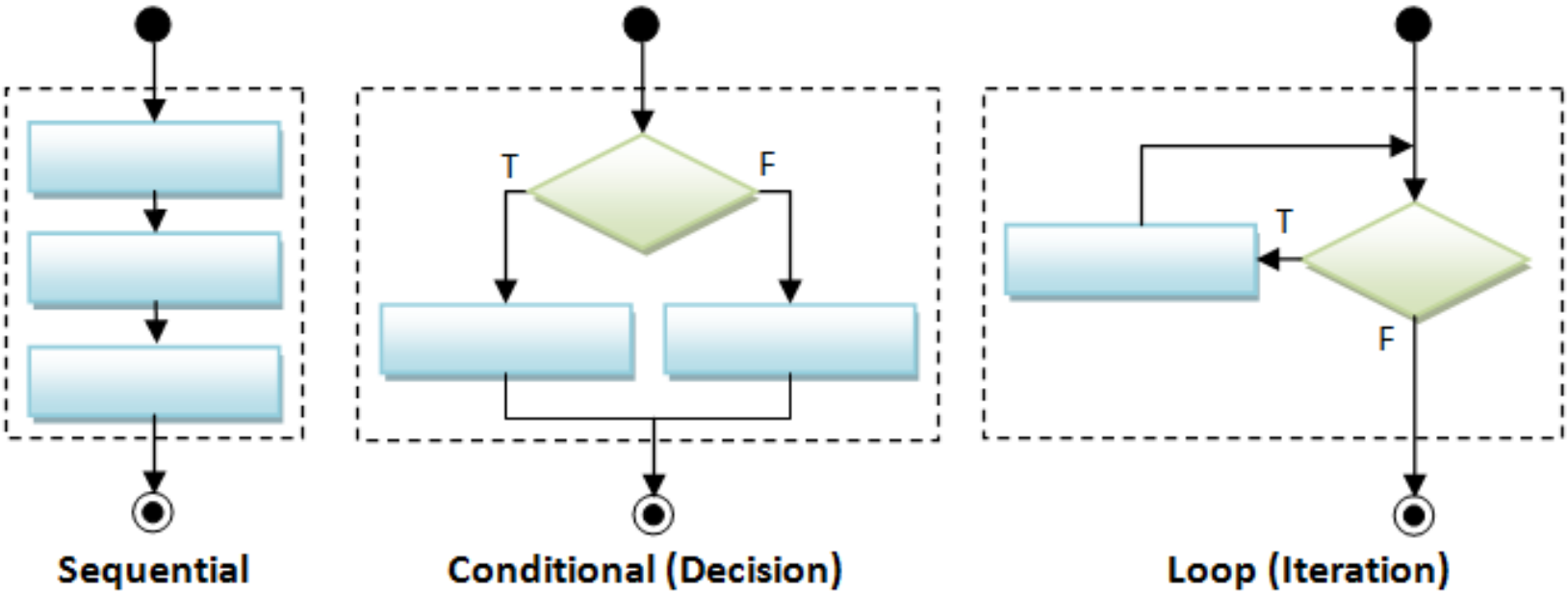
```
System.out.println("The sum is: " + sum);    // Value of "sum" converted
                                              // to String and concatenated
System.out.println("The square of " + input + " is " + squareInput);
```

Remarque

Nous reviendrons plus tard sur le type `String`*, qui est un type spécial.*

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons


T
F
T
F
Sequential
Conditional (Decision)
Loop (Iteration)

RdV sur YAIP...

https://www3.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html#zz-5.2

- it-then et if-then-else
- if imbriqué

Problème : le "else pendant", ou *dangling*-else

```
int i = 0, j = 0;
if (i == 0)          // outer-if
   if (j == 0)       // inner-if
      System.out.println("i and j are zero");
else System.out.println("xxx");   // Ce else peut correspondre au if interieur,
                                  // ou au if exterieur
```

Solution (*dangling*-else)

Java associe un *dangling*-else au if *le plus intérieur*, c'est-à-dire *celui le plus proche*.

Voir § Nested-`if` vs. Sequential-`if` (`if` *imbriqué contre* `if` *séquentiel*)

RdV sur YAIP...

`https://www3.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html#zz-5.2`

- `switch-case-default`
- Expression conditionnelle `(...? ...: ...)`

Faire un maximum d'exercices, pour acquérir les automatismes

RdV sur YAIP...

`https://www3.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html#zz-5.3`

- `while-do`
- `do-while`
- `for`
- exemples de code

Remarque : le `for` avec virgues

L'expression d'initialisation peut contenir plusieurs instructions :

```
// for (init; test; update) { ..... }
for (int row = 0, col = 0; row < SIZE; ++row, ++col) {
   // Process diagonal elements (0,0), (1,1), (2,2),.........
}
```

En revanche le test doit être une expression booléenne qui retourne `true` ou `false`.

System.exit(int *exitCode*)

L'expression d'initialisation peut contenir plusieurs instructions :

```
if (errorCount > 10) {
   System.out.println("too many errors");
   System.exit(1);  // Terminate the program with abnormal exit code of 1
}
```

Par convention, la valeur de retour 0 (zéro) indique une terminaison normale.

return

L'expression d'initialisation peut contenir plusieurs instructions :

```
public static void main(String[] args) {
   ...
   if (errorCount > 10) {
      System.out.println("too many errors");
      return;  // Terminate and return control to Java Runtime from main()
   }
   ...
}
```

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons

```
printf(formattingString, arg1, arg2, arg3, ...);
```

Remarque

La méthode `format` *fait la même chose que* `printf`

- YAIP : §§ 6.1, 6.2
- Javadoc : `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Formatter.html#syntax`

Syntaxe Java

§ 6.1 Entrées/Sorties : formatage des sorties

Example	Output
`// Without specifying field-width` `System.out.printf("Hi,|%s|%d|%f|,@xyz%n", "Hello", 123, 45.6);`	`Hi,|Hello|123|45.600000|,@xyz`
`// Specifying the field-width and decimal places for double` `System.out.printf("Hi,|%6s|%6d|%6.2f|,@xyz%n", "Hello", 123, 45.6);`	`Hi,| Hello|   123| 45.60|,@xyz`

`// Various way to format integers:`
`// flag '-' for left-align, '0' for padding with 0`
`System.out.printf("Hi,|%d|%5d|%-5d|%05d|,@xyz%n", 111, 222, 333, 444);`
	`Hi,|111|  222|333  |00444|,@xyz`

`// Various way to format floating-point numbers:`
`// flag '-' for left-align`
`System.out.printf("Hi,|%f|%7.2f|%.2f|%-7.2f|,@xyz%n", 11.1, 22.2, 33.3, 44.4);`
	`Hi,|11.100000|  22.20|33.30|44.40  |,@xyz`

`// To print a '%', use %% (as % has special meaning)` `System.out.printf("The rate is: %.2f%%.%n", 1.2);`	`The rate is: 1.20%.`

Java prévoit 3 flux d'E/S standards :

- `System.in` (périphérique d'entrée standard, typiquement le clavier)
- `System.out` (périphérique de sortie standard, typiquement l'écran)
- `System.err` (périphérique d'erreur standard, par défaut l'écran, mais il est classique de rediriger vers un fichier)
- `System.in` ne permet de lire que des `String`, il faut donc *parser* ce qui est lu pour en faire un `int`, ou `double`, etc.
- À partir de Java 5, on peut utiliser `java.util.Scanner`

Syntaxe Java

§ 6.2 Entrées/Sorties : lecture d'entrées formatées

```java
import java.util.Scanner;     // Needed to use the Scanner
/**
 * Test input scanner
 */
public class ScannerTest {
  public static void main(String[] args) {
    // Declare variables
    int num1;
    double num2;
    String str;

    // Read inputs from keyboard
    // Construct a Scanner named "in" for scanning System.in (keyboard)
    Scanner in = new Scanner(System.in);
    System.out.print("Enter an integer: ");  // Show prompting message
    num1 = in.nextInt();        // Use nextInt() to read an int
    System.out.print("Enter a floating point: ");  // Show prompting message
    num2 = in.nextDouble();     // Use nextDouble() to read a double
    System.out.print("Enter a string: ");  // Show prompting message
    str  = in.next();  // Use next() to read a String token, up to white space
    in.close();   // Scanner not longer needed, close it

    // Formatted output via printf()
    System.out.printf("%s, Sum of %d & %.2f is %.2f%n",
      str, num1, num2, num1+num2);
  }
}
```

Pour plus d'exemples, voir :


YAIP : §§ 6.3-6.6


YAIP : § 6.7 Exercices

Scanner peut également servir à lire depuis un fichier d'entrée.

Exemple

```
Scanner in = new Scanner(new File("in.txt"));  // Construct a Scanner to scan a
// Use the same set of methods to read from the file
int anInt = in.nextInt();           // next String
double aDouble = in.nextDouble(); // next double
String str = in.next();             // next int
String line = in.nextLine();        // entire line\end{frame}
```

Plan

1. 1. Syntaxe de base
2. 2. Variables et types
3. 3. Types primitifs et `String`
4. 4. Opérations de base
5. 5. Structures de contrôle
6. 6. Entrées/Sorties
7. 7. Écrire des programmes corrects et bons

- Respecter les conventions établies. Vous DEVEZ lire les conventions de codage de Java : `http://www.oracle.com/technetwork/java/codeconv-138413.html`
- Formater et indenter le code correctement (voir les conventions)
- Apprendre les raccourcis claviers de votre éditeur/IDE, par ex. pour formater/indenter le code automatiquement
- Choisir des noms qui ont un sens
- Commenter le code, pour expliquer les concepts importants
- Documenter son programme **en même temps** que son écriture
- Éviter les instructions déstructurantes, comme `break` ou `continue`

Syntaxe Java

Conventions de codage Java

http://www.oracle.com/technetwork/java/codeconv-138413.html

Les erreurs sont généralement de 3 types :

- *Erreur de compilation*. Facile à régler : suivre les messages du compilateur
- *Erreur d'exécution*. Le programme compile, mais échoue à l'exécution. Facile à régler : suivre les messages d'erreur (par ex., les messages d'exception)
- *Erreur logique*. Le programme compile et s'exécute, mais l'exécution ne produit pas le résultat attendu.

1. Commencer par regarder l'écran ! C'est là que ça se passe...
2. Étudier les messages d'erreur ! ! Les lire, au lieu de les ignorer comme s'ils étaient là pour la déco
3. Insérer des messages écrits aux endroits appropriés du code. Souvent suffisant pour des programmes simples, mais rapidement insuffisant, inéfficace et imprécis pour des programmes plus complexes
4. Utiliser un débogueur (graphique). Permet notamment d'exécuter un programme pas-à-pas, en suivant l'état des variables. Demande un temps de prise en main, mais l'effort est inévitable
5. Utiliser des programmes plus avancés, comme un profiler, pour identifier les fuites de mémoire
6. Tester le programme pour supprimer les erreurs logiques