

Syntaxe

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 5 janvier 2021

Syntaxe élémentaire

- Instructions terminées par un point virgule
- Blocs d'instructions
- Commentaires

Bloc

```
{  
    instruction1;  
    instruction2;  
}
```

Commentaires

```
//commentaire sur une ligne  
/* commentaire sur  
    plusieurs lignes */
```

Variables

Déclaration variable

```
<type> nom;
```

Affectation variable

```
nom = valeur;
```

Déclaration et affectation variable

```
<type> nom = valeur;
```

⇒ Essayer avec jshell !

Types primitifs et valeurs primitives

Types primitifs (non exhaustif)

- boolean
- int (32 bits)
- double (64 bits)

Valeurs primitives

- true, false
- littéral entier : 156, 100_000
- littéral flottant : 1.5d, 2d

Type String

- Type String
- Valeur : utiliser guillemets
- Exemple ?

Type String

- Type String
- Valeur : utiliser guillemets
- Exemple? `String ploum = "coucou";`

Portée

- Variables connues en fonction de leur endroit de déclaration
- Connues dans le bloc (après déclaration) (et ceux inclus)
- Inconnues hors du bloc de déclaration

Exemple

```
{  
    int a = 2;  
    {  
        int b = 2;  
        // a ?  
    }  
    // a ?  
    // b ?  
}
```

Bonne pratique : adopter portée la plus étroite possible

Portée

- Variables connues en fonction de leur endroit de déclaration
- Connues dans le bloc (après déclaration) (et ceux inclus)
- Inconnues hors du bloc de déclaration

Exemple

```
{  
    int a = 2;  
    {  
        int b = 2;  
        // a ? Connu  
    }  
    // a ?  
    // b ?  
}
```

Bonne pratique : adopter portée la plus étroite possible

Portée

- Variables connues en fonction de leur endroit de déclaration
- Connues dans le bloc (après déclaration) (et ceux inclus)
- Inconnues hors du bloc de déclaration

Exemple

```
{  
    int a = 2;  
    {  
        int b = 2;  
        // a ? Connue  
    }  
    // a ? Connue  
    // b ?  
}
```

Bonne pratique : adopter portée la plus étroite possible

Portée

- Variables connues en fonction de leur endroit de déclaration
- Connues dans le bloc (après déclaration) (et ceux inclus)
- Inconnues hors du bloc de déclaration

Exemple

```
{  
    int a = 2;  
    {  
        int b = 2;  
        // a ? Connue  
    }  
    // a ? Connue  
    // b ? Inconnue  
}
```

Bonne pratique : adopter portée la plus étroite possible

Opérateurs unaires

Opérateurs unaires :

- renvoie le négatif
- ! Négation logique
- ++ Incrmente un entier (affectation implicite)
- Décrmente un entier (affectation implicite)

Exemples

```
int x = -3;  
int z = -(x - y);  
++i;  
boolean b = !true;  
boolean c = !b;
```

Opérations binaires

`+`, `-`, `*`, `/` NB `/` sur entiers \neq `/` sur double : toujours préciser le type pour meilleure lisibilité

`+` sur `String` : concaténation de chaînes

`%` reste de la division entière

`<`, `>`, `<=`, `>=` compare et renvoie un booléen

`==` teste égalité de valeur et renvoie un booléen

`!=` inverse de `==`

`&&` renvoie vrai ssi les deux opérandes sont vrais (court-circuite)

`||` renvoie vrai ssi au moins un opérande est vrai (court-circuite)

`+=`, `-=` opération puis affectation

Exemple

```
boolean c = (!b && (x > y)) || (a != 3)
```

Tableaux

Tableaux

```
int[] primes = {2, 3, 5, 7, 11, 13, 17};  
int x = primes[0] + 2 * primes[1];
```

```
double[] array = new double[10];  
array[0] = 3d;
```

- Tableaux à plusieurs dimensions possibles
- Dans ce cours on n'utilisera (presque) pas les tableaux

Tests

- `if (test) bloc`
- `if (test) bloc else bloc`
- `if (test) bloc else if (test) bloc else bloc`

Exemple

```
if (value == 3) {  
    openGate = true;  
} else {  
    openGate = false;  
}
```

Amélioration ?

Tests

- `if (test) bloc`
- `if (test) bloc else bloc`
- `if (test) bloc else if (test) bloc else bloc`

Exemple

```
if (value == 3) {  
    openGate = true;  
} else {  
    openGate = false;  
}
```

Amélioration ? `openGate = (value == 3);`

Boucles while

While

```
while (openGate) {  
    ...  
}  
do {  
    ...  
} while (openGate);
```


Boucles for

For

```
int a = 4;
for (int i = 0; i < 3; ++i) {
    a += i;
}
```

Valeur de a?

For each

```
int[] values = {3, 2, 14};
int tot = 0;
for (int v : values) {
    tot += v;
}
```

Boucles for

For

```
int a = 4;
for (int i = 0; i < 3; ++i) {
    a += i;
}
```

Valeur de a ? 7

For each

```
int[] values = {3, 2, 14};
int tot = 0;
for (int v : values) {
    tot += v;
}
```

Méthodes : utilité

- Tout code doit se trouver dans une méthode
- Méthode peut renvoyer une valeur
- Peut être réutilisée, clarifie le code

Méthodes : syntaxe

- `<type de retour> <nom> (<type param1> <nom param1>, ...)`
- `void` pour indiquer absence de retour
- `return` pour renvoyer valeur (exécution méthode cesse)
- Une telle méthode peut être vue comme une fonction

Exemple

```
double weightedSum(double a, double b,  
    double w1, double w2) {  
    return w1 * a + w2 * b;  
}  
  
void useWeightedSum() {  
    double r = weightedSum(2d, 4d, 0.8d, 0.2d);  
}
```

Valeur de r ?

Méthodes : syntaxe

- `<type de retour> <nom> (<type param1> <nom param1>, ...)`
- `void` pour indiquer absence de retour
- `return` pour renvoyer valeur (exécution méthode cesse)
- Une telle méthode peut être vue comme une fonction

Exemple

```
double weightedSum(double a, double b,  
    double w1, double w2) {  
    return w1 * a + w2 * b;  
}  
  
void useWeightedSum() {  
    double r = weightedSum(2d, 4d, 0.8d, 0.2d);  
}
```

Valeur de r ? 2.4d

Classes : méthodes statiques

- Toute méthode doit se trouver dans une classe (sauf avec jshell)
- Syntaxe : `class MyClassName` suivi d'un bloc
- Sert (entre autres) à grouper les méthodes
- En-tête de la méthode commence par `static`

Exemple

```
class MyMathClass {  
    static double sum(double a, double b) {  
        return a + b;  
    }  
    static double mult(double a, double b) {  
        return a * b;  
    }  
}
```

Puis appel avec : `MyMathClass.sum(3d, 1d);`

Classes : variables statiques

Une classe peut aussi contenir des variables statiques

Exemple

```
class MyMathClass {  
    static double w1;  
    static double w2;  
    static double wSum(double a, double b) {  
        return w1 * a + w2 * b;  
    }  
}
```

```
MyMathClass.w1 = 0.8d;  
MyMathClass.w2 = 0.2d;  
double r = MyMathClass.wSum(2d, 4d);
```

Classes : usage

- Référence à une variable statique : nom de la classe point nom de la variable
- Référence à une méthode statique : nom de la classe point nom de la méthode parenthèses paramètres

```
MyMathClass.w1 = 0.8d;  
MyMathClass.w2 = 0.2d;  
double r = MyMathClass.wSum(2d, 4d);
```

Il est donc toujours possible de distinguer une variable d'une méthode

Classes : variables privées

Une classe peut restreindre l'accès à ses variables à elle-même

Exemple

```
class MyMathClass {  
    private static double w1;  
    private static double w2;  
    static void setWeights(double wt1, wt2) {  
        w1 = wt1;  
        w2 = wt2;  
    }  
    static double wSum(double a, double b) {  
        return w1 * a + w2 * b;  
    }  
}
```

Fortement conseillé pour simplifier l'analyse !

Variables et méthodes dans le JDK

De nombreuses méthodes et variables statiques sont définies pour vous dans les classes de la JDK

- `Math.random()`; `Math.abs(-4)`; `Math.PI`;
- `System.out` : une variable pour écrire sur la sortie standard
⇒ `System.out.println("Coucou")`;
- Méthodes pour créer des entiers, par exemple
`Integer.parseInt("10")`;
- Méthodes pour créer des chaînes, par exemple
`String.format("Hello %s, your age is %s.", "Ann", "25")`;

Packages : principe

- Une classe Java a un nom simple
- Exemple : `Math`
- Une classe a aussi un nom complet, « fully qualified name »
- Exemple : `java.lang.Math`, `java.util.Scanner`,
`com.google.common.base.Stopwatch`
- Utilité du nom complet ?
- Chaque classe déclarée dans un *package* Sauf par défaut, non recommandé.
- Nom complet = nom du package point nom de la classe

Packages : principe

- Une classe Java a un nom simple
- Exemple : `Math`
- Une classe a aussi un nom complet, « fully qualified name »
- Exemple : `java.lang.Math`, `java.util.Scanner`,
`com.google.common.base.Stopwatch`
- Utilité du nom complet ? Assurer unicité ! (`MathUtils`)
- Chaque classe déclarée dans un *package* Sauf par défaut, non recommandé.
- Nom complet = nom du package point nom de la classe

Organisation

- Packages structurés hiérarchiquement, comme un arbre
- Structure indiquée par des points
- Organiser par thème (exemple)
- Référence à la classe par son nom complet ou nom simple avec import

Déclaration

```
package io.github.oliviercailloux.math;  
class Maths {  
    static boolean isEven(int a) {  
        return (a % 2) == 0;  
    }  
}
```

- Fichier correspondant dans répertoire dépendant du package!
(Pourquoi?)
 - Son chemin relatif doit être?
 - Relatif à la racine de votre code source
- ⇒ autant de répertoires que de packages dans source
- Chemin relatif du fichier compilé?
 - Relatif à la racine de votre code compilé

Déclaration

```
package io.github.oliviercailloux.math;  
class Maths {  
    static boolean isEven(int a) {  
        return (a % 2) == 0;  
    }  
}
```

- Fichier correspondant dans répertoire dépendant du package!
(Pourquoi? Unicité!)
 - Son chemin relatif doit être?
 - Relatif à la racine de votre code source
- ⇒ autant de répertoires que de packages dans source
- Chemin relatif du fichier compilé?
 - Relatif à la racine de votre code compilé

Déclaration

```
package io.github.oliviercailloux.math;  
class Maths {  
    static boolean isEven(int a) {  
        return (a % 2) == 0;  
    }  
}
```

- Fichier correspondant dans répertoire dépendant du package!
(Pourquoi? Unicité!)
 - Son chemin relatif doit être?
io/github/oliviercailloux/math/Maths.java
 - Relatif à la racine de votre code source
- ⇒ autant de répertoires que de packages dans source
- Chemin relatif du fichier compilé?
 - Relatif à la racine de votre code compilé

Déclaration

```
package io.github.oliviercailloux.math;  
class Maths {  
    static boolean isEven(int a) {  
        return (a % 2) == 0;  
    }  
}
```

- Fichier correspondant dans répertoire dépendant du package!
(Pourquoi? Unicité!)
 - Son chemin relatif doit être?
io/github/oliviercailloux/math/Maths.java
 - Relatif à la racine de votre code source
- ⇒ autant de répertoires que de packages dans source
- Chemin relatif du fichier compilé?
io/github/oliviercailloux/math/Maths.class
 - Relatif à la racine de votre code compilé

Référence

- Au sein d'un package et pour `java.lang` : utiliser le nom simple
- Référence à la classe par son nom complet ?
- Pas pratique !
- « Importer » son nom dans l'espace de noms courants
- Puis référence à la classe par son nom simple

```
package io.github.oliviercailloux.calendar;  
import io.github.oliviercailloux.math.Maths;  
class MyCalendarClass {  
    static boolean isTimeToWork(int dayNb) {  
        return Maths.isEven(dayNb);  
    }  
}
```

Référence

- Au sein d'un package et pour `java.lang` : utiliser le nom simple
- Référence à la classe par son nom complet ?
`io.github.oliviercailloux.math.Maths.isEven(4);`
- Pas pratique !
- « Importer » son nom dans l'espace de noms courants
- Puis référence à la classe par son nom simple

```
package io.github.oliviercailloux.calendar;  
import io.github.oliviercailloux.math.Maths;  
class MyCalendarClass {  
    static boolean isTimeToWork(int dayNb) {  
        return Maths.isEven(dayNb);  
    }  
}
```

License

This presentation, and the associated \LaTeX code, are published under the MIT license. Feel free to reuse (parts of) the presentation, under condition that you cite the author. Credits are to be given to Olivier Cailloux, Université Paris-Dauphine.