

Programmation Objet – Initiation à Java

Letícia SEIXAS PEREIRA

Cours 02- Éléments de syntaxe

21/10/2019



Éléments de syntaxe

- Déclaration et affectation
- Expressions
- Structures de contrôle

Argument de la ligne de commande

- Dans l'en-tête d'une méthode *main* apparaît ce qui s'appelle un paramètre qui est ci-dessous nommé **args**:

```
public class Bonjour {  
    public static void main(String[] args) {  
        System.out.println("Bonjour le monde!");  
    }  
}
```

Argument de la ligne de commande

- Dans l'en-tête d'une méthode *main* apparaît ce qui s'appelle un paramètre qui est ci-dessous nommé **args**:

```
public class Bonjour {  
  
    public static void main(String[] listeArguments) {  
  
        System.out.println("Bonjour le monde!");  
  
    }  
  
}
```

- On peut nommer ce paramètre selon son propre choix : par ex: **listeArguments**
- Cet argument est de type tableau de String ou encore tableau de chaînes de caractères.
- Grâce à ce paramètre, on peut « *passer des arguments à la méthode main* »

```
> java NomDeLaClasse arg0 arg1 ... argN
```

Argument de la ligne de commande

- Exemple:

```
public class Bonjour {  
    public static void main(String[] args) {  
        System.out.println(args[0]);  
    }  
}
```

```
> java Bonjour Leticia
```

```
> java Bonjour "Bonjour le monde!"
```

```
> java NomDeLaClasse arg0 arg1 ... argN
```

Éléments de syntaxe- Déclaration et affectation

Déclaration

<type de la variable> <nom de la variable>

```
double x; // déclaration de la variable x de type int  
char y;  
  
int age;  
  
int qtEtudiants;  
  
int qtEtudiantsM1;
```

Éléments de syntaxe- Déclaration et affectation

Déclaration

`<type de la variable> <nom de la variable>`

- Les déclarations de variables d'un même type peuvent être *factorisées*:

```
double x; // déclaration de la variable x de type int
char y;

int age;

int qtEtudiants;

int qtEtudiantsM1;
```

Éléments de syntaxe - Déclaration et affectation

Déclaration

`<type de la variable> <nom de la variable>`

- Les déclarations de variables d'un même type peuvent être *factorisées*:

```
double x;  
  
char y;  
  
int age;  
  
int qtEtudiants;  
  
int qtEtudiantsM1;
```

```
double x;  
  
char y;  
  
int age, qtEtudiants, qtEtudiantsM1;
```


Éléments de syntaxe- Déclaration et affectation

Portée des variables

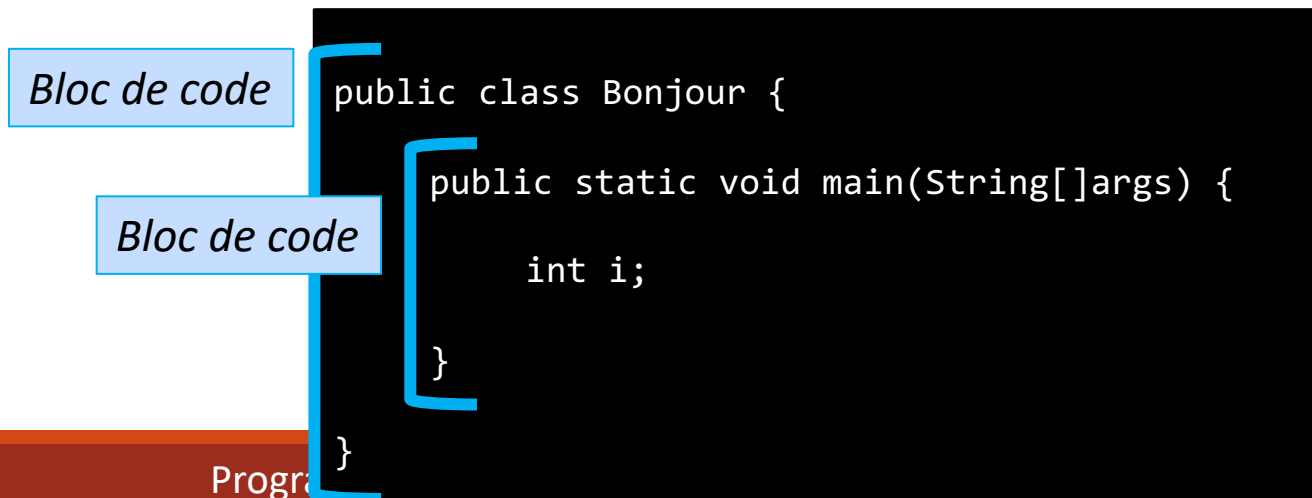
- Une variable est nécessairement déclarée dans un *bloc de code* délimité par des accolades;

```
public class Bonjour {  
    public static void main(String[] args) {  
        int i;  
    }  
}
```

Éléments de syntaxe- Déclaration et affectation

Portée des variables

- Une variable est nécessairement déclarée dans un *bloc de code* délimité par des accolades;
- Cette variable a alors une *portée* limitée à ce bloc => elle n'est ni visible, ni accessible à l'*extérieure* de ce bloc.



Éléments de syntaxe- Déclaration et affectation

Affectation

`<nom de la variable> = <valeur>`

```
char y;  
int qtEtudiants;  
  
y = 'L';  
qtEtudiants = 15;
```

```
char y = 'L';  
int qtEtudiants = 15;
```

Éléments de syntaxe - Déclaration et affectation

Affectation

`<nom de la variable> = <valeur>`

```
public class Cours{  
    public static void main(String[] args) {  
        String nomEtudiant = "Marie Curie";  
        double noteEtudiant = 19.4;  
    }  
}
```

Éléments de syntaxe- Déclaration et affectation

Affectation

- Le compilateur détecte et rejette l'incompatibilité de type entre la variable et la valeur affectée:

Éléments de syntaxe- Déclaration et affectation

Affectation

- Le compilateur détecte et rejette l'incompatibilité de type entre la variable et la valeur affectée:

```
public class Cours{  
  
    public static void main(String[] args) {  
  
        String nomEtudiant = 19.4;  
        double noteEtudiant = "Marie Curie";  
  
    }  
}
```

Éléments de syntaxe - Déclaration et affectation

Affectation

- Le compilateur détecte et rejette l'incompatibilité de type entre la variable et la valeur affectée:

```
public class Cours{  
  
    public static void main(String[] args) {  
  
        String nomEtudiant = 19.4;  
        double noteEtudiant = "Marie Curie";  
  
    }  
}
```



```
> javac Cours.java  
Cours.java:6: error: illegal character: '\u201c'  
double noteEtudiant = "Marie Curie";  
                        ^  
Cours.java:6: error: illegal character: '\u201d'  
double noteEtudiant = "Marie Curie";  
                        ^  
2 errors
```

Éléments de syntaxe- Déclaration et affectation

Affectation

Initialisation obligatoire des variables locales:

- Les variables locales n'ont pas de valeur initiale par défaut:

```
public class Cours{  
  
    public static void main(String[] args) {  
  
        String nomEtudiant;  
        double noteEtudiant;  
  
        System.out.println(nomEtudiant);  
        System.out.println(noteEtudiant);  
  
    }  
}
```


Éléments de syntaxe- Déclaration et affectation

Affectation

Initialisation obligatoire des variables locales:

- Les variables locales n'ont pas de valeur initiale par défaut:

```
public class Cours{  
  
    public static void main(String[] args) {  
  
        String nomEtudiant;  
        double noteEtudiant;  
  
        System.out.println(nomEtudiant);  
        System.out.println(noteEtudiant);  
  
    }  
}
```



```
> javac Cours.java  
Cours.java:8: error: variable nomEtudiant might not have been  
initialized  
    System.out.println(nomEtudiant);  
                        ^  
Cours.java:9: error: variable noteEtudiant might not have been  
initialized  
    System.out.println(noteEtudiant);  
                        ^  
2 errors
```

Éléments de syntaxe- Déclaration et affectation

Affectation

Initialisation obligatoire des variables locales:

- Les variables locales n'ont pas de valeur initiale par défaut: **Il faut donc explicitement les initialiser!**

```
public class Cours{  
  
    public static void main(String[] args) {  
  
        String nomEtudiant = "Marie Curie";  
        double noteEtudiant = 19.4;  
  
        System.out.println(nomEtudiant);  
        System.out.println(noteEtudiant);  
  
    }  
}
```



```
> javac Cours.java  
> java Cours  
Marie Curie  
19.4
```

Opérateurs

- **Opérateurs de calcul / Opérateurs arithmétiques:**
 - + Addition de deux variables numériques / Concaténation des chaînes de caractères (String);
 - - Soustraction de deux variables numériques;
 - * Multiplication de deux variables numériques;
 - / Division de deux variables numériques;
 - % Renvoi du reste de la division entière de deux variables numériques « modulo ».

Opérateurs

- **Opérateurs de calcul / Opérateurs arithmétiques:**

```
int nbre1, nbre2, nbre3;  
nbre1 = nbre2 = nbre3 = 0;  
  
nbre1 = nbre1 + 1;  
nbre1 = nbre1 + 1;  
nbre2 = nbre1;  
nbre2 = nbre2 * 2;  
nbre3 = nbre2;  
nbre3 = nbre3 / nbre3;  
nbre1 = nbre3;  
nbre1 = nbre1 - 1;
```

Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1, n2, n3, n4;  
        n1 = n2 = n3 = n4 = 0;  
  
        n1 = n1 + 1;  
        n2 += 1;  
        n3++; // Post-incrémentation  
        ++n4; // Pré-incrémentation  
        //Affichez les trois nombres  
  
    }  
}
```

Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1, n2, n3, n4 = 0;  
        n1 = n2 = n3 = n4 = 0;  
  
        n1 = n1 + 1;  
        n2 += 1;  
        n3++; // Post-incrémentation  
        ++n4; // Pré-incrémentation  
        //Affichez les trois nombres  
  
    }  
}
```



```
> javac Operateurs.java  
> java Operateurs  
1  
1  
1  
1
```

Opérateurs

- **Opérateurs d'incrémentation et de décrémentation:**
 - Incrémentation ++
 - Décrémentation --
- Opérateurs *unaires* qui s'appliquent à une *variable* de type numérique;
- Ces opérateurs incrémentent ou décrémentent d'une unité la valeur contenue dans la variable.

Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```


Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```



```
> javac Operateurs.java  
> java Operateurs  
1  
2
```

Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Opérateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        n1++;  
        n2--;  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```



```
> javac Opérateurs.java  
> java Opérateurs  
2  
1
```

Opérateurs

- Opérateurs d'incrémentation et de décrémentation:

```
public class Opérateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        n1++; // n1 = n1+1  
        n2--; // n2 = n2-1  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```



```
> javac Opérateurs.java  
> java Opérateurs  
2  
1
```

Opérateurs

- **Opérateurs d'incrémentation et de décrémentation:**

- Post-incrémentation:

`i++`

- Pré-incrémentation:

`++ i`

- Post-décrémentation:

`i--`

- Pré-décrémentation :

`-- i`

Opérateurs

- **Opérateurs d'incrémentation et de décrémentation:**

- Pré-incrémentation:

`++ i` (Renvoie comme résultat la valeur de la variable **après** l'incrémentation)

Opérateurs

- **Opérateurs d'incrémentation et de décrémentation:**

- Pré-incrémentation:

`++ i` (Renvoie comme résultat la valeur de la variable **après** l'incrémentation)

- Post-incrémentation:

`i++` (Renvoie comme résultat la valeur de la variable **avant** l'incrémentation)

Opérateurs

- **Opérateurs d'incrémentation et de décrémentation:**

- Pré-incrémentation:

`++ i` (Renvoie comme résultat la valeur de la variable **après** l'incrémentation)

```
i = ++j;
```

```
// j = j+1; i=j;
```

- Post-incrémentation:

`i++` (Renvoie comme résultat la valeur de la variable **avant** l'incrémentation)

```
i = j++;
```

```
// i = j; j = j+1;
```

Opérateurs

- **Opérateurs d'affectation**

Exemple (Operateurs.java)

```
public class Operateurs{

    public static void main(String[] args) {

        int n1 = 1;
        int n2 = 2;

        n2 = ++n1;
        System.out.println(n1);
        System.out.println(n2);

    }
}
```


Opérateurs

- Opérateurs d'affectation

Exemple (Operateurs.java)


```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        n2 = ++n1;  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```

++ i (Renvoie comme résultat la valeur de la variable **après** l'incrément)

n2 = ++n1;

n1 = n1 + 1; // n1 = 1 + 1 = 2

n2 = n1; // n2 = 2



2
2

Opérateurs

- **Opérateurs d'affectation**

Exemple (Operateurs.java)

```
public class Operateurs{

    public static void main(String[] args) {

        int n1 = 1;
        int n2 = 2;

        n2 = n1++;
        System.out.println(n1);
        System.out.println(n2);

    }
}
```

Opérateurs

- Opérateurs d'affectation

Exemple (Operateurs.java)


```
public class Operateurs{  
  
    public static void main(String[] args) {  
  
        int n1 = 1;  
        int n2 = 2;  
  
        n2 = n1++;  
        System.out.println(n1);  
        System.out.println(n2);  
  
    }  
}
```

i++ (Renvoie comme résultat la valeur de la variable **avant** l'incrément)

n2 = n1++;

n2 = n1; // n2 = 1

n1 = n1 + 1; // n1 = 1 + 1 = 2



2
1

Opérateurs

- Opérateurs d'affectation

Opérateur	Exemple	Equivalence
=	i = 90	
+=	i += 20	i = i + 20
-=	i -=10	i = i -10
*=	i *= 2.5	i = 1 * 2.5
/=	i /= 10	i = i / 10
%=	i %= 10	i = i % 10
++	i++ / ++i	i = i +1 / i +=1
--	i-- / --i	i = i -1 / i -= 1

Opérateurs

- Opérateurs logiques

Opérateur	Signification
$x > y$	Strictement supérieur
$x < y$	Strictement inférieur
$x \geq y$	Supérieur ou égal
$x \leq y$	Inférieur ou égal
$x == y$	Égal
$x != y$	Différent

Opérateurs

- Opérateurs logiques

Opérateur	Signification
<code>x > y</code>	Strictement supérieur
<code>x < y</code>	Strictement inférieur
<code>x >= y</code>	Supérieur ou égal
<code>x <= y</code>	Inférieur ou égal
<code>x == y</code>	Égal
<code>x != y</code>	Différent

Opérateur	Signification
<code>x & y</code>	ET logique
<code>x y</code>	OU logique
<code>x ? y : z</code>	L'opérateur ternaire
<code>!x</code>	L'opérateur de négation

Opérateurs

- Opérateurs logiques

Opérateur	Signification
<code>x > y</code>	Strictement supérieur
<code>x < y</code>	Strictement inférieur
<code>x >= y</code>	Supérieur ou égal
<code>x <= y</code>	Inférieur ou égal
<code>x == y</code>	Égal
<code>x != y</code>	Différent

Opérateur	Signification
<code>x & y</code>	ET logique
<code>x y</code>	OU logique
<code>x ? y : z</code>	L'opérateur ternaire
<code>!x</code>	L'opérateur de négation
<code>x & y</code>	ET binaire
<code>x ^ y</code>	OU exclusif binaire
<code>x y</code>	OU binaire

Opérateurs

- Opérateurs logiques

Opérateur	Signification
<code>x > y</code>	Strictement supérieur
<code>x < y</code>	Strictement inférieur
<code>x >= y</code>	Supérieur ou égal
<code>x <= y</code>	Inférieur ou égal
<code>x == y</code>	Égal
<code>x != y</code>	Différent

Opérateur	Signification
<code>x & y</code>	ET logique
<code>x y</code>	OU logique
<code>x ? y : z</code>	L'opérateur ternaire
<code>!x</code>	L'opérateur de négation
<code>x & y</code>	ET binaire
<code>x ^ y</code>	OU exclusif binaire
<code>x y</code>	OU binaire
<code>x && y</code>	ET logique (Court-circuit)
<code>x y</code>	OU logique (Court-circuit)

Opérateurs

- **Priorité des opérateurs:**
 - Java définit les priorités dans les opérateurs comme suit (du plus prioritaire au moins prioritaire):

les parenthèses	()
les opérateurs d'incrémentation	++ --
les opérateurs de multiplication, division et modulo	* / %
les opérateurs d'addition et soustraction	+ -
les opérateurs de comparaison	< > <= >=
les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	
l'opérateur ET logique	&&
l'opérateur OU logique	
les opérateurs d'assignement	= += -=

Opérateurs

- Opérateurs logiques

Opérateur	Signification
<code>x > y</code>	Strictement supérieur
<code>x < y</code>	Strictement inférieur
<code>x >= y</code>	Supérieur ou égal
<code>x <= y</code>	Inférieur ou égal
<code>x == y</code>	Égal
<code>x != y</code>	Différent

Opérateur	Signification
<code>x & y</code>	ET logique
<code>x y</code>	OU logique
<code>x ? y : z</code>	L'opérateur ternaire
<code>!x</code>	L'opérateur de négation
<code>x & y</code>	ET binaire
<code>x ^ y</code>	OU exclusif binaire
<code>x y</code>	OU binaire
<code>x && y</code>	ET logique (Court-circuit)
<code>x y</code>	OU logique (Court-circuit)

Opérateurs

- (| OU) (& ET) (^ OU exclusive) Binaire – Opérations bit à bit

Opérateurs

- **(| OU) (& ET) (^ OU exclusive) Binaire – Opérations bit à bit**
 - L'opérateur AND (&) bit à bit retourne la valeur 1 (un) si les deux bits correspondants sont égaux à 1(un), sinon il retourne la valeur zéro.

Opérateurs

- **(| OU) (& ET) (^ OU exclusive) Binaire – Opérations bit à bit**
 - L'opérateur AND (&) bit à bit retourne la valeur 1 (un) si les deux bits correspondants sont égal a 1(un), sinon il retourne la valeur zéro.
 - L'opérateur OR (|) bit à bit retourne la valeur 1 (un) si l'un des deux bits correspondants sont égal a 1(un), sinon il retourne la valeur zéro.

Opérateurs

- **(| OU) (& ET) (^ OU exclusive) Binaire – Opérations bit à bit**
 - L'opérateur AND (&) bit à bit retourne la valeur 1 (un) si les deux bits correspondants sont égal a 1(un), sinon il retourne la valeur zéro.
 - L'opérateur OR (|) bit à bit retourne la valeur 1 (un) si l'un des deux bits correspondants sont égal a 1(un), sinon il retourne la valeur zéro.
 - L'opérateur XOR / Ou exclusive (^) bit à bit retourne la valeur 1 (un) si les deux bits correspondants sont égal a 1(un), mais pas les deux.

Opérateurs

- Opérateurs logiques

Opérateur	Signification
<code>x > y</code>	Strictement supérieur
<code>x < y</code>	Strictement inférieur
<code>x >= y</code>	Supérieur ou égal
<code>x <= y</code>	Inférieur ou égal
<code>x == y</code>	Égal
<code>x != y</code>	Différent

Opérateur	Signification
<code>x & y</code>	ET logique
<code>x y</code>	OU logique
<code>x ? y : z</code>	L'opérateur ternaire
<code>!x</code>	L'opérateur de négation
<code>x & y</code>	ET binaire
<code>x ^ y</code>	OU exclusif binaire
<code>x y</code>	OU binaire
<code>x && y</code>	ET logique (Court-circuit)
<code>x y</code>	OU logique (Court-circuit)

Opérateurs

- **OU & ET logiques (&&, ||)**
 - En travaillant avec les opérateurs logiques on rencontre un comportement appelé « court-circuit »
 - Cela signifie que l'évaluation de l'expression sera poursuivie jusqu'à ce que la vérité ou la fausseté de l'expression soit déterminée sans ambiguïté.
 - Certaines parties d'une expression logique peuvent ne pas être évaluées.
 - Cela signifie que lorsque la valeur de la première opérande permet de déduire la valeur de l'expression, alors la deuxième opérande n'est pas évaluée.

Opérateurs

- **OU & ET logiques (&&, ||)**
 - En travaillant avec les opérateurs logiques on rencontre un comportement appelé « court-circuit »
 - Cela signifie que l'évaluation de l'expression sera poursuivie jusqu'à ce que la vérité ou la fausseté de l'expression soit déterminée sans ambiguïté.
 - Certaines parties d'une expression logique peuvent ne pas être évaluées.
 - Cela signifie que lorsque la valeur de la première opérande permet de déduire la valeur de l'expression, alors la deuxième opérande n'est pas évaluée.
 - **a && b**, si **a** vaut *false*, **b** n'est pas évalué et l'expression vaut *false*
 - **a || b**, si **a** vaut *true*, **b** n'est pas évalué et l'expression vaut *true*

Opérateurs

- **OU & ET logiques (&&, ||)**
 - En travaillant avec les opérateurs logiques on rencontre un comportement appelé « court-circuit »
 - Cela signifie que l'évaluation de l'expression sera poursuivie jusqu'à ce que la vérité ou la fausseté de l'expression soit déterminée sans ambiguïté.
 - Certaines parties d'une expression logique peuvent ne pas être évaluées.
 - Cela signifie que lorsque la valeur de la première opérande permet de déduire la valeur de l'expression, alors la deuxième opérande n'est pas évaluée.
 - Dans le cas contraire (& , |), l'expression logique sera évaluée en totalité même si la vérité ou la fausseté de l'expression est déjà déterminée.

Opérateurs

- OU & ET logiques (&&, ||)

```
public class CourtCircuit{

    public static void main(String[] args) {

        int i = 5;
        int j = 2;
        if ((i > 2) | (j++ > 10)) {
            System.out.println (j);
        }

    }

}
```

```
public class CourtCircuit{

    public static void main(String[] args) {

        int i = 5;
        int j = 2;
        if ((i > 2) || (j++ > 10)) {
            System.out.println (j);
        }

    }

}
```

Opérateurs

L'opérateur conditionnel ternaire

`(condition) ? Valeur-vrai : valeur-faux;`

Opérateurs

L'opérateur conditionnel ternaire

`(condition) ? Valeur-vrai : valeur-faux;`

`a < b ? a : b;`

Opérateurs

L'opérateur conditionnel ternaire

`(condition) ? Valeur-vrai : valeur-faux;`

`a < b ? a : b;`

```
public class MiniVal{  
    public static void main(String[] args) {  
  
        int minVal, a=3, b=2;  
        minVal = a < b ? a : b;  
        System.out.println("min = " + minVal);  
    }  
}
```

(En utilisant l'opérateur ternaire, écrivez un programme pour déterminer si un nombre est pair.

L'opérateur conditionnel ternaire

(condition) ? Valeur-vrai : valeur-faux;

a < b ? a : b;

```
public class MiniVal{  
    public static void main(String[] args) {  
  
        int minVal, a=3, b=2;  
        minVal = a < b ? a : b;  
        System.out.println("min = " + minVal);  
    }  
}
```

Conversions / transtypage

- Transtypage = la conversion des types
- Convertir une donnée d'un type primitif vers un autre type primitif.

Conversions / transtypage

- Transtypage = la conversion des types
 - Convertir une donnée d'un type primitif vers un autre type primitif.
 - Il faut faire attention lors de ces conversions car il y a risque de perdre de l'information.

Conversions / transtypage

- Transtypage = la conversion des types
 - Convertir une donnée d'un type primitif vers un autre type primitif.
 - Il faut faire attention lors de ces conversions car il y a risque de perdre de l'information.

Exemple: La conversion d'un nombre réel en nombre entier ne peut se réaliser qu'en supprimant les nombres situés après la virgule et en ne gardant que la partie entière du nombre

Conversions / transtypage

- Transtypage = la conversion des types
 - Convertir une donnée d'un type primitif vers un autre type primitif.
 - Il faut faire attention lors de ces conversions car il y a risque de perdre de l'information.

Conversions / transtypage

- Transtypage = la conversion des types
- **Le cast:**
 - La conversion explicite avec perte d'information:

Conversions / transtypage

- Transtypage = la conversion des types
- **Le cast:**
 - La conversion explicite avec perte d'information
 - Placer le type de conversion désiré devant la variable ou l'opération:

```
double i = 1.23;  
double j = 2.99999999;  
int k = (int)i; //k vaut 1  
k = (int)j; //k vaut 2
```

Conversions / transtypage

- Transtypage = la conversion des types

byte -> short -> int -> long -> float -> double

Type	Taille (en bits)
byte	8 (1 o)
short	16 (2 o)
int	32 (4 o)
float	32 (4 o)
long	64 (8 o)
double	64 (8 o)

Conversions / transtypage

- Transtypage = la conversion des types
- Il est toujours possible de convertir un byte en long ou un int en float.
- Il est impossible de transformer un float en short sans perte d'information.

byte -> short -> int -> long -> float -> double

Type	Taille (en bits)
byte	8 (1 o)
short	16 (2 o)
int	32 (4 o)
float	32 (4 o)
long	64 (8 o)
double	64 (8 o)

Promotion arithmétique

- Conversion *implicite*
- Ces conversions transforment *implicitement* une variable <type> en une variable <type2>, lorsque le contexte l'exige:

Promotion arithmétique

- Conversion *implicite*
- Ces conversions transforment *implicitement* une variable <type> en une variable <type2>, lorsque le contexte l'exige:

```
int i = 1;
```

```
double j = 2.99999999;
```

```
int k = i + j;
```



La valeur de <int i> : double

Promotion arithmétique

- Les opérateurs qui convertissent leurs opérandes sont :
 - Les opérateurs unaires -, +, -- et ++, ils convertissent les types `char`, `byte` et `short` en `int` automatiquement. Les restes conservent leurs types.
 - Les opérateurs +, -, *, /, %, <, <=, >, >=, == et !=. Il faut regarder le type des deux opérandes, et appliquer ces règles dans l'ordre :
 1. Si l'un des opérandes est de type `double`, l'autre est converti en `double` ;
 2. Si l'un des opérandes est de type `float`, l'autre est converti en `float` ;
 3. Si l'un des opérandes est de type `long`, l'autre est converti en `long` ;
 4. Dans tous les autres cas, les deux opérandes sont convertis en `int`.

Structures de contrôle

- Alternatives
 - if-else

```
if (<condition>){  
    <block1>;  
}  
else {  
    <block2>;  
}
```

Structures de contrôle

- Alternatives
 - if-else

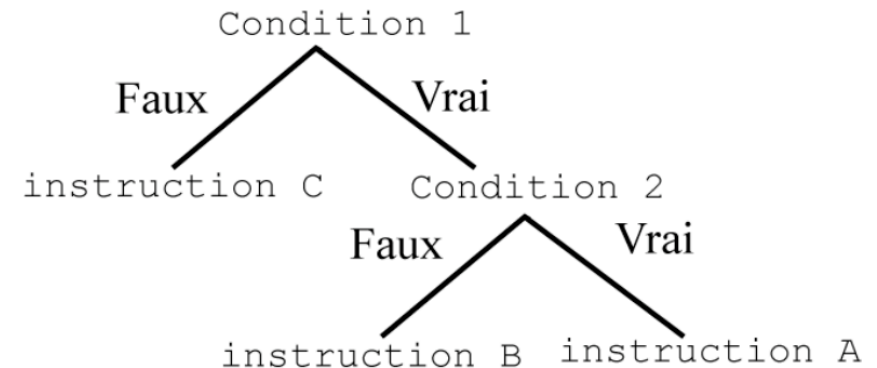
```
if (<condition>){  
    <block1>;  
}  
else {  
    <block2>;  
}
```

(Regarder les opérations relatives à chaque type de données utilisée – la condition doit être toujours booléenne)

Structures de contrôle

- Alternatives
 - if-else

```
if (<condition>){  
    <block1>;  
}  
else {  
    <block2>;  
}
```



(Regarder les opérations relatives à chaque type de données utilisée – la condition doit être toujours booléenne)

Structures de contrôle

- Alternatives
 - switch

```
switch (<exp>){  
    case <val0>: <block0>;  
    case <val1>: <block1>;  
    default: <blockn>;  
}
```

(Le type de la variable valeur ne peut être que char ou int, byte, short ou long)

Structures de contrôle

- Alternatives
 - switch

```
switch (<exp>){  
    case <val0>: <block0>;  
    case <val1>:  
        <block1>;  
        break; // facultatif, pour sortir du bloc switch  
    ...  
    default: <blockn>;  
}
```

(Le type de la variable valeur ne peut être que char ou int, byte, short ou long)

Structures de contrôle

- Alternatives

- switch

```
...
int option = 2;

String reponse;

switch (option) {
    case 1:
        reponse = "Option 1";
        break;
    case 2:
        reponse = "Option 2";
        break;
    case 3:
        reponse = "Option 3";
        break;
    default:
        reponse = "Option invalide";
        break;
}
System.out.println(reponse);
...
```


Structures de contrôle

- Boucles
 - while

```
while (<condition>){  
    <block0>;  
}
```

Structures de contrôle

- Boucles
 - while

```
int i = 10;
```

```
while (i > 1){  
    System.out.println(i);  
    i--;  
}
```

Structures de contrôle

- Boucles

- while

```
int i = 10;
```

```
while (i > 1){  
    System.out.println(i);  
    i--;  
}
```



10
9
8
7
6
5
4
3
2

Structures de contrôle

- Boucles
 - while

```
int i = 10;
```

```
while (i > 1){  
    System.out.println(i);  
    i++;  
}
```



?

Structures de contrôle

- Boucles
 - do-while

```
do {  
    <block>  
}  
while (<condition>;
```

Structures de contrôle

- Boucles
 - do-while

```
int i = 10;
```

```
do {  
    System.out.println(i);  
    i--;  
}
```

```
while (i > 1);
```



10
9
8
7
6
5
4
3
2

Structures de contrôle

- Boucles
 - do-while

```
int i = 10;
```

```
do {  
    System.out.println(i);  
    i++;  
}
```



?

```
while (i < 1);
```

Structures de contrôle

- Boucles
 - for

```
for (<initialisation>; <condition>; incrément){  
    <block>;  
}
```


Structures de contrôle

- Boucles
 - for

```
int tableau[]={2,11,45,9};  
  
for(int i = 0; i < tableau.length; i++){  
    System.out.println(tableau[i]);  
}
```

Structures de contrôle

- Boucles
 - for

```
int tableau[]={2,11,45,9};
```

```
for(int i = 0; i < tableau.length; i++){  
    System.out.println(tableau[i]);  
}
```



```
2  
11  
45  
9
```