

# R1.01 : Initiation au développement

## Cours 8

M.Adam, N.Delomez, JF.Kamp, L.Naert

IUT de Vannes

13 octobre 2022

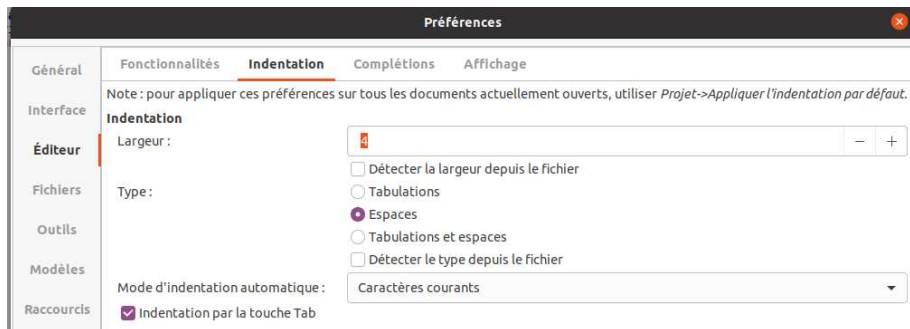
- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

Chaque éditeur, chaque environnement de travail, geany, atom, visual studio code, emacs, etc. possède ses avantages et ses inconvénients. Aucun n'est parfait, car aucun ne possède toutes les qualités que chacun recherche.

Par contre pour chacun, il est intéressant de connaître quelques paramètres :

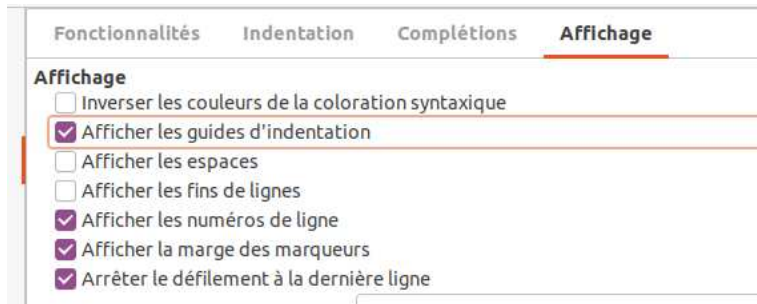
- tabulation
- complétion automatique
- la fermeture automatique des ( { " et '
- l'indentation

# Indentation



## Fermeture automatique des guillemets, accolades et crochets

- ✓ Parenthèses ()
- ✓ Accolades {}
- ✓ Crochets []
- ✓ Guillemets simples ''
- ✓ Guillemets doubles ""



- 1 Paramétrer son éditeur
- 2 Style du code**
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

Écrire un code propre, facile à lire, respectant les conventions assure une meilleure lisibilité des programmes :

- partager son code
- faciliter sa compréhension
- faciliter la correction des erreurs et sa mise au point.

```
NB=SimpleInput.getInt("nb = ");  
if(NB>10) NB-- else {  
    NB+=1;}  
System.out.print(NB);
```



# Indentation

- Utiliser une indentation de 4 espaces, sans tabulation
- Bien placer les indentations

```
// Mauvaise pratique
while (p != q) {
    if (p > q) {
        p = p - q;
    } else {
        q = q - p;
    }
}
```

```
// Bonne pratique
while (p != q) {
    if (p > q) {
        p = p - q;
    } else {
        q = q - p;
    }
}
```

```
class PGCD {  
    void principal() {  
        int val1;  
        int val2;  
  
        val1 = SimpleInput.getInt ("Première valeur : ");  
        val2 = SimpleInput.getInt ("Deuxième valeur : ");  
  
        while (val1 != val2) {  
            if (val1 > val2) {  
                val1 = val1 - val2;  
            } else {  
                val2 = val2 - val1;  
            }  
        }  
        System.out.println("Le résultat est : " + val1);  
    }  
}
```

# Les accolades des blocs de codes

Il est préférable de toujours mettre les accolades autour des structures de contrôle `if`, `while`, `for`, `else`...

```
// Mauvaise pratique
```

```
if (i > 0)
    System.out.println (i);
```

```
// Bonne pratique
```

```
if (i > 0) {
    System.out.println (i);
}
```

# Éviter les écritures complexes

Certaines écritures peuvent paraître plus courtes à écrire. Elles sont à éviter car elles compliquent la lecture et la compréhension du code et peuvent créer des ambiguïtés.

**En aucun cas elles n'accélèrent l'exécution du programme !**

# Éviter les opérateurs ternaires

```
// Mauvaise pratique
result = (isEnabled ())
        ? getValidCode ( )
        : ( isBusy ( ) )
          ? getBusyCode ( )
          : getErrorCode ( );
```

```
// Bonne pratique
if (isEnabled ()) {
    result = getValidCode ();
} else if (isBusy ()) {
    result = getBusyCode ( ) ;
} else {
    result = getErrorCode ( );
}
```

# Éviter les affectations dans les structures de contrôle

```
// Mauvaise pratique
if ((a = b) == 0) {
    // suite du code
}
```

```
// Bonne pratique
a = b;
if (a == 0) {
    // suite du code
}
```

# Éviter les affectations multiples

```
int i = 1;  
int j = 2;  
int k = 3;
```

```
// Mauvaise pratique  
i = j = k;
```

```
// Bonne pratique  
i = k;  
j = k;
```

```
// Mauvaise pratique  
valeur = (i = j + k ) + m;
```

```
// Bonne Pratique  
i = j + k;  
valeur = i + m;
```



# Déclaration et initialisation des variables

- Il est fortement recommandé d'initialiser les variables au moment de leur déclaration.
- Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc. (un bloc est un morceau de code entouré par des accolades).

```
// Mauvaise pratique
{
    System.out.println ("Bonjour");
    int i;
    i = 10;
    System.out.println ("i = " + i);
    int j;
    i = i + 1;
    j = 0;
    System.out.println ("j = " + j);
}
```

```
// Bonne pratique
{
    int i = 10;
    int j = 0;

    System.out.println ("Bonjour");
    System.out.println ("i = " + i);
    i = i + 1;
    System.out.println ("j = " + j);
}
```

# Déclarations multiples

Proscrire la déclaration d'une variable qui masque une variable définie dans un bloc parent afin de ne pas complexifier inutilement le code.

```
// Mauvaise pratique
int taille;
...
void maMethode() {
    int taille;
}
```

# Éviter les conditionnelles négatives

Les expressions négatives sont plus difficiles à comprendre que les expressions positives. Les expressions devraient être exprimées de manière positive.

```
// Mauvaise pratique  
if (!shouldNotCompact ())
```

```
// Bonne pratique  
if (shouldNotCompact ())
```

# Remplacer les nombres et les chaînes par des constantes

Éviter de conserver les nombres et les chaînes de caractères en brut dans le code. La bonne pratique est d'utiliser des constantes nommées. Par exemple, le nombre 3 600 000 devrait être affecté à la constante `MILLIS_PER_HOUR`.

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :

- avant la déclaration d'une méthode,
- entre les déclarations des variables locales et la première ligne de code,
- avant un commentaire d'une seule ligne,
- avant chaque section logique dans le code d'une méthode.

# Utiliser les conventions de nommage du langage

```
void maMethode (int paramUn, int paramDeux) {  
  
    int maVariable;  
  
    final int MA_CONSTANTE = ...;  
  
    class MaClasse {...}
```

- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation**
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

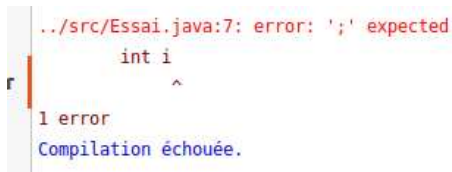


Savoir lire et comprendre les messages d'erreur de la compilation permet de gagner en vitesse. Il faut notamment :

- déterminer la ligne où se trouve l'erreur
- comprendre le message du compilateur

*Un compilateur ne se trompe jamais !*

# Analyser le message d'erreur

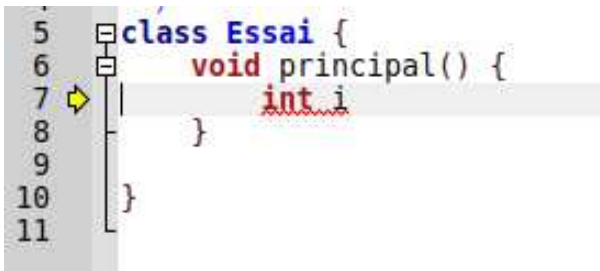


```
../src/Essai.java:7: error: ';' expected
    int i
      ^
1 error
Compilation échouée.
```

- `../src/Essai.java:7` indique le nom du fichier `Essai.java` et la ligne 7,
- `error: ';' expected` indique l'erreur détectée par le compilateur,
- `^` indique l'endroit où le compilateur a détecté l'erreur,
- `1 error` indique le nombre d'erreurs détectées par le compilateur.

Les bons éditeurs indiquent où se trouvent les erreurs :

```
5  class Essai {  
6      void principal() {  
7          int i  
8      }  
9  }  
10  
11
```

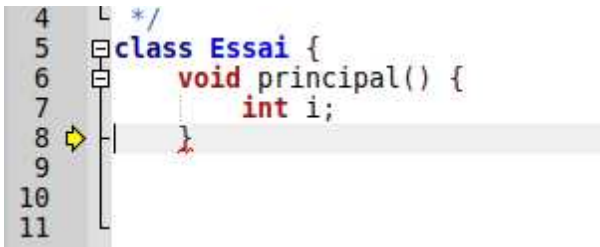


Il n'est pas question de lister toutes les erreurs rencontrées lors de la compilation d'un programme java. Seules sont présentées celles qui sont les plus difficiles à résoudre.

# Error : reached end of file while parsing

```
../src/Essai.java:8: error: reached end of file while parsing
    }
    ^
```

Il manque des accolades fermantes { } .



The screenshot shows a code editor with the following code:

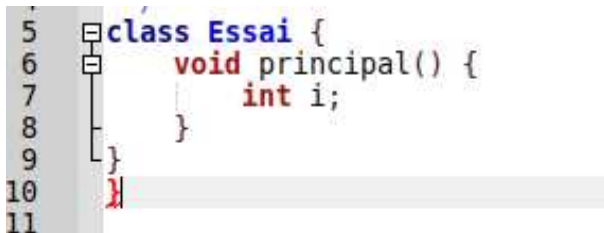
```
4  */
5  class Essai {
6      void principal() {
7          int i;
8      }
9
10
11
```

A yellow arrow points to the closing brace on line 8, indicating the end of the file. The code is missing the closing brace for the class.

# Error : class, interface, or enum expected

```
../src/Essai.java:10: error: class, interface, or enum expected
}
^
```

Des accolades fermantes, }, sont en trop. Le compilateur est perdu et ne comprend pas la présence de l'accolade fermante en ligne 10.



```
5  class Essai {
6      void principal() {
7          int i;
8      }
9  }
10 }
11
```

# Error : variable i might not have been initialized

```
../src/Essai.java:12: error: variable i might not have been initialized
    System.out.println("i = " + i);
                               ^
```

La variable `i` ne peut pas être utilisée car il est possible qu'elle ne contienne pas de valeur, par exemple si l'utilisateur rentre une valeur négative pour `v`.

```
5  class Essai {
6      void principal() {
7          int i;
8          int v = SimpleInput.getInt("v = ");
9          if (v > 0) {
10             i = v;
11         }
12         System.out.println("i = " + i);
13     }
14 }
15
```

- Écrire un programme par morceau en le compilant rapidement et ne pas hésiter si nécessaire, à commenter des lignes en cours d'écriture
- Faire en sorte que la première version du programme contienne un minimum d'erreurs
- Corriger le maximum d'erreurs à chaque compilation



# Sommaire

- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution**
- 5 Mise au point
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

Les erreurs d'exécution sont les plus difficiles à corriger. La meilleure façon de les corriger est d'écrire des programmes corrects.

# Analyser le message d'erreur

```
$ java -cp ../class Start "Essai"  
v = 0  
java.lang.reflect.InvocationTargetException  
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0  
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(  
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.in  
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)  
    at Start.main(Start.java:16)  
Caused by: java.lang.ArithmeticException: / by zero  
    at Essai.principal(Essai.java:8)  
    ... 5 more
```

- Caused by: java.lang.ArithmeticException: / by zero la cause de l'erreur
- at Essai.principal(Essai.java:8) la ligne de l'erreur

# Les principaux messages d'erreurs

Les messages d'erreurs à l'exécution sont rares, mais ils sont une chance. Il est plus facile de corriger un programme qui s'arrête sur un message d'erreurs que de corriger un programme qui ne produit pas le résultat attendu ou qui ne se s'arrête pas.

# Index 11 out of bounds for length 10

```
$ java -cp ../class Start "Essai"
java.lang.reflect.InvocationTargetException
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.in
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at Start.main(Start.java:16)
Caused by: java.lang.ArrayIndexOutOfBoundsException: Index 11 out of bound
    at Essai.principal(Essai.java:8)
    ... 5 more
```

- La valeur de l'indice d'un tableau est en dehors des bornes, strictement inférieure à 0 ou supérieure ou égale à `.length`
- La valeur de l'indice est 11 et la longueur du tableau est 10
- L'erreur est sur la ligne 8

```
class Essai {  
    void principal() {  
        int[] t = new int[10];  
        System.out.println(t[11]);  
    }  
}
```

# java.lang.ClassNotFoundException

```
$ java -cp ../class Start "Essai"  
Erreur : impossible de trouver ou de charger la classe principale S  
Causé par : java.lang.ClassNotFoundException: Start
```

Le programme a besoin de la class start pour s'exécuter, mais elle n'est pas présente dans le répertoire ../class.

*Attention cette erreur est "spécifique" à la mise en œuvre de java dans la ressource R1.01.*

# java.lang.NullPointerException

```
$ java -cp ../class Start "Essai"
java.lang.reflect.InvocationTargetException
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.in
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at Start.main(Start.java:16)
Caused by: java.lang.NullPointerException
    at Essai.principal(Essai.java:8)
    ... 5 more
```

- La référence au tableau `t` est `null`
- La valeur `.length` n'est donc pas disponible
- L'erreur est sur la ligne 8



```
class Essai {  
    void principal() {  
        int[] t = null;  
        int l = t.length;  
    }  
}
```

# java.lang.StackOverflowError

```
$ java -cp ../class Start "Essai" 2> err.txt
$ head err.txt
java.lang.reflect.InvocationTargetException
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at Start.main(Start.java:16)
Caused by: java.lang.StackOverflowError
    at Essai.rec(Essai.java:11)
    at Essai.rec(Essai.java:11)
    at Essai.rec(Essai.java:11)
```

- Un appel récursif entraîne un dépassement de la capacité de la pile (stack))
- Noter la redirection de la sortie d'erreurs 2>
- L'erreur est sur la ligne 11

```
class Essai {  
    void principal() {  
        rec();  
    }  
  
    void rec() {  
        rec();  
    }  
}
```

Les exemples de cette partie sont volontairement simples. Les erreurs sont généralement moins évidentes à déterminer.

- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point**
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

- La mise au point de programme est difficile.
- Plusieurs techniques sont possibles :
  - l'exécution à la main
  - la visualisation de l'exécution avec par exemple javaTutor
  - l'ajout de l'affichage des variables
  - L'utilisation d'un débogueur (debugger) qui n'est pas abordé dans ce cours
  - Commenter les grandes étapes du programme long pour faciliter la lecture et la relecture

La meilleure manière de faire est d'écrire directement un programme qui fonctionne. L'écriture d'un programme par essais/erreurs est souvent plus longue, plus hasardeuse et produit des programmes difficile à lire et à comprendre.

# Exécution à la main

```
class Essai {  
    void principal() {  
        int p = SimpleInput.getInt ("p = ");  
        int q = SimpleInput.getInt ("q = ");  
        while (p != q){  
            if (p > q) {  
                p = p - q;  
            } else {  
                q = p - q;  
            }  
        }  
    }  
}
```

```
$ java -cp ../class Start "Essai"  
p = 12  
q = 10  
^C
```

	p	q
	12	10
if ( $p > q$ )		
$p = p - q$	2	10
while ( $p \neq q$ )	2	10
if ( $p > q$ )	2	10
$q = p - q$	2	-8
while ( $p \neq q$ )	2	-8
if ( $p > q$ )	2	-8
$p = p - q$	10	-8



Java 8  
([known limitations](#))

```
1 public class YourClassNameHere {  
2     public static void main(String[] args) {  
3         int p = 12;  
4         int q = 10;  
5         while (p != q){  
6             if (p > q) {  
7                 p = p - q;  
8             } else {  
9                 q = p - q;  
10            }  
11        }  
12    }  
13 }
```

[Edit this code](#)

→ line that just executed

→ next line to execute



<< First

< Prev

Next >

Last >>

Frames

Objects

main:5

p 12

q 10

## Java Tutor

*Évidemment, il ne faut pas qu'il y ait de saisies dans le corps de boucle.*

# Ajout de traces dans le programme

```
class Essai {  
    void principal() {  
        int v;  
        int nb = 0;  
        int somme = 0;  
  
        v = SimpleInput.getInt ("v = ");  
        while (v != -1){  
            somme = somme + v;  
            nb = nb + 1;  
            v = SimpleInput.getInt ("v = ");  
        }  
        if (nb != 0){  
            float moyenne = somme / nb;  
            System.out.println ("Moyenne = " + moyenne);  
        }  
    }  
}
```

```
$ java -cp ../class Start "Essai"  
v = 10  
v = 5  
v = -1  
Moyenne = 7.0
```

```
class Essai {  
    void principal() {  
        int v;  
        int nb = 0;  
        int somme = 0;  
  
        v = SimpleInput.getInt ("v = ");  
        while (v != -1){  
            somme = somme + v;  
            nb = nb + 1;  
            v = SimpleInput.getInt ("v = ");  
        }  
        System.out.println ("somme = " + somme);  
        System.out.println ("nb = " + nb);  
        if (nb != 0){  
            float moyenne = somme / nb;  
            System.out.println ("Moyenne = " + moyenne);  
        }  
    }  
}
```

```
$ java -cp ../class Start "Essai"  
v = 10  
v = 5  
v = -1  
somme = 15  
nb = 2  
Moyenne = 7.0
```

```
class Essai {  
    final boolean DEBUG = true;  
    void principal() {  
        ..  
        if (DEBUG) {  
            System.out.println ("somme = " + somme);  
            System.out.println ("nb = " + nb);  
        }  
        ...  
    }  
}
```

```
$ java -cp ../class Start "Essai"  
v = 10  
v = 5  
v = -1  
somme = 15  
nb = 2  
Moyenne = 7.0
```

Dans le cadre d'un programme plus long :

- Déterminer la partie de code fautive
- Isoler la partie fautive pour la coder de manière unitaire
- Le débogueur est le meilleur outil pour mettre au point un programme

# Sommaire

- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point
- 6 Les tests**
- 7 Conclusion sur la mise au point de programme

- Tester un programme est une tâche difficile
- Tester un programme c'est essayer le maximum de cas possibles
- Tester un programme ce n'est pas choisir des valeurs au hasard
- Tester un programme c'est connaître le résultat attendu
- Tester un programme ne prouve pas sa correction



# Exemple

```
class Essai {
    final boolean DEBUG = false;
    void principal() {
        int v;
        int nb = 0;
        int somme = 0;

        v = SimpleInput.getInt ("v = ");
        while (v != -1){
            somme = somme + v;
            nb = nb + 1;
            v = SimpleInput.getInt ("v = ");
        }
        if (DEBUG) {
            System.out.println ("somme = " + somme);
            System.out.println ("nb = " + nb);
        }
        if (nb != 0){
            float moyenne = somme / nb;
            System.out.println ("Moyenne = " + moyenne);
        }
    }
}
```

# Un cas normal

```
$ java -cp ../class Start "Essai"  
v = 11  
v = 5  
v = 15  
v = 3  
v = 12  
v = -1  
Moyenne = 9.0
```

Le résultat est difficile à calculer !!!

# Un autre cas normal

```
$ java -cp ../class Start "Essai"  
v = 10  
v = 5  
v = 15  
v = -1  
Moyenne = 10.0
```

- Le résultat attendu est connu avant même la saisie des valeurs
- Le nombre de valeurs est impair

# Un troisième cas normal

```
$ java -cp ../class Start "Essai"  
v = 12  
v = 14  
v = 8  
v = 6  
v = -1  
Moyenne = 10.0
```

- Le résultat attendu est connu avant même la saisie des valeurs
- Le nombre de valeurs est pair
- La moyenne est une valeur entière

# Un quatrième cas normal

```
$ java -cp ../class Start "Essai"  
v = 10  
v = 5  
v = -1  
Moyenne = 7.0
```

- Le résultat attendu est connu avant même la saisie des valeurs
- Le résultat n'est pas correct

# Découper son code

Découper son programme en méthodes de taille moyenne qui peuvent être testées unitairement.

- 1 Paramétrer son éditeur
- 2 Style du code
- 3 Les messages d'erreurs à la compilation
- 4 Les messages d'erreurs à l'exécution
- 5 Mise au point
- 6 Les tests
- 7 Conclusion sur la mise au point de programme

- Écrire un programme n'est pas une tâche facile
- Respecter les règles d'écriture, écrire un code simple, lisible facilite la mise au point de programme
- Bien écrire le programme dès le début en rejetant la tentation de "Je le ferai plus tard !"
- Le modèle essai-erreur est à éviter
- Réfléchir avant de coder, produire un code exempt d'erreurs de compilation est la meilleure manière d'écrire un programme correct
- Décomposer le programme en méthodes moins compliquées à coder.
- Tester un programme n'est pas une tâche à prendre à la légère