HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Professeur : Nastaran Fatemi
Assitant : Jonathan Bischof

Installation, exemples et exercices

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Professeur : Nastaran Fatemi
Assitant : Jonathan Bischof

1 Scala IDE

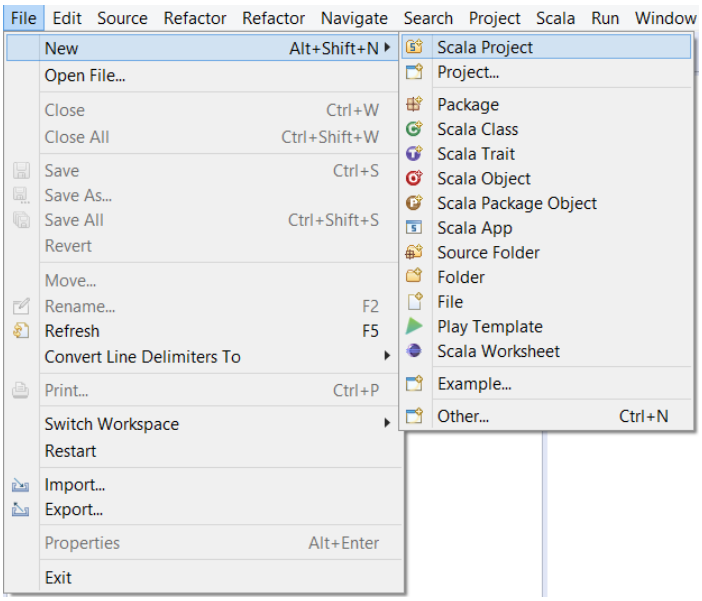
1.1 Installation de l'IDE

Téléchargez Scala IDE for Eclipse, qui contient tout ce qui est nécessaire pour développer en Scala. Cette version d'Eclipse possède un interpréteur de Scala (*Worksheet*), permettant de faciliter le développement.

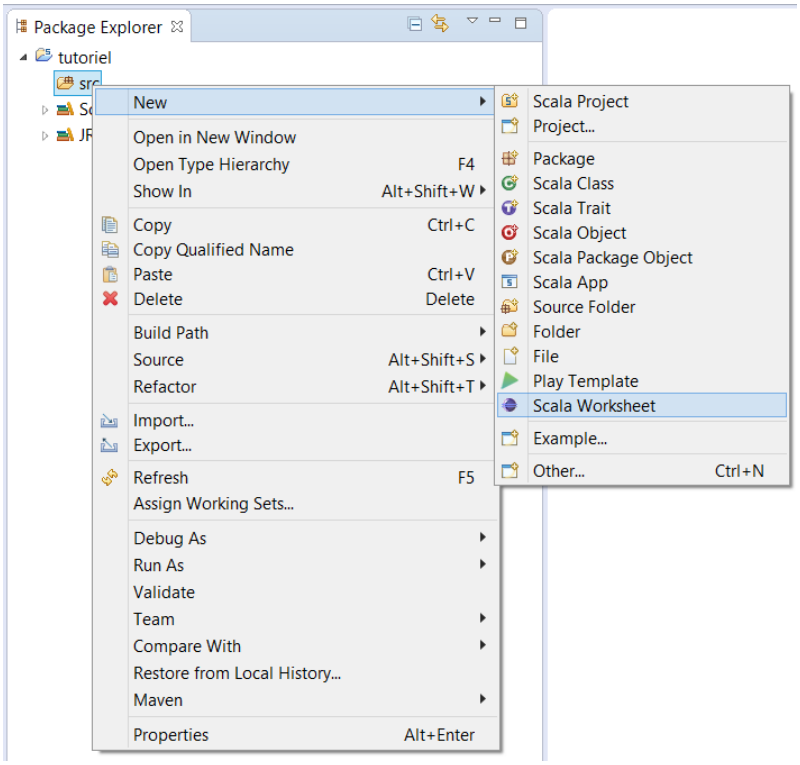
Que ce soit pour **Windows**, **Mac OS X** ou **Linux**, la procédure d'installation est identique. Décompressez l'archive précédemment téléchargée, démarrez Eclipse via l'exécutable correspondant et configurez votre espace de travail (*workspace*).

1.2 Utilisation de la *Worksheet*

Commencez par créer un projet nommé **tutoriel** :

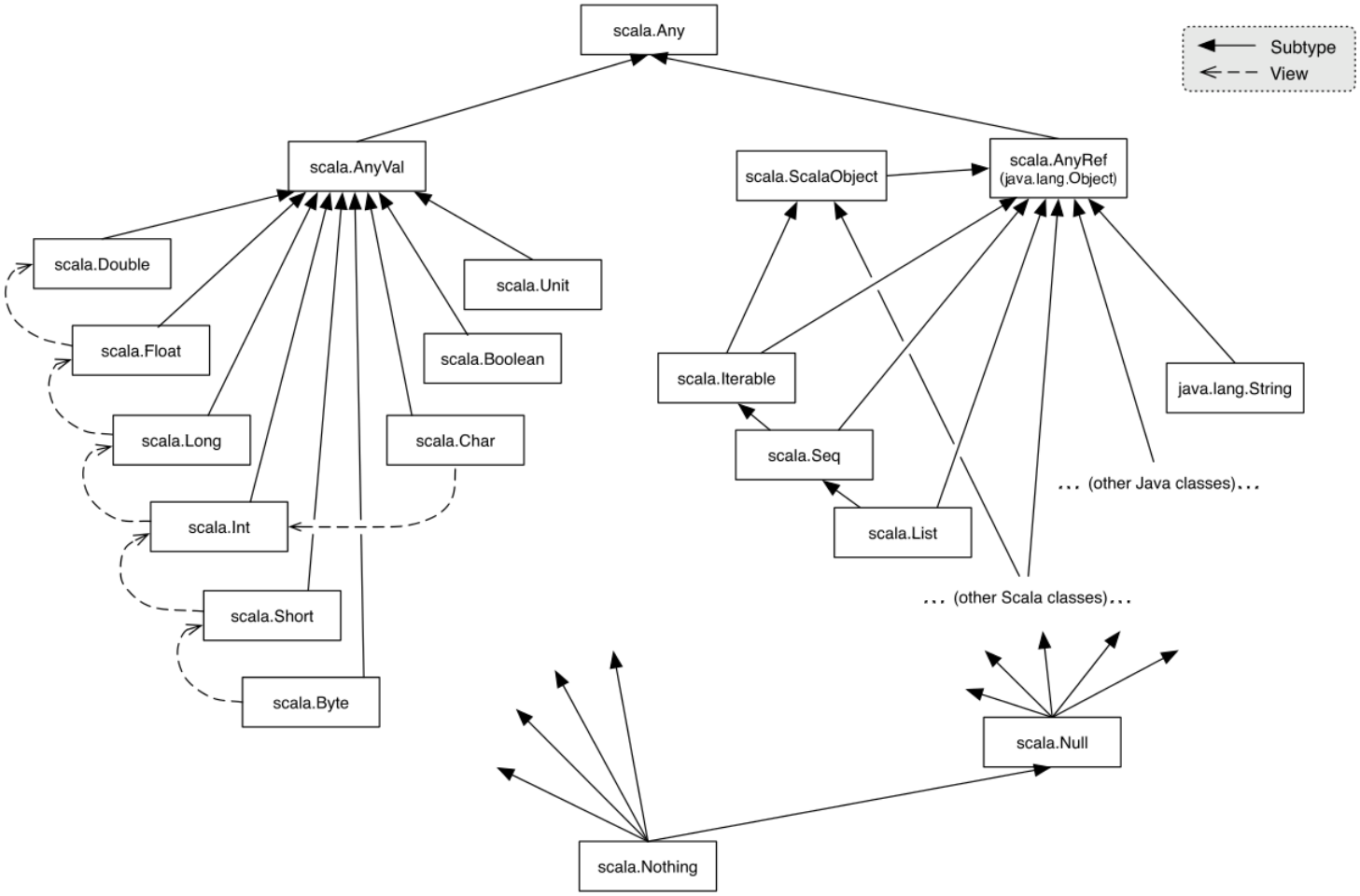


Puis faites un clique droit sur le dossier **src**, et créez une *Worksheet* nommée **tutoriel** :



2 Syntaxe du langage

2.1 Types



Comme vous pouvez le voir sur l'image ci-dessus, il existe les types habituels (**Double**, **Float**, ...), ainsi que des types propres à Scala. Ils seront présentés plus en détails pendant le cours, et ce tutoriel n'utilisera que des types simples comme **Int** ou **String**.

2.2 Variables

Il existe deux catégories de variables :

— Les *immuables*, déclarées avec le mot-clé **val**.

— Les *mutables*, déclarées avec le mot-clé **var**.

En règle générale, on préfère utiliser les variables immuables, bien que les mutables existent aussi. Il n'est pas obligatoire de déclarer explicitement le type d'une variable, l'interpréteur peut s'en occuper.

```
var foo = 0.5                          //> foo  : Double = 0.5
var bar: Int = 5                       //> bar  : Int = 5
val foo2 = 'c'                         //> foo2  : Char = c
var bar2: String = "Hello World !"     //> bar2  : String = Hello World !
```

La valeur d'une variable mutable peut être modifiée, tandis que celle d'une variable immuable non, c'est en quelque sorte, une constante.

2.3 Raccourcis du langage

Il n'est pas nécessaire d'utiliser le **;**, vous pouvez utiliser le retour à la ligne comme délimiteur d'instruction. Il n'est d'ailleurs pas toujours nécessaire d'utiliser le **.** ou les **( )** pour certaines instructions.

```
bar2 == "Hello World !"                  //> res0: Boolean = true
bar2.equals("Hello World !")             //> res1: Boolean = true
bar2 equals "Hello World !"              //> res2: Boolean = true
```

Mais attention, cela peut vite devenir très difficile debugger !

2.4 Conditions booléennes

L'instruction **if** ... **else if** ... **else** ... est implémentée de manière similaire à Java.

```
if (foo > 1.0) {
  println("Hello World !")
} else {
  println("Goodbye !")
}                                        //> Goodbye !
```

2.5 Boucles

2.5.1 do ... while

La boucle **do** ... **while** est implémentée de manière similaire à Java.

```
do {
  println("Hello World !")
  foo += 0.1
} while (foo < 1.0)                      //> Hello World !
```

2.5.2 while

La boucle **while** est implémentée de manière similaire à Java.

```
while (foo > 0.5) {
  println("Hello world !")
  foo -= 0.1
}                                        //> Hello world !
```

2.5.3 for

Pour écrire une boucle **for** *conventionnelle*, il est nécessaire de connaître les mots-clés **to** (inclusif) et **until** (exclusif).

```
for (foo <- 0 to 2) {
  println(foo)                           //> 0
                                         //| 1
                                         //| 2
}
for (foo <- 0 until 2) {
  println(foo)                           //> 0
                                         //| 1
}
```

Il est également possible d'ajouter des conditions booléennes directement dans les boucles **for** :

```
for (foo <- 0 to 5 if foo % 2 == 0) {
  println(foo)                                    //> 0
                                                  //| 2
                                                  //| 4
}
```

Ou d'imbriquer directement 2 (ou plus) boucles **for** :

```
for (foo <- 0 to 2; bar <- 5 to 7) {
   println(foo + " : " + bar)                     //> 0 : 5
                                                  //| 0 : 6
                                                  //| 0 : 7
                                                  //| 1 : 5
                                                  //| 1 : 6
                                                  //| 1 : 7
                                                  //| 2 : 5
                                                  //| 2 : 6
                                                  //| 2 : 7
}
```

2.6 Match cases

Il existe également une instruction équivalente au **switch** de Java :

```
val foo = 10                                      //> foo  : Int = 10
foo match {
  case 10 => true
  case _ => false
}                                                 //> res0: Boolean = true
```

L'instruction _ est un *joker* du langage, qui ici, signifie le cas par défaut (ou *dans tous les cas*). Il est également possible d'ajouter des conditions booléennes directement dans les **case** :

```
val foo = 10                                      //> foo  : Int = 10
foo match {
  case f if f % 2 == 0 => false
  case g if g % 10 == 0 => true
  case 10 => true
  case _ => false
}                                                 //> res0: Boolean = false
```

Ici, **f** et **g** sont des alias pour la condition, dans ce cas-ci, ils sont équivalent à _, et on peut les nommer selon les mêmes règles que pour le nommage d'une variable (foo, bar_2, ...). Ceci permet d'utiliser la valeur de cette condition, dans une condition booléenne. Attention cependant à l'ordre dans lequel les conditions sont vérifiées.

2.7 Fonctions

Une fonction qui ne retourne rien est de type **Unit**, et elle peut être déclarée avec ou sans le signe =

```
def foo() {
  println("bar")
}                                          //> foo: ()Unit

def foo() = {
  println("bar")
}                                          //> foo: ()Unit
```

Elle peut également avoir des paramètres :

```
def foo(bar: Int) {
  println(bar)
}                                          //> foo: (bar: Int)Unit
def foo(bar: Int) = {
  println(bar)
}                                          //> foo: (bar: Int)Unit
```

Une fonction qui retourne une valeur **doit** être déclarée avec le signe =

```
def foo(bar: Int): Int = {
  return bar
}                                          //> foo: (bar: Int)Int
```

Ici, l'instruction **: Int** signifie que la fonction retournera une valeur de type **Int**. Le mot-clé **return** n'est toutefois pas obligatoire :

```
def foo(bar: Int): Int = {
  bar
}                                          //> foo: (bar: Int)Int
```

Et on peut encore simplifier l'écriture :

```
def foo(bar: Int): Int = bar               //> foo: (bar: Int)Int
```

L'appel de fonction est tout ce qu'il y a de plus standard :

```
foo(5)                                     //> res0: Int = 5
```

Il est également possible de créer une fonction à l'intérieur d'une autre fonction, et de l'utiliser comme valeur de retour :

```
def foo(x: Int): Int = {
  def bar(y: Int): Int = x*y
  bar(x + 1)
}                                          //> foo: (x: Int)Int
foo(5)                                     //> res0: Int = 30
```

Ce type de fonctions sera développé plus en détails dans le cours sur la récursion. Bien que le type de retour d'une fonction ne soit pas obligatoire, en dehors de fonctions récursives, il est fortement conseillé de toujours l'écrire.

Il est également possible de combiner de manière simple une **fonction** et un **match cases** :

```
def foo(bar: String) = bar match {
  case l if l.length() == 5 => "La longueur est 5"
  case "Hello World" => "Goodbye !"
  case _ => "Unknown :("
}                                          //> foo: (bar: String)String

foo("Hello")                               //> res0: String = La longueur est 5
foo("Hello World")                         //> res1: String = Goodbye !
foo("bar")                                 //> res2: String = Unknown :(
```

Le langage est en partie similaire à Java, il est d'ailleurs compilé en bytecode et on peut implémenter directement les classes Java dans un projet Scala.

3 Exercices

Les exercices de ce tutoriel sont classés en 2 catégories :

— Niveau 1 : Facile

— Niveau 2 : Intermédiaire

Ces exercices ne sont pas notés !

3.1 Exercices : Niveau 1

3.1.1 Déclaration de variables

Déclarez les variables mutables `nom` et `prenom` de type `String`, avec la valeur vide comme valeur par défaut.

3.1.2 Affectation de variables

Affectez votre nom et votre prénom aux deux variables déclarées dans l'exercice précédent.

3.1.3 Immuables vs mutables

Déclarez la variable immuable `dateDeNaissance` de type `String`, avec la valeur vide comme valeur par défaut.

— Affectez votre date de naissance à la variable `dateDeNaissance`.

— Est-ce possible ? Pourquoi ?

— Si ça ne fonctionne pas, faites en sorte que la variable `dateDeNaissance` possède comme valeur, votre date de naissance.

3.1.4 Affichage

Affichez, à l'aide de la fonction `println(...)` et des variables `nom`, `prenom` et `dateDeNaissance` le texte :

```
Bonjour, je m'appelle «votre_prenom» «votre_nom» et je suis né le «votre_dateDeNaissance».
```

3.1.5 Boucle I

Déclarez la variable immuable `compteur` avec `0` comme valeur par défaut. Utilisez une boucle, avec la condition d'arrêt `(compteur != 0)`, qui affiche *au moins une fois* le message de l'exercice précédent.

— Quelle boucle avez-vous utilisé ?

— Pourquoi celle-ci et pas une autre ?

3.1.6 Boucle II

En utilisant une boucle `while` et la variable `compteur`, affichez 5 fois le message de l'exercice `3.1.4`.

3.1.7 Boucle III

En utilisant une boucle `for`, affichez 5 fois le message de l'exercice `3.1.4`.

3.1.8 Boucle IV

En utilisant une boucle `for`, et une **autre condition d'arrêt** qu'à l'exercice précédent, affichez 5 fois le message de l'exercice `3.1.4`.

3.1.9 Condition booléenne

En utilisant les variables `compteur`, `nom` et `prénom`, utilisez une condition booléenne pour afficher votre `nom` si le `compteur` est égal à `0`, sinon votre `prénom`.

3.1.10 Match cases

Réimplémentez l'exercice précédent en remplaçant les conditions booléennes par un `match cases` **Scala**.

3.2 Exercices : Niveau 2

3.2.1 Boucle I

Implémentez une boucle `while` qui n'affiche que les nombres impairs et multiple de 3 ou de 5, compris entre 1 et 100.
Hint : 3, 5, 9, 15, 21, 25, 27, 33, 35, 39, 45, 51, 55, 57, 63, 65, 69, 75, 81, 85, 87, 93, 95, 99

Implémentez la même condition d'affichage, en utilisant une boucle `for` et une seule ligne de code.
Hint : vous pouvez écrire for (...) println(...) sur la même ligne.

3.2.2 Fonction I

Implémentez une fonction, prenant deux `Int` en paramètre, qui affiche le plus grand des deux paramètres, en une seule ligne de code.
Hint : func(3, 5) affiche 5.

3.2.3 Fonction II

Implémentez une fonction qui prend un `String` en paramètre, si ce paramètre vaut

- `"Hello"`, la fonction affiche `"World"`.
- `"World"`, la fonction affiche `"Hello"`.
- Sinon, la fonction affiche `"Goodbye"`.

Sans utiliser de conditions booléennes.
Hint : match cases.

3.2.4 Boucle II

Implémentez la boucle si dessous, en **une seule boucle `for` Scala**.

```
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j <= 3; ++j) {
        for (int k = 0; k < 4; ++k) {
            System.out.println("Hello World !");
        }
    }
}
```

- Combien de fois le message `"Hello World !"` est-il affiché ?
- Comment le compter facilement ?

3.2.5 Match cases I

Implémentez un `match cases` sur un `String`, qui vérifie dans l'ordre :

- Si la longueur du `String` est impaire, affiche la longueur.
- Si le `String` est égal à `"Hello World"`, l'affiche.
- Sinon, affiche `"Goodbye"`.

3.2.6 Boucle III & Match cases II

A l'aide d'une variable mutable `compteur` valant `0`, implémentez une boucle `while` ayant pour condition, tant que le `compteur` est plus petit que `10`. Sans utiliser de conditions booléennes, incrémentez le `compteur` comme suit :

- Si le `compteur` est pair, on l'incrémente de `3`.
- Si le `compteur` est impair, on l'incrémente de `1`.

Combien de fois la boucle est-elle exécutée ? Quelle est la valeur du `compteur` après ses exécutions ?

3.2.7 Fonctions III & Match cases III

Implémentez une fonction qui prend un `Int` en paramètre.

- Si `x` est plus grand que `100`, retourne la valeur de `x`.
- Si `x` est un multiple de `7`, appelle la fonction avec `x + 8`.
- Si `x` est impair, appelle la fonction avec `x + 12`.
- Sinon, appelle la fonction avec `x + 1`.

Appelez cette fonction avec la variable immuable `x` ayant comme valeur `0`.

- Quelle est la valeur finale retournée par la fonction ?
- Est-ce la nouvelle valeur de `x` ?
- Combien de fois la fonction est-elle exécutée ?
- Pourquoi pouvons-nous appeler la fonction à l'intérieur de celle-ci sans explicitement retourner une valeur de type `Int` ?

3.2.8 Fonction IV

Implémentez une fonction qui prends trois `Int` (`x`, `y`, `z`) en paramètre.

- Si `z` est pair, retournez la somme des carrés de `x` et `y`.
- Si `z` est impair, retourner le carré de la somme de `x` et de `y`.
- Si `z` vaut 0, retournez la somme des cas `z` pair et `z` impair.

Appelez la fonction avec `x = 2`, `y = 3`, et `z = 1`, puis `2`, puis `0`.

- Est-ce que le cas `z = 0` est correct ? Pourquoi ?
- S'il n'est pas correct, modifiez le code pour obtenir le bon résultat.

3.2.9 Fonction V

Implémentez deux versions d'une fonction qui prends deux `Int` (`x` et `y`) en paramètre, une fois avec des conditions booléennes et une fois avec des `match cases`.

- Si `x == y`, retourne `x`.
- Si `x < y`, retourne `y`.
- Si `x > y`, alors :
  - Si `x` est impair, retourne `2x + 3y`.
  - Si `y` est impair, retourne `4x - 7y`.
  - Si `x` est un multiple de `3` et `y` un multiple de 4, retourne $x^2 + y^3$
- Si `x < y` et `x > 4`, retourne `2x`.
- Si `x == y` et `y == 0`, retourne `42`.

Faites en sorte que **toutes** les conditions soient vérifiables.
Hint : il faut faire attention à l'ordre.

3.2.10 Fonction VI

Implémentez une fonction que retourne la somme des `n` premiers nombres de Fibonacci.
Appelez cette fonction avec `n = 20`.