Récapitulatif syntaxique du langage Scala

Variables	
`var x = 5`	variable
Good `val x = 5` **Bad** `x=6`	constante
`var x: Double = 5`	type explicite

Fonctions	
Good `def f(x: Int) = { x*x }` **Bad** `def f(x: Int) { x*x }`	définition d'une fonction erreur cachée : sans le = c'est une procédure qui retourne un Unit ; occasionnant des problèmes incontrôlés.
Good `def f(x: Any) = println(x)` **Bad** `def f(x) = println(x)`	définition d'une fonction erreur de syntaxe : chaque argument à besoin d'être typés.
`type R = Double`	alias de type
`def f(x: R)` vs. `def f(x: => R)`	appel par valeur appel par nom (paramètres paresseux (lazy))
`(x:R) => x*x`	fonction anonyme
`(1 to 5).map(_*2)` vs. `(1 to 5).reduceLeft( _+_ )`	fonction anonyme : l'underscore est associé à la position du paramètre en argument.
`(1 to 5).map( x => x*x )`	fonction anonyme : pour utiliser un argument deux fois, il faut le nommer.
Good `(1 to 5).map(2*)` **Bad** `(1 to 5).map(*2)`	fonction anonyme : méthode bornée et infixée. Il faut utiliser `2*_` .
`(1 to 5).map { val x=_*2; println(x); x }`	fonction anonyme : la dernière expression d'un bloc est celle qui est retournée.
`(1 to 5) filter {_%2 == 0} map {_*2}`	fonctions anonymes : style "pipeline". (ou avec des parenthèses).
`def compose(g:R=>R, h:R=>R) = (x:R) => g(h(x))` `val f = compose({_*2}, {_-1})`	fonctions anonymes : pour passer plusieurs blocs, il faut les entourer par des parenthèses.
`val zscore = (mean:R, sd:R) => (x:R) => (x-mean)/sd`	curryfication, syntaxe évidente.
`def zscore(mean:R, sd:R) = (x:R) => (x-mean)/sd`	curryfication, syntaxe évidente.

`def zscore(mean:R, sd:R)(x:R) = (x-mean)/sd`
 	
`val normer = zscore(7, 0.4)_`
	curryfication, sucre syntaxique. mais alors :
	il faut traiter l'underscore dans la fonction partielle, mais ceci uniquement pour la version avec le sucre syntaxique.

`def mapmake[T](g:T=>T)(seq: List[T]) = seq.map(g)`	type générique.
`5.+(3); 5 + 3` `(1 to 5) map (_*2)`	sucre syntaxique pour opérateurs infixés.
`def sum(args: Int*) = args.reduceLeft(_+_)`	arguments variadiques.

Paquetages	
`import scala.collection._`	import global.
`import scala.collection.Vector` `import scala.collection.{Vector, Sequence}`	import sélectif.
`import scala.collection.{Vector => Vec28}`	renommage d'import.
`import java.util.{Date => _, _}`	importe tout de java.util excepté Date.
`package pkg` *en début de fichier* `package pkg { ... }`	déclare un paquetage.

Pattern matching	
Good `(xs zip ys) map { case (x,y) => x*y }` **Bad** `(xs zip ys) map( (x,y) => x*y )`	cas d'utilisation d'une fonction utilisée avec un "pattern matching".

Bad
`val v42 = 42`
`Some(3) match {`
`case Some(v42) => println("42")`
`case _ => println("Not 42")`
`}`
	"v42" est interprété comme un nom ayant n'importe quelle valeur de type Int, donc "42" est affiché.

Good
`val v42 = 42`
`Some(3) match {`
`case Some(`v42`) => println("42")`
`case _ => println("Not 42")`
`}`
	"`v42`" x les "backticks" est interprété avec la valeur de val `v42`, et "Not 42" est affiché.

Good
`val UppercaseVal = 42`
`Some(3) match {`
`case Some(UppercaseVal) =>`
`println("42")`
`case _ => println("Not 42")`
`}`
	`UppercaseVal`i est traité avec la valeur contenue dans val, plutôt qu'un nouvelle variable du "pattern", parce que cela commence par une lettre en capitale. Ainsi, la valeur contenue dans `UppercaseVal` est comparée avec `3`, et "Not 42" est affiché.

Structures de données	
`(1,2,3)`	tuple littéral. (`Tuple3`)
`var (x,y,z) = (1,2,3)`	liaison déstructurée : le déballage du tuple se fait par le “pattern matching”.
Bad `var x,y,z = (1,2,3)`	erreur cachée : chaque variable est associée au tuple au complet.
`var xs = List(1,2,3)`	liste (immuable).
`xs(2)`	indexe un élément par le biais des parenthèses. (transparents)
`1 :: List(2,3)`	créé une liste par le biais de l’opérateur “cons”.
`1 to 5` *est équivalent à* `1 until 6` `1 to 10 by 2`	sucre syntaxique pour les plages de valeurs.
`()` *(parenthèses vides)*	l’unique membre de type Unit (à l’instar de void en C/Java).
structures de constrôle	
`if (check) happy else sad`	test conditionnel.
`if (check) happy` *est équivalent à* `if (check) happy else ()`	sucre syntaxique pour un test conditionnel.
`while (x < 5) { println(x); x += 1}`	boucle while.
`do { println(x); x += 1} while (x < 5)`	boucle do while.

`import scala.util.control.Breaks._`
`breakable {`
`  for (x <- xs) {`
`  if (Math.random < 0.1) break`
`  }`
`}`
	break. (transparents)

`for (x <- xs if x%2 == 0) yield x*10` *est équivalent à* `xs.filter(_%2 == 0).map(_*10)`	pour la compréhension : filter/map
`for ((x,y) <- xs zip ys) yield x*y` *est équivalent à* `(xs zip ys) map { case (x,y) => x*y }`	pour la compréhension : liaison déstructurée
`for (x <- xs; y <- ys) yield x*y` *est équivalent à* `xs flatMap {x => ys map {y => x*y}}`	pour la compréhension : produit cartésien.

`for (x <- xs; y <- ys) {`
`println("%d/%d = %.1f".format(x,y, x*y))`
`}`
	pour la compréhension : à la manière impérative sprintf-style

`for (i <- 1 to 5) {`
`println(i)`
`}`
	pour la compréhension : itère jusqu’à la borne supérieure comprise.

`for (i <- 1 until 5) {`
`println(i)`
`}`
	pour la compréhension : itère jusqu’à la borne supérieure non comprise.

L'orienté objet	

`class C(x: R)` *est équivalent à*
`class C(private val x: R)`
`var c = new C(4)`
	paramètres du constructeur - privé

`class C(val x: R)`
`var c = new C(4)`
`c.x`
	paramètres du constructeur - public

`class C(var x: R) {`
`assert(x > 0, "positive please")`
`var y = x`
`val readonly = 5`
`private var secret = 1`
`def this = this(42)`
`}`
	le constructeur est dans le corps de la classe
	déclare un membre public
	are un accesseur
	déclare un membre privé
	constructeur alternatif

`new{ ... }`	classe anonyme
`abstract class D { ... }`	définition d'une classe abstraite. (qui n'est pas instanciable).
`class C extends D { ... }`	définition d'une classe qui hérite d'une autre.
`class D(var x: R)` `class C(x: R) extends D(x)`	héritage et constructeurs paramétrés. (souhaits : pouvoir passer les paramètres automatiquement par défaut).
`object O extends D { ... }`	définition d'un singleton. (à la manière d'un module)

`trait T { ... }`
`class C extends T { ... }`
`class C extends D with T { ... }`
	traits.
	interfaces avec implémentation. constructeur sans paramètre. mixin-able.

`trait T1; trait T2`
`class C extends T1 with T2`
`class C extends D with T1 with T2`
	multiple traits.

`class C extends D { override def f = ...}`	doit déclarer une méthode surchargée.
`new java.io.File("f")`	créé un objet.
Bad `new List[Int]` **Good** `List(1,2,3)`	erreur de typage : type abstrait : au contraire, par convention : la fabrique appelée masque le typage.
`classOf[String]`	classe littérale.
`x.isInstanceOf[String]`	vérification de types (à l'exécution)
`x.asInstanceOf[String]`	"casting" de type (à l'exécution)
`x: String`	attribution d'un type (à la compilation)