Initiation au Génie Logiciel

Cours 1 et 2

–

Initiation à Scala

Bibliographie

En ligne :

- *Programming in Scala*, M. Odersky, L. Spoon, B. Venners. Artima. `http://www.artima.com/pins1ed/index.html`.
- *An Overview of the Scala Programming Language*, M. Odersky & al. `http://www.scala-lang.org/docu/files/ScalaOverview.pdf`
- *Scala web site*. `http://www.scala-lang.org`

Disponibles à la bibliothèque universitaire :

- *Programming in Scala*, M. Odersky, L. Spoon, B. Venners. Artima. 2010.
- *Scala for the impatient*, Cay Horstmann. Addison-Wesley. 2012.
- *Programming Scala*, Dean Wampler, Alex Payne. O'Reilly. 2009.

Scala en bref

- Scala pour "Scalable language"
  Depuis les petits scripts jusqu'à l'architecture de gros systèmes
- Conçu par Martin Odersky de l'EPFL
  - Expert des langages de programmation
  - Un des concepteurs du compilateur Java courant

- Model objet pur ($\neq$ Java) : *uniquement objets et appels de méthodes*
- Avec de la prog. fonctionnelle : pattern-matching, ordre supérieur, ...
- Scala est compatible avec Java dans les deux sens
- Syntaxe concise ($\neq$ Java)
- Un compilateur et un interprète dans l'IDE (ici : VisualStudio+SBT) `https://video.univ-rennes1.fr/videos/outils-pour-gen-visual-studio-et-sbt/`
- Langage récent mais se propageant très rapidement :

xerox SONY SIEMENS twitter LinkedIn ...

Plan

1. Les constructions de base du langage
   - Types de base et inférence de type
   - Contrôle : if et match - case
   - Boucles : while et for
   - Structures : Arrays, Lists, Sets, Maps
   - Fonctions
2. Modèle Objet
   - Définition de classes et de constructeurs
   - Méthodes : redéfinition, surcharge
   - Objets singletons

Supports et solutions des exercices :
`http://people.irisa.fr/Thomas.Genet/GEN`

Plan

1. Les constructions de base du langage
   - Types de base et inférence de type
   - Contrôle : if et match - case
   - Boucles : while et for
   - Structures : Arrays, Lists, Sets, Maps
   - Fonctions

2. Modèle Objet
   - Définition de classes et de constructeurs
   - Méthodes : redéfinition, surcharge
   - Objets singletons

Types de base et annotations de type

- En Scala, l'annotation de type s'écrit `valeur:Type` au lieu de `Type valeur` utilisé en Java
- `1:Int` `1.45:Double` `"toto":String` `'a':Char` `null:Null`
- Toute donnée est un objet, même les types de base ($\neq$ Java)
  e.g. `1` est un objet et `Int` est son type (c-à-d sa classe)
- Les types sont (généralement) inférés automatiquement

Remarque 1

En cours, on utilise les "worksheets" Scala. Dans VisualStudio, ces fichiers ont l'extension `.worksheet.sc`. L'évaluation de la "worksheet" est déclenchée par l'enregistrement du fichier (Démo).

Exercice 1

Utiliser la méthode `max(Int)` de la classe `Int` pour calculer le maximum de `1+2` et `4`.

Types de base et annotations de type (II)

- En Scala, une instruction=expression qui a une valeur et un type
- Les expressions réalisant des effets de bords, *e.g.* `print("toto")`, ont comme valeur `()` et comme type `Unit`
- La valeur/type d'une séquence d'expressions `i1 ; i2 ; ...; in` est la valeur/type de la dernière instruction

Quiz 1

- *La valeur de `3+3; 4` est* | V | 10 || R | 4 |
- *La valeur de `3+3; 4; print("toto")` est* | V | 4 || R | () |
- *Le type de `3+3; 4; print("toto")` est* | V | Int || R | Unit |

Remarque 2 (le séparateur ";" peut être omis)

S'il y a un saut de ligne entre les expressions

```
print("toto");
print("titi")
```

est équivalent à

```
print("toto")
print("titi")
```


Sous typage : la hierarchie des classes (II)

Définition 1 (Relation sous-type, notée `<:`)

`T'` est un sous-type de `T` (noté `T' <: T`) si toute fonction opérant sur des valeurs de type `T` peut être utilisée sur des valeurs de type `T'`.

Définition 2 (Super-type)

Si `T'` est un sous-type de `T` alors `T` est un super-type de `T'`.

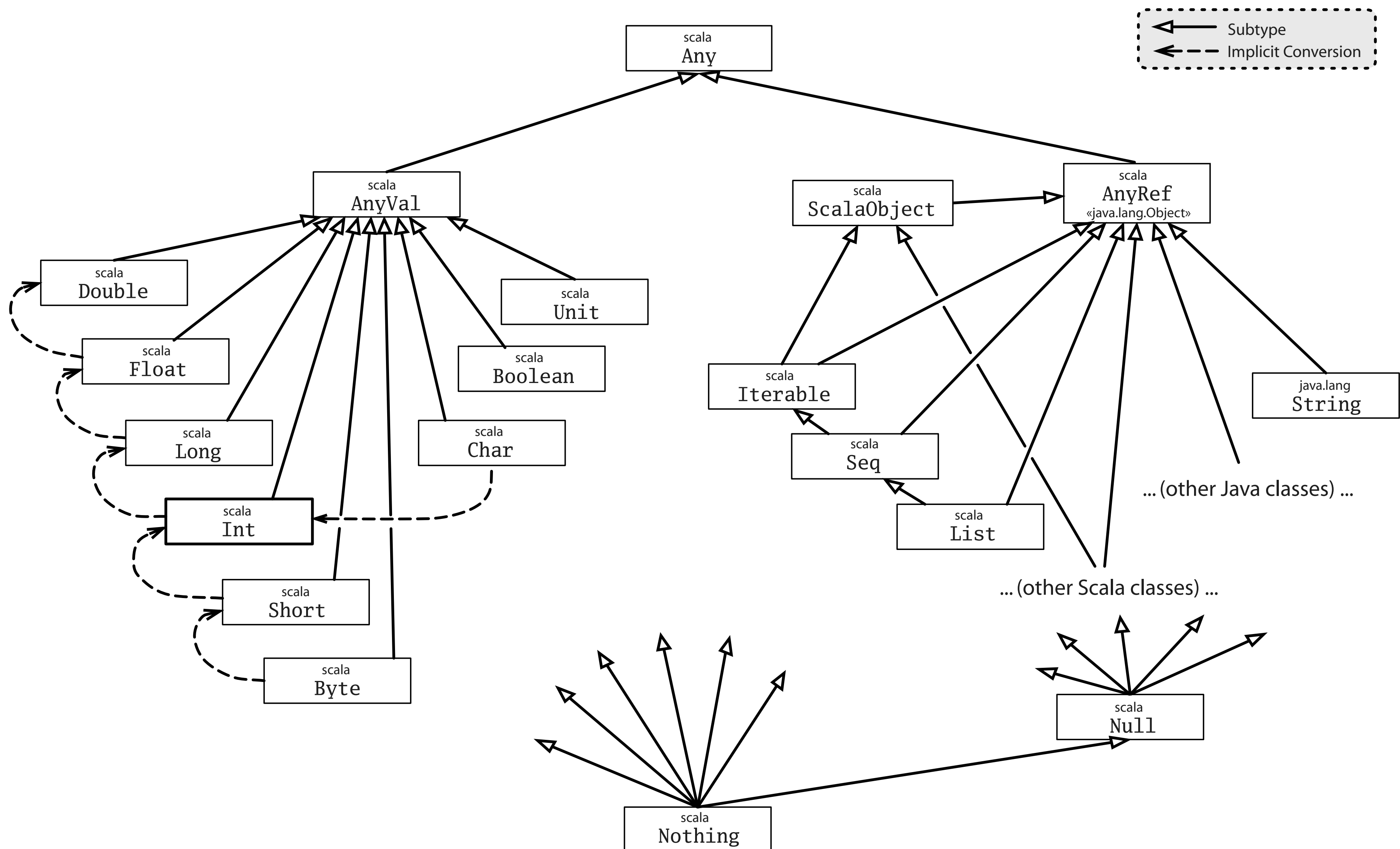
Remarque 3

La relation `<:` correspond à la relation $\rightarrow$ de la figure précédente.

Exemple 3 (Super-types et sous-types)

`AnyVal` est un super-type de `Int`. `List` est un sous-type de `ScalaObject`. `Any` est le super-type de tout type. `Nothing` est sous-type de tout type.

Sous-typage : la hierarchie des classes



Sous typage : la hierarchie des classes (III)

Quiz 2

1. `12` *est de type* `Int` ? — V Vrai | R Faux
2. *A-t-on* `Int <: Any` ? — V Vrai | R Faux
3. `12` *est de type* `Any` ? — V Vrai | R Faux
4. *A-t-on* `Int <: Double` ? — V Vrai | R Faux
5. `12` *est de type* `Double` ? — V Vrai | R Faux
6. `null` *est de type* `List` ? — V Vrai | R Faux
7. `12` *est de type* `Nothing` ? — V Vrai | R Faux
8. `"toto"` *est de type* `Any` ? — V Vrai | R Faux

`val` et `var`

- `val` associe un objet à un identificateur qui ne peut pas être réassigné
- `var` associe un objet à un identificateur qui peut être reassigné
- La philosophie Scala est de privilégier `val` quand c'est possible

---

```
scala> val x=1                    // or val x:Int = 1
x: Int = 1

scala> x=2
<console>:8: error: reassignment to val
       x=2
        ^
scala> var y=1
y: Int = 1

scala> y=2
y: Int = 2
```


Expressions booléennes

- Opérateurs de comparaison `==` et `!=`
  Pour tous les objets (types de base et types définis par l'utilisateur)

Remarque 4

Différent de Java où `==` *compare les types de base et les références, et* `equal` *compare les types définis par l'utilisateur*

- Opérateurs de comparaison `>` et `>=` pour les types "numériques"
  Pour comparer les `Int`, `Double`, `Char`, `String`, . . .
- Connecteurs logiques `&&` (et), `||` (ou) et `!` (not)

Expressions conditionnelles `if`

- La syntaxe est similaire aux instructions `if` Java ...
  mais ce ne sont pas des instructions mais des expressions typées !
- `if ( condition ) e1 else e2`
  type de cette expression est le plus petit super-type de e1 et de e2
- `if ( condition ) e1     // else ()`
  type de cette expr. est le plus petit super-type of e1 et de `Unit`

Quiz 3 (Donnez le plus petit type des expressions suivantes)

1. `if (1==2) 1 else 2` — V `Int` | R `Any`
2. `if (1==2) 1 else "toto"` — V `Int` | R `Any`
3. `if (1==2) 1` — V `AnyVal` | R `Int`
4. `if (1==1) println(1)` — V `Any` | R `Unit`

Expressions conditionnelles match - case

- Remplacent et étendent les switch - case
- La syntaxe est la suivante :

```
e  match {
   case pattern1  => r1   //les patterns sont soit
                          //des valeurs, soit des termes
   case pattern2  => r2   // avec des variables ou des
  ...                     // termes avec des jokers : '_'
   case _ => rn
 }
```

- Rmq : le type de cette expression est le super-type de r1, r2, ... rn

Exemple 4 (Expressions Match-case)

```
x match {
  case "bonjour" => "hello"
  case "au revoir" => "goodbye"
  case _ => "don't know"
 }
```


Expressions conditionnelles match - case

Quiz 4 (Que vaut l'expression suivante)

```
val x= "bonjour"
x match {
  case "au revoir" => "goodbye"
  case _ => "don't know"
  case "bonjour" => "hello"
}
```

V	"hello"
R	"don't know"

Quiz 5 (Que vaut l'expression suivante)

```
val x= "bonj"
x match {
  case "au revoir" => "goodbye"
  case "bonjour" => "hello"
}
```

V	*Non définie*
R	"hello"

Boucles while et for

- Syntaxe du while identique à celle de Java

```
while (x<=10){ ... }
```

- Syntaxe du for est spécifique à Scala

```
for(i <-1 to 10){
... }
```

```
for(i <-10 to (1,-2)){
... }
```


Les tableaux : Array[A]

- Definition de tableaux
  `val t:Array[String]= new Array[String](3)`
- En utilisant l'inférence de types, la définition est moins verbeuse
  `val t= new Array[String](3)`
- La mise à jour des tableaux est standard mais utilise () et non []
  `t(0)="zero"; t(1)="un"; t(2)="deux"`
- Grâce à l'inférence de type et aux constructeurs simplifiés, résumé en :
  `val t= Array("zero","un","deux")`
- Il est possible de définir des tableaux à plusieurs dimensions :

```
val t= Array.fill(2,3)(false)
t(1)(2)=true
```

Les tableaux : Array[A] (II)

Quiz 6 (Le programme suivant est-il valide ?)

```
val t= new Array[Any](10)
t(1)="toto"
t(1)=189
```

V	Oui	R	Non

Quiz 7 (Le programme suivant est-il valide ?)

```
val t= Array("zero","un","deux")
t(1)="one"
```

V	Oui	R	Non

Quiz 8 (Le programme suivant est-il valide ?)

```
val t= Array("zero","un","deux")
t(1)=189
```

V	Oui	R	Non

Les listes : List[A]

- Definition de listes (avec de l'inférence de type)
  `val l= List(1,2,3,4,5)`
- Ajouter un élément en tête d'une liste
  `val l1= 0::l`
- Ajouter un élément en queue d'une liste
  `val l2= l1:+6`
- Concaténer des listes
  `val l3= l1++l2`
- Renverser des listes
  `val l5= l4.reverse // reverse() définie pour List`
  Rmq : les méthodes sans paramètres peuvent être appelées sans ()
- Consulter la valeur d'un élément à une position donnée
  `val i= l(2)`
  Rmq : on ne peut pas modifier la valeur dans l'objet !
  `l(2)=6` échoue ! Les listes sont des objets immutables

Interlude : objets mutables et objets immutables

Définition 5 (Objets mutables et objets immutables)

Un objet est mutable (variable en français) si on peut modifier son contenu après son initialisation. Un objet est immutable (immuable en français) si cela n'est pas possible.

Exemple 6

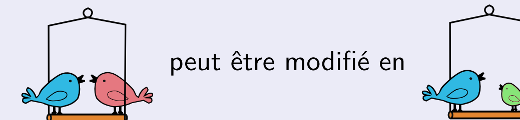
En Scala, les tableaux sont des objets mutables et les listes sont des objets immutables.

Remarque 5

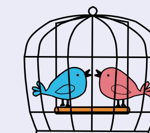
En Scala, la préférence est toujours donnée aux objets immutables. Par exemple, tous les objets de base des librairies Scala sont immutables : collections, listes, ensembles, maps, etc. C'est une volonté des concepteurs de Scala pour éviter les bugs liés aux effets de bords.

Interlude : objets mutables et objets immutables (II)

Un objet mutable peut être modifié après initialisation

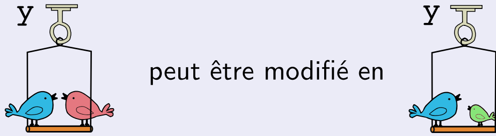


Un objet immutable ne peut pas

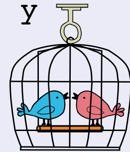


Interlude : objets mutables et objets immutables (III)

Un objet mutable associé à une variable y par val peut être modifié



Mais un objet immutable associé à y par val ne peut pas



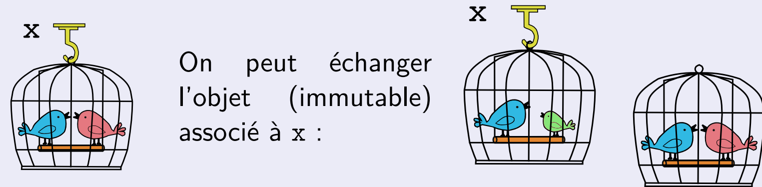
Exemple 7

```
val y= Array("bleu","rouge")
y(1)= "vert"
```

```
val y= List("bleu","rouge")
y(1)= "vert" // échoue!
```


Interlude : objets mutables et objets immutables (IV)

Un objet associé à une variable x par var peut être échangé



Exemple 8

```
var x= List("bleu","rouge")
x= List("bleu","vert") // ou x= x.updated(1,"vert")
                       // qui crée aussi une nouvelle liste
```

Remarque 6

On arrive au même résultat que dans le cas des tableaux mais sans avoir modifié les objets existants : on ne fait que créer des nouveaux objets.

Les listes : List[A] (II)

Quiz 9 (Le programme suivant est-il valide ?)

```
val li= List("zero","un","deux")
li(1)="one"
```

V	Oui	R	Non

Quiz 10 (Le programme suivant est-il valide ?)

```
var li= List("zero","un","deux")
li(1)="one"
```

V	Oui	R	Non

Quiz 11 (Le programme suivant est-il valide ?)

```
val li= List(1,"toto",2)
val l2= li ++ List(3,4)
```

V	Oui	R	Non

Les listes : List[A] (III)

Quiz 12 (Le programme suivant est-il valide ?)

```
var li= List(1,2,3)
li= li ++ List(5,6)
```

V	Oui	R	Non

Quiz 13 (Quel est le résultat affiché par le programme suivant)

```
val t1= Array(4,5,6)
val t2= t1
t2(1)= -4
println(t1(1))
```

V	-4	R	5

Quiz 14 (Quel est le résultat affiché par le programme suivant)

```
var li= List(1,2,3)
var l2= li
l2= l2.updated(1,10)
println(li(1))
```

V	10	R	2

Boucles `for` revisitées pour les types `Traversable`

- Pour tous les sous-types de `Traversable` (Tableaux, Listes, Collections, Ensembles, Tables, ...)
- `for (ident <- s ) e`
  Rmq : `s` doit être de type `Traversable`
- L'effet est d'exécuter `e` autant de fois qu'il y a d'éléments dans `s`, avec `ident` prenant successivement sa valeur parmi les éléments de `s`

Exercice 2

En utilisant `for` *et* `println`*, afficher tous les carrés d'une liste d'entiers.*

Exercice 3

En utilisant `for`*, construire la liste des carrés d'un tableau d'entiers.*

Les ensembles (immutables) : `Set[A]`

- Définition d'ensemble avec inférence de type
  `val s= Set(1,2,3,4,5)`
- Pour tester l'appartenance : méthode `contains(e:A)`
  e.g. `s.contains(2)`
- Il existe des dizaines d'autres méthodes (démo Scala library) + (démo complétion automatique sous VisualStudio)

`intersec(s:Set[A])`	`union(s:Set[A])`	`-(e:A)`
`+(e:A)`	`--(s:Set[A])`	`==(s:Set[A])`
`empty:Set[A]`	`max: A`	`nonEmtpy:Boolean`
`size:Int`	`subsetOf(s:Set[A])`	`...`

Exercice 4

Soit e1 *l'ensemble* $\{1, 2, 3, 4\}$ *et* e2 *l'ensemble* $\{3, 4, 10\}$*. Vérifiez en Scala que l'intersection de* e1 *et de* e2 *est incluse dans* e1 *et dans* e2*.*

Les n-uplets (immutables) : `(A,B,C,...)`

- Definition de n-uplets (avec inférence de type)
  `val t= (1,"toto",18.3)`
  Rmq : le type inféré pour `t` est : `(Int, String, Double)`
- Accesseurs pour les n-uplets : `t._1, t._2,` etc.
- ... ou avec `match-case` :

```
t match { case (2,"toto",_) => "found!"
          case (_,x,_) => x
}
```

Quiz 15 (L'expression précédente s'évalue en)

V	"found!"	R	"toto"

- ... ou simplement avec `val` ou `var` :
  `val (e1,e2,e3)= (1,"toto",18.3)`

Les tables (immutables) : `Map[A,B]`

- Les tables associent des clés (type `A`) à des valeurs (type `B`)
- Définition d'une table (avec inférence de type)
  `var m= Map('C' -> "Carbon",'H' -> "Hydrogen")`
  Rmq : le type inférré pour `m` est `Map[Char,String]`
- Savoir si une clé figure dans une table
  `m.contains('C')`
- Recherche avec valeur par défaut quand la clé n'apparaît pas
  `m.getOrElse('K',"Unknown")`
- Ajout d'une association à une table (immutable)
  `m= m+('O' -> "Oxygen")`

Les tables (immutables) : Map[A,B] (II)

Quiz 16 (A la fin de l'exécution du programme, le booléen b vaut ?)

```
val m= Map(1 -> "toto",2 -> "titi")
val b= (m.getOrElse(3,"non") == "ok")
```

V	true
R	false

Quiz 17 (A la fin de l'exécution du programme, le booléen b vaut ?)

```
var m= Map(1 -> "toto",2 -> "titi")
m= m +(3 -> "ok")
val b= (m.getOrElse(3,"non") == "ok")
```

V	true
R	false

Quiz 18 (A la fin de l'exécution du programme, le booléen b vaut ?)

```
var m= Map(1 -> "toto",2 -> "titi")
m= m +(2 -> "ok")
val b= (m.getOrElse(2,"non") == "ok")
```

V	true
R	false

Définition de fonctions

- `def f ( arg1: Type1, ..., argn:Typen ): R = e`
- le type de e et R doivent concorder
- e peut être une séquence d'expressions `{e1; e2; ...; en}`
- Le type de f est : `(Type1,...,Typen) R`

Remarque 7 (Si la fonction *n'est pas récursive*, R peut être inférré)

Dans ce cas, la syntaxe définition de fonction devient :
`def f ( arg1: Type1, ..., argn: Typen )= e`

Exemple 9

```
def plus(x:Int,y:Int):Int={
     println("Sum of "+x+" and "+y+" is equal to "+(x+y))
     x+y  // mot clé return inutile
   }      // le résultat de la fonction est la dernière expression
```


Définitions de fonctions (II)

Quiz 19 (Ces définitions de fonction sont-elles valides ?)

1. `def plus(x:Int,y:Int):Unit={x+y; println("x+y")}`

V	Oui	R	Non

2. `def plus(x,y):Int={x+y}`

V	Oui	R	Non

3. `def plus(x:Int,y:Int){x+y}`

V	Oui	R	Non

4. `def plus(x:Int,y:Int)=x+y`

V	Oui	R	Non

Exercice 5

En utilisant une table, définissez un annuaire ainsi que les fonctions
`ajoutAbonné(nom:String,tel:String):Unit`,
`obtenirTel(nom:String):String`,
`obtenirListeAbonnés:List[String]`.

Plan

1 Les constructions de base du langage
- Types de base et inférence de type
- Contrôle : if et match - case
- Boucles : while et for
- Structures : Arrays, Lists, Sets, Maps
- Fonctions

2 Modèle Objet
- Définition de classes et de constructeurs
- Méthodes : redéfinition, surcharge
- Objets singletons

Définition de classes et de constructeurs

- `class C(v1 : type1, . . . , vn :typen) { ... }` — constructeur primaire

  e.g.
```
class Rational(n:Int,d:Int){
      val num=n       // peut utiliser var
      val den=d       // pour avoir des objets mutables
      def isNull:Boolean=(num==0)
      ...
```
- Les classes peuvent avoir des constructeurs auxiliaires :
```
      def this(n:Int)={this(n,1)}
```
- Création d'objets avec les constructeurs primaires/auxiliaires
```
      new Rational(3,4)          new Rational(4)
```
  Démo dans VisualStudio (création d'objet + toString par défaut)

Raccourcis d'écriture pour les appels de méthodes

Soit le programme suivant :

```
class T(s:String){
  var lab=s
  def concat(x:String)=lab+x
  def double=lab+lab
  def sep(x:Int,y:String)=x.toString+y
}

val t= new T("toto")
```

Sur l'objet `t` de type `T` les appels suivants sont possibles :

```
t.lab              // donne la valeur du champ lab
t.lab="titi"       // reassigne la valeur du champ lab
t.double           // appel d'une méthode sans paramètres
t.concat("hop")
t concat "hop"     // équivalent au précédent
t.sep(1,"hop")
t sep (1,"hop")     // équivalent au précédent
```


Raccourcis d'écriture pour les appels de méthodes (II)

Remarque 8 (Respect du modèle objet)

En Scala toute opération, réalise un appel de méthode. C'est également vrai pour les accès aux champs et les opérations arithmétiques (≠ Java).

- `1 + 2` *exécute* `1.+(2)`
- `t.lab` *exécute* `t.lab()`
- `t.lab="titi"` *exécute* `t.lab_=("titi")`

où `lab():String` *et* `lab_=(x:String)` *sont définies automatiquement*

Remarque 9 (Explication de la syntaxe du `for`)

Dans `for (i <- 1 to 5)`*, l'expression* `1 to 5` *est en fait un raccourci d'écriture pour* `1.to(5)` *qui vaut* `Range(1,2,3,4,5)` *où* `Range` *est un type* `Traversable`*.*

Exercice 6 (Compléter la classe `Rational`)

Définissez une fonction `add(r:Rational):Rational`*.*

Définition de Classes et de constructeurs (II)

Quiz 20 (Ces programmes sont-ils corrects ?)

Programme	Réponse
`class T(x:Int)` `val t= new T(10)`	V Oui / R Non

`class T(x:Int){`
`  val y=x }`
`val t= new T(10)`
`println(t.y)`
	V Oui / R Non

`class T(x:Int){`
`  val y=x }`
`val t= new T(10)`
`t.y=20`
	V Oui / R Non

Méthodes : redéfinition et surcharge

- On peut redéfinir une méthode existante en utilisant override

```
override def f(...)
```

Exercice 7

La méthode toString:String est définie par défaut pour tous les types. Redéfinissez-la pour la classe Rational.

- Toutes les méthodes/fonctions/opérateurs peuvent être surchargés

Définition 10 (Surcharge de méthodes)

Une méthode est surchargée s'il existe d'autres méthodes de même nom ne se différentiant que par le type de ses entrées ou le type de son résultat.

- En particulier, les opérateurs peuvent être surchargés `def +(x:T):T`

Exercice 8

Définir l'opérateur + pour la classe Rational.

Conversions implicites

Quiz 21

1. *Peut-on calculer `"toto" + 1` en Scala ?* V Oui R Non
2. *Quel opérateur de `String` est utilisé ?* V `+(x:String):String` R `+(x:Int):String`
3. *`1` est-il de type `String` ?* V Oui R Non

- Une conversion implicite traduit un objet `t:T` vers un objet `t':T'`
- Un grand nombre de conversions implicites sont prédéfinies en Scala : par exemple du type `Int` vers le type `String`. Pour `"toto" + 1`, `1` est converti automatiquement en `"1":String`.
- Il est possible de définir vos propres conversions à l'aide d'implicit
  e.g. `implicit def bool2int(b:Boolean):Int= if b 1 else 0`

Exercice 9

Pour Rational, ajouter une conversion implicite de Int vers Rational.

Objets singletons

- Les objets sont soit des *instances* de classes (`new`) soit des *singletons*
- Les objets singletons sont définis en utilisant le mot clé `object`

```
object PlusInfinity{
      override def toString="+oo"
      def +(x:Double)=PlusInfinity
      def +(x:this.type)=PlusInfinity
}
```

Remarque 10

Comme en Java, dans la définition d'une classe/objet, `this` fait référence à l'objet courant. Pour les singletons, `this.type` est le type de cet objet.

Un singleton est désigné par son nom et est déjà initialisé (~~new~~)

```
scala> PlusInfinity + 876
PlusInfinity.type = +oo
scala> PlusInfinity + PlusInfinity
PlusInfinity.type = +oo
```

Exercice 10

Que faut-il ajouter pour avoir `876 + PlusInfinity= +oo` ?

Objets singleton spécifiques : les applications

Un objet singleton étendant la classe `App` est exécutable

Exemple 11 (Objet singleton exécutable)

```
object HelloWorld extends App{
    println("Hello, world!")
}
```

Exercice 11

Créez votre première application Scala, en définissant un objet singleton étendant la classe App. Cette application créera deux rationnels et en affichera la somme.

Classes, objets, objets singletons

Quiz 22 (Ce programmes créent combien d'objets en mémoire ?)

```
class C(i:Int){ val x=i }
val a= new C(1)
val b= new C(1)
val c= new C(2)
```

V	2	R	3

```
class C(i:Int){ val x=i }
val a= new C(1)
val b= a
val c= new C(2)
```

V	2	R	3

```
class C(i:Int){ val x=i }
object O{ val y=1 }
val a= new C(1)
val b= O
val c= O
```

V	2	R	3