CM GEN - Table des matières

- **CM1 & 2 : Initiation à Scala**
  - 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
  - Modèle Objet
    - Définition de classes et de constructeurs
    - Méthodes : redéfinition, surcharge
    - Objets singletons
- **CM3 : Architecture logicielle**
  - Principes de base d'architecture logicielle orientée objet
    - Encapsulation
    - Héritage et association
    - Diagrammes de classes
    - Héritage et association en Scala
    - Traits
  - Principes avancés de programmation orientée objet
    - Case classes et match-case
    - Packages
    - Importation de code
- **CM4 : Quelques outils pour le génie logiciel**
  - Gestion de version
  - Tests unitaires
  - Couverture et pertinence de tests
  - Génération de documentation

- **CM5 : Concepts avancés de programmation fonctionnelle et de programmation objet**
  - Fonctions d'ordre supérieur et fonctions anonymes
  - Polymorphisme
  - Traits et fabriques d'objets
  - Héritage, visibilité et redéfinition
  - Traits étendus
  - Exceptions
- **CM6 : Introduction au “développement agile”**
  - Cycles de développement
  - Méthodes agiles, principes généraux
  - Comment se passe un Sprint ?
  - Principes de développement agiles : TDD, YAGNI, KISS
- **CM7 : Propriétés logiques et programmation par contrats**
  - Invariants, pre et post conditions
  - Des formules logiques pour exprimer des propriétés
  - Application en GL : factorisation des tests
  - Application en GL : programmation défensive
  - Application en GL : programmation par contrats
  - Parenthèse culturelle : au delà des tests et des contrats

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 :

...

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'`.

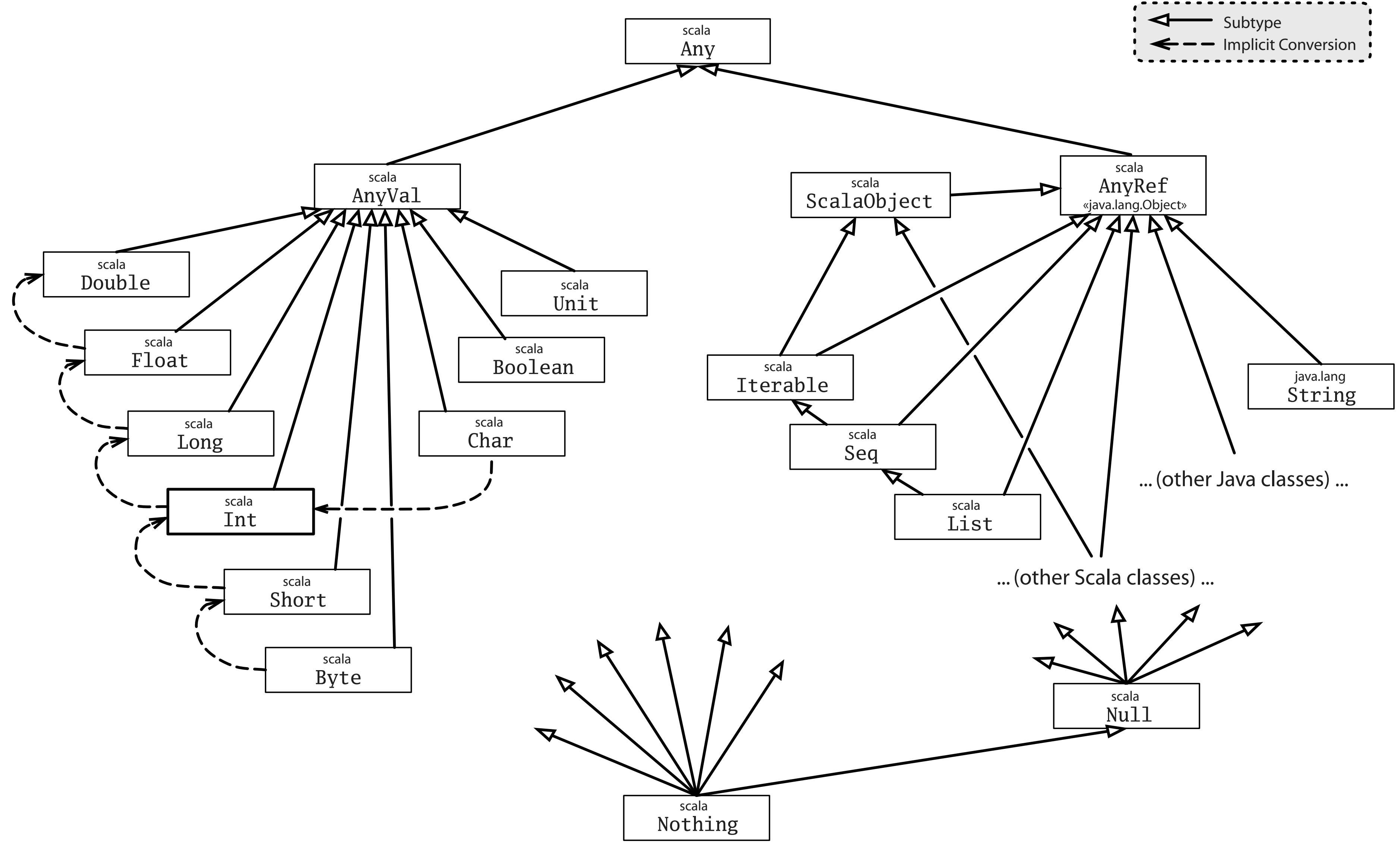
Remarque 3

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

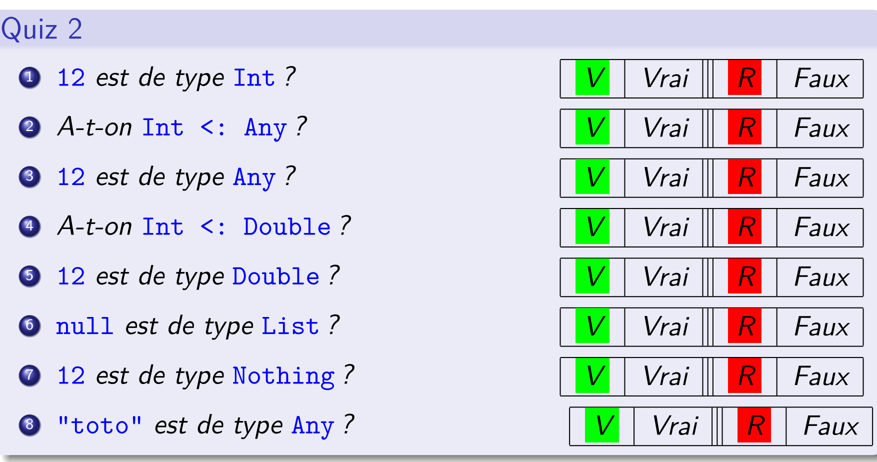
Exemple 3 (Super-types et sous-types)

`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)



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 ?)

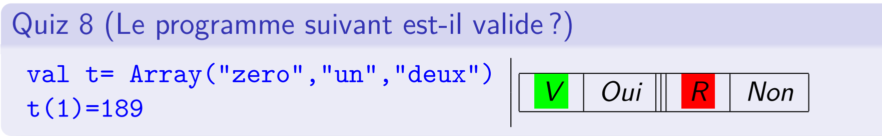
```
val t= new Array[Any](10)
t(1)="toto"
t(1)=189
```

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

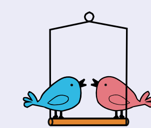
En Scala, les tableaux sont des objets mutables et les listes sont des objets immutables.

Remarque 5

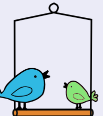
En Scala, la préférence est toujours donnée aux objets immutables. Par exemple, tous les objets de base des librairies Scala sont immutables : collections, listes, ensembles, maps, etc. C'est une volonté des concepteurs de Scala pour éviter les bugs liés aux effets de bords.

Interlude : objets mutables et objets immutables (II)

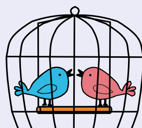
Un objet mutable peut être modifié après initialisation

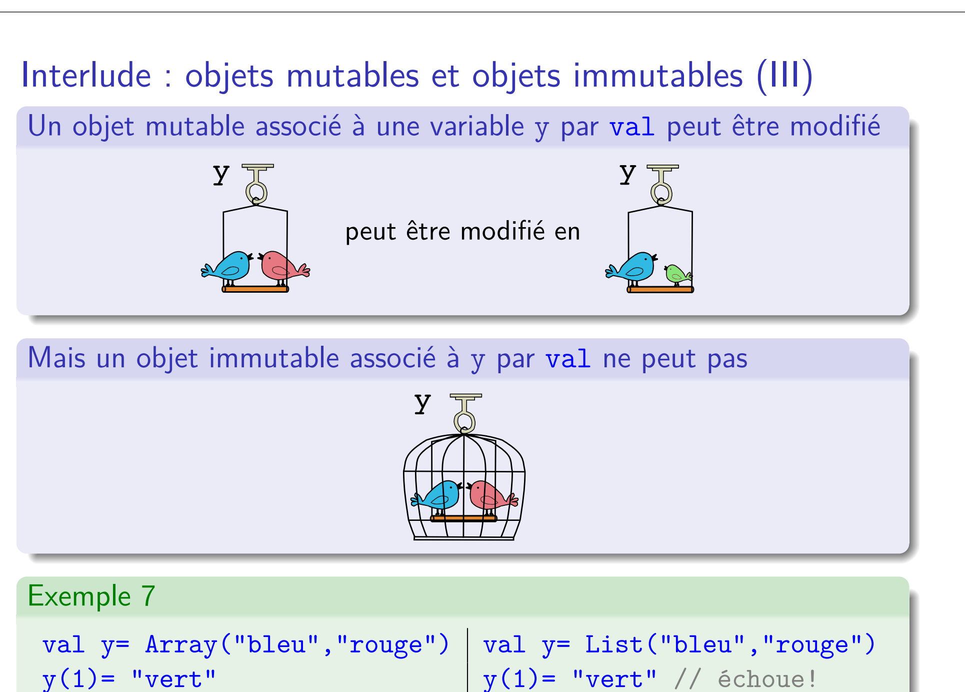


peut être modifié en



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

y — peut être modifié en — y

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

y

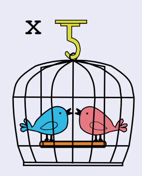
Exemple 7

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

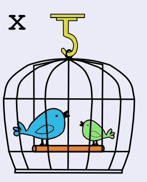
```
val y= List("bleu","rouge")
y(1)= "vert" // échoue!
```

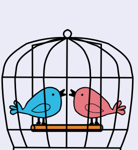

Interlude : objets mutables et objets immutables (IV)

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



On peut échanger l'objet (immutable) associé à `x` :



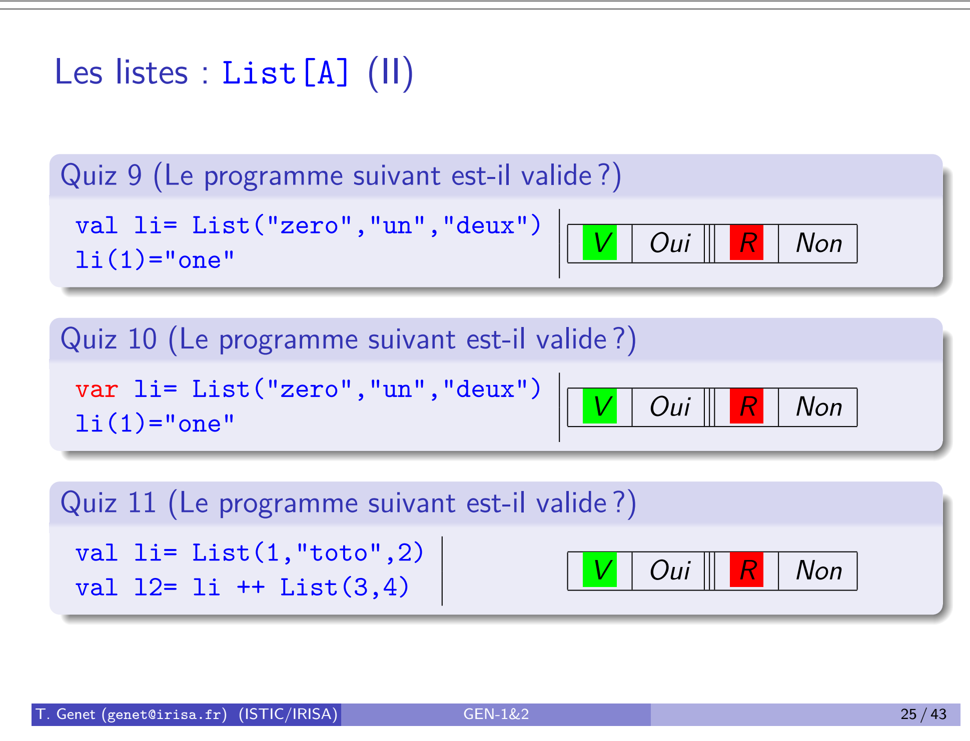


Exemple 8

```
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

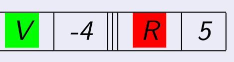
Les listes : `List[A]` (III)

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

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

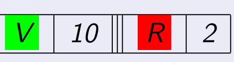
Quiz 13 (Quel est le résultat affiché par le programme suivant)

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



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))
```



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

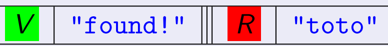
Soit e1 l'ensemble $\{1, 2, 3, 4\}$ et e2 l'ensemble $\{3, 4, 10\}$. Vérifiez en Scala que l'intersection de e1 et de e2 est incluse dans e1 et dans e2.

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)



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

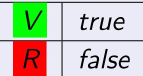
Les tables (immutables) : Map[A,B]

- 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)

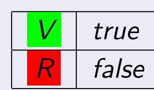
Quiz 16 (A la fin de l'exécution du programme, le booléen b vaut ?)

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



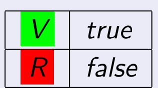
Quiz 17 (A la fin de l'exécution du programme, le booléen b vaut ?)

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



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")
```



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)

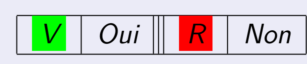
Dans ce cas, la syntaxe définition de fonction devient :
`def f ( arg1: Type1, ..., argn: Typen )= e`

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}`
   

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 (désigne `C(v1 : type1, ..., vn :typen)`)

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 ($\neq$ 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

Initiation au Génie Logiciel

Cours 3

–

Architecture logicielle

Ce que nous allons voir dans ce cours

Quelques principes de base d'*Architecture logicielle* (orientée objet)

- Encapsulation
- Relation d'association
- Relation d'héritage
- Interfaces/Traits
- Le découpage en packages

L'objectif commun de ces notions est d'*abstraire* un programme

... pouvoir l'utiliser ou le concevoir en s'abstrayant des détails

Ces principes seront approfondis en L3 (UE BMO)

Nous verrons la mise en oeuvre de ces principes en Scala ainsi que :

- Les case classes

Plan

1. Principes de base d'architecture logicielle orientée objet
   - Encapsulation
   - Héritage et association
   - Diagrammes de classes
   - Héritage et association en Scala
   - Traits
2. Principes avancés de programmation orientée objet
   - Case classes et match-case
   - Packages
   - Importation de code

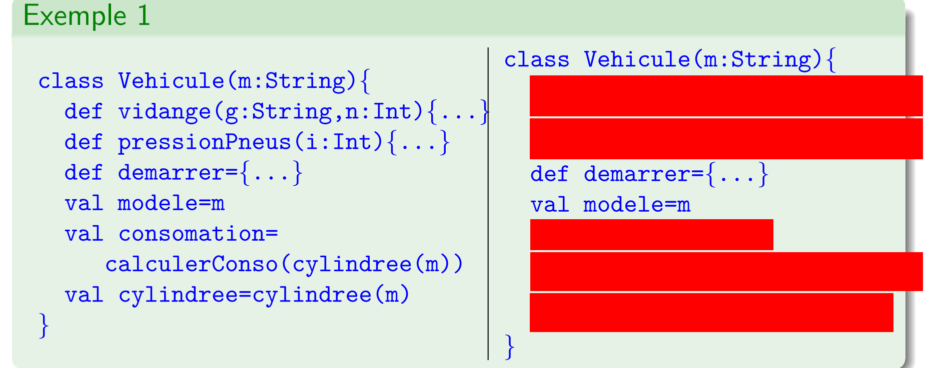
Plan

1. Principes de base d'architecture logicielle orientée objet
   - Encapsulation
   - Héritage et association
   - Diagrammes de classes
   - Héritage et association en Scala
   - Traits
2. Principes avancés de programmation orientée objet
   - Case classes et match-case
   - Packages
   - Importation de code

Encapsulation

L'encapsulation abstrait un code du point de vue de son utilisateur :

- masque les détails inutiles (les rend invisibles)
- améliore la visibilité des éléments importants



Rmq : c'est ce qui manquait à la classe `Graphe` du package `C` du TP2 !

Encapsulation (II)

En Scala (comme en Java), en déclarant un *membre* `private`, on le rend invisible en dehors de l'objet/classe. Membre=(champ/méthode/classe/objet)

Remarque 1 (Visibilité par défaut : `public`)

Si on ne déclare pas un membre `private` *il est implicitement visible (comme déclaré* `public`*).*

Remarque 2 (Protège les objets contre des altérations extérieures)

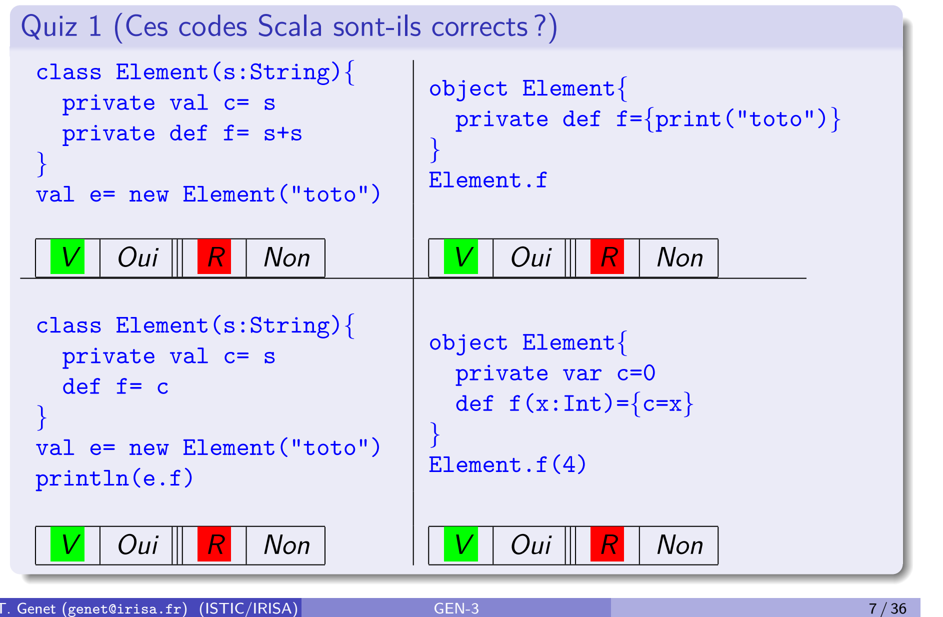
```
class Compte{
  var m=0
  def déposer(x:Int){m=m+x}
  def retirer(x:Int){m=m-x}
}
val c1= new Compte
c1.déposer(1000)

c1.m = c1.m - 200 // réussi! ☹
```

```
class Compte{
  private var m=0
  def déposer(x:Int){m=m+x}
  def retirer(x:Int){m=m-x}
}
val c2= new Compte
c2.déposer(1000)

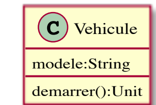
c2.m= c2.m - 200 // échoue! ☺
```


Encapsulation (III)

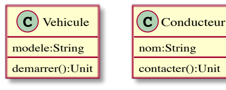


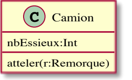
Et après l'encapsulation ?

- On peut voir un objet/classe encapsulé comme un « composant » disposant de valeurs (ici, `modele`) et offrant des opérations (ici, `demarrer`) :



- On peut obtenir des « composants » similaires pour d'autres classes :





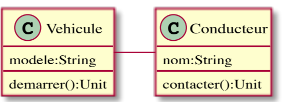
C Roue
diametre:Long
gonfler():Unit

- Pour les combiner on utilise deux relations : l'association et l'héritage

Association d'objets/classes

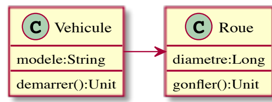
Définition 2 (Relation d'association - association réciproque)

L'*association* entre une classe A et une classe B, notée [A — B] signifie que les objets de classe A « ont des » objets de classe B, et réciproquement.



Remarque 3 (Association orientée)

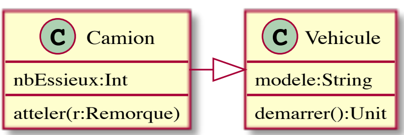
Si la relation n'est pas réciproque, on utilise l'association orientée [A → B]



Héritage d'objets/classes

Définition 3 (Relation d'héritage)

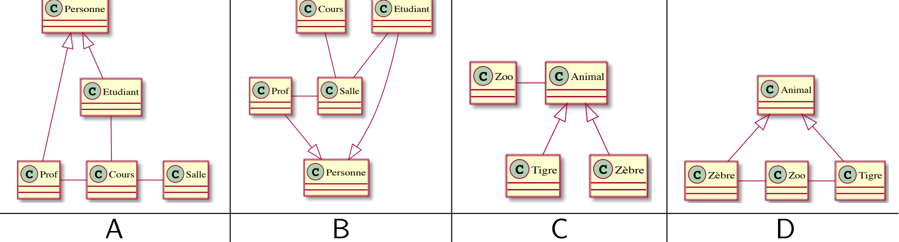
L'héritage entre une classe A et une classe B, noté [A ▷ B], signifie que les objets de classe A « sont des »objets de classe B. Ainsi, les objets de la classe A ont (au moins) tous les champs et opérations de la classe B.



Remarque 4 (Lien avec le sous-typage (par exemple en Scala))

La relation d'héritage coïncide avec la relation de sous-typage. Si une classe A *hérite d'une classe* B*, alors* A *est un sous-type de* B*. Par exemple,* `Int` *hérite de* `Any`*.*

Diagrammes de classes, le quizz !

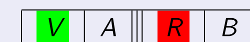


Quiz 2

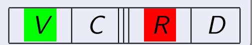
1. *Les diagrammes A,B,C et D sont-ils corrects ?*

V	Oui	R	Non

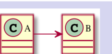
2. *Entre A et B, quel est le diagramme le plus pertinent ?*



3. *Entre C et D, Quel est le diagramme le plus pertinent ?*

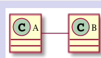


Implantation des associations et héritages en Scala

Implantation d'une association orientée 

On ajoute à la classe A un champ contenant une (ou des) références vers des objets de classe B. Par exemple :

- Un A a exactement un B : `class A{val objB: B ...}`
- Un A a 0, 1, 2,..., n B : `class A{val objB: Set[B]...}`
- Un A a 0, 1, 2,..., n B, et l'ordre des B est important : `class A{val objB: List[B]...}`

Implantation d'une association réciproque 

Pour une association réciproque, en plus du code précédent, on ajoute à la classe B un champ contenant une (ou des) références vers des objets de A. Par exemple, un A a exactement un B qui a 0, 1, 2,..., n A :

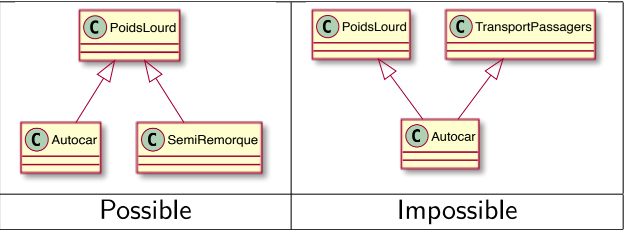
- `class A{val objB: B ...}` et `class B{val objA: Set[A]...}`

Implantation des associations et héritages en Scala (II)

Implantation d'une relation d'héritage avec extends

```
class B { ... code de la classe B }
class A extends B { ... code de la classe A }
```

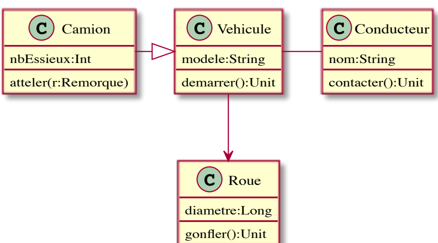
Remarque : En Scala on ne peut hériter que d'une classe au plus !



On en reparlera dans le cours de concepts objets avancés...

Association et héritage en Scala

Les diagrammes obtenus sont nommés *diagrammes de classe*



Exercice 1

Implantez succinctement en Scala le diagramme de classe précédent.

Remarque 5 (Un diagramme de classe peut contenir plus d'informations)

Les types de relations sont plus nombreux, elles peuvent présenter des cardinalités, etc. Vous les étudierez plus en détail en L3 (UE BMO)

Traits

Définition 4 (Traits)

Un trait définit des types d'objets, sans donner leur implantations. On parle aussi de types abstraits (en SI2) ou d'interfaces (en PO/Java).

« Une interface est la couche limite entre deux éléments par laquelle ont lieu des échanges et des interactions »

Remarque 6 (Les traits facilitent le développement collaboratif)

Un trait déclare un ensemble de champs/méthodes avec leurs types. Il sépare un logiciel en deux parties : une partie utilisatrice des champs/méthodes et une partie implantant ces champs/méthodes.

Traits (II)

Exemple 5 (Trait pour un logiciel manipulant des files d'entiers)

Utilisateurs	Interface	Implanteurs

Une équipe réalise un logiciel *utilisant* des `IntQueue`	`trait IntQueue {`
	`def get:Int`
	`def put(x:Int):Unit`
	`def empty:Boolean`
	`}`
		Une équipe *implante* le trait `IntQueue` dans une classe `MyQueue`

```
def vider(q:IntQueue):Unit=
  while(!q.empty) q.get

def incAll(q:IntQueue):Unit={
  var l= List[Int]()
  while(!q.empty) l= q.get::l
  for (e<-l.reverse) q.put(e+1)
}
```

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0)
            b=b.drop(1)
            h}
  def put(x:Int):Unit= {b=b:+x}
  def empty=b.isEmpty
}
```

Traits (III)

- Le mot clé `extends` précise qu'une classe implante un trait

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0); b=b.drop(1); h}
  def put(x:Int):Unit= {b=b:+x}
  def empty=b.isEmpty
 }
```

Remarque 7

Pour implanter un trait, une classe doit implanter tous les champs et méthodes qui n'ont pas d'implantation (en respectant les types de ceux-ci).

- Les utilisateurs de `IntQueue` n'ont pas besoin de connaître l'implantation `MyQueue` pour développer leur partie :

```
def vider(q: IntQueue):Unit= while(!q.empty) q.get
```

- Sauf pour créer un objet de ce type : `val q= new MyQueue`

Traits (IV)

Remarque 8

Abstraire le fonctionnement d'une partie d'un logiciel par un trait, le rend plus robuste à l'évolution de l'implantation de cette partie.

Exemple 6

On peut remplacer l'implantation `MyQueue` de `IntQueue` par une autre sans toucher au code de la partie utilisateur : `vider`, `incAll`.

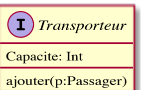
Dans le TP2, les implantations `D`, `E` et `F` sont interchangeables car elles respectent le même **trait**. Ce n'est pas le cas pour `B` et `C`.

Exercice 2

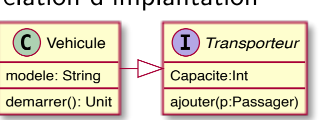
Quel pourrait être le trait abstrayant le fonctionnement de la classe `Graphe` *du TP1, pour pouvoir facilement changer d'implantation de graphes sans changer le code de l'objet* `Canalisations`.

Représentation des traits dans les diagrammes

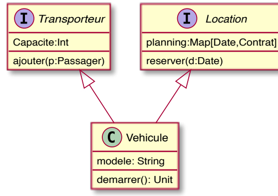
Représentation des `traits`/Interfaces



Représentation de la relation d'implantation



Une classe/objet peut implanter plusieurs traits :




```
class Vehicule
  extends Transporteur
  with Location {
   ... code de la classe
       Vehicule ...
}
```

Exercice 3 (Complétez la déclaration des classes `Vehicule`, etc.)

Faites apparaître les traits `Location` *et* `Transporteur`.

Traits (III)

Quiz 3 (Ces codes Scala sont-ils corrects ?)

```
trait A{
  val c:String
}
val a= new A
```

V Oui | R Non

```
class A{
    val c:String
}
val a= new A
```

V Oui | R Non

Quiz 4 (Ces codes Scala sont-ils corrects ?)

```
trait A{
  val c:String
}
object B extends A{
  val c="toto"
}
```

V Oui | R Non

```
trait A{
  def f(x:Int):String
}

class B extends A{
  def f(x:Any):String="oui!"
}
```

V Oui | R Non

Traits (IV)

Exercice 4 (Développement collaboratif d'un navigateur Web)

Le navigateur parcourt des pages localisées par des URLs. Les pages contiennent des éléments clicables ou non (du texte et des images). Le navigateur est muni d'un historique des pages visitées.

- *Quels traits proposeriez-vous (pour au moins 6 équipes) ? avec quels champs/opérations ?*
- *Quelles relations définiriez-vous entre ces traits ? (association/héritage)*

Quiz 5 (Pour le navigateur, quelle est la solution la plus pertinente ?)

```
trait Navigateur{ ... }
class MonNav
      extends Navigateur {
  ... }
```

V

```
trait Navigateur{ ... }
object MonNav
       extends Navigateur {
  ...}
```

R

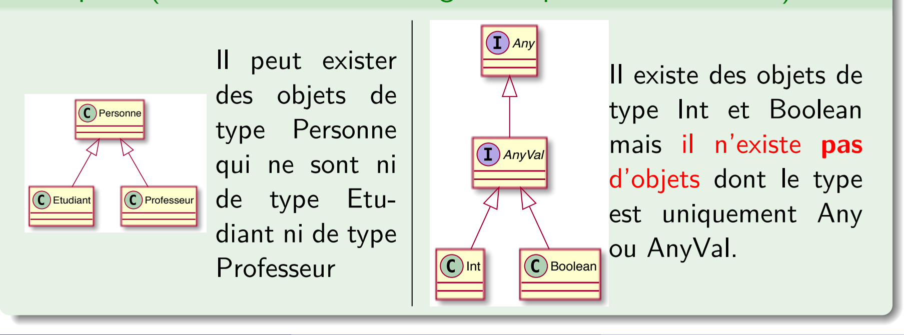
Traits (V)

Les traits facilitent le développement collaboratif mais aussi le sous-typage

Remarque 9 (Les traits pour le sous-typage)

Un trait permet de définir quelles sont les champs/opérations communes entre plusieurs classes (c'est un super-type). Cela remplace l'héritage quand le super-type **n'a pas d'objets propres**.

Exemple 7 (Différence entre héritage et implantation de traits)

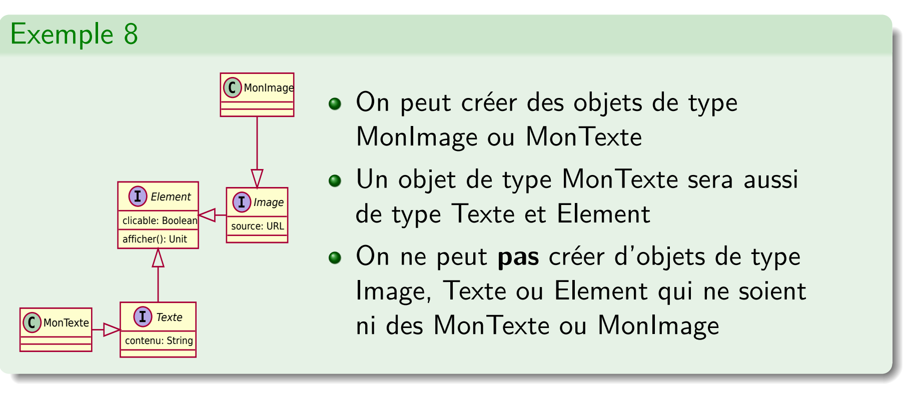


Il peut exister des objets de type Personne qui ne sont ni de type Etudiant ni de type Professeur

Il existe des objets de type Int et Boolean mais il n'existe **pas** d'objets dont le type est uniquement Any ou AnyVal.

Traits (VI)

Exemple 8



- On peut créer des objets de type MonImage ou MonTexte
- Un objet de type MonTexte sera aussi de type Texte et Element
- On ne peut **pas** créer d'objets de type Image, Texte ou Element qui ne soient ni des MonTexte ou MonImage

Plan

1 Principes de base d'architecture logicielle orientée objet
- Encapsulation
- Héritage et association
- Diagrammes de classes
- Héritage et association en Scala
- Traits

2 Principes avancés de programmation orientée objet
- Case classes et match-case
- Packages
- Importation de code

Case class/object définissent des types algébriques

« Un type algébrique de données est un type de données dont chacune des valeurs est une donnée d'un autre type enveloppée dans un des constructeurs du type. Toutes les données enveloppées sont des arguments du constructeur. La seule manière d'opérer sur les données est d'enlever le constructeur en utilisant le filtrage par motif. »

- `case class Rect(x1:Int,y1:Int,x2:Int,y2:Int) // Rectangles`
- Les instances de `case class` sont initialisées sans `new` (c-à-d que les objets de `case class` sont obtenus sans `new`)

```
scala>  val r1= Rect(0,0,5,7)
```

- L'accès aux données de l'objet peut se faire par `match-case`

```
scala>  r1  match {case  Rect(_,_,x,y)  => (x,y)}
(Int,Int) = (5,7)
```

- Par défaut, `toString` et `equals` respectent la structure des objets

```
scala> println(r1)
Rect(0,0,5,7)
```

```
scala>  val r2= Rect(0,0,5,7)
scala>  r1==r2
Boolean = true
```


Case classes et case objects, premier quizz

Quiz 6 (Qu'affichent ces programmes ?)

```
class P(x:Int)
val p= new P(1)
val p2= new P(1)
println(p==p2)
```

V true || R false

```
case class P(x:Int)
val p= P(1)
val p2= P(1)
println(p==p2)
```

V true || R false

Quiz 7 (Qu'affichent ces programmes ?)

```
class P(x:Int)
val p= new P(1)
val p2= new P(2)
println(p==p2)
```

V true || R false

```
case class P(x:Int)
val p= P(1)
val p2= P(2)
println(p==q)
```

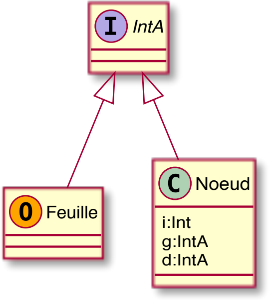
V true || R false

Case classes et case objects : applications

Les case classes permettent de définir simplement et de manipuler efficacement tout type d'objet non mutable

Exemple 9 (Définition des arbres binaires d'entiers par case class)

```
trait IntA   // Arbres binaires d'entiers
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```




```
          Noeud 20
     Noeud 10     Noeud 10
Feuille Feuille Feuille Feuille
```

```
val a1= Noeud(10,Feuille,Feuille)
val a2= Noeud(20,a1,a1)
```


Case classes et le pattern matching par `match-case`

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```

- `match-case` permet de décomposer les objets issus de case classes

```
def sommetPositif(e:IntA):Boolean= {
  e match {
    case Feuille => false
    case Noeud(e,_,_) => e>0
  }
}
```

Exercice 5 (Définissez la fonction (récursive) suivante)

`dans(i:Int,a:IntA):Boolean` *qui teste si un entier est dans un arbre.*

Case classes et le pattern matching par `match-case` (II)

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
```

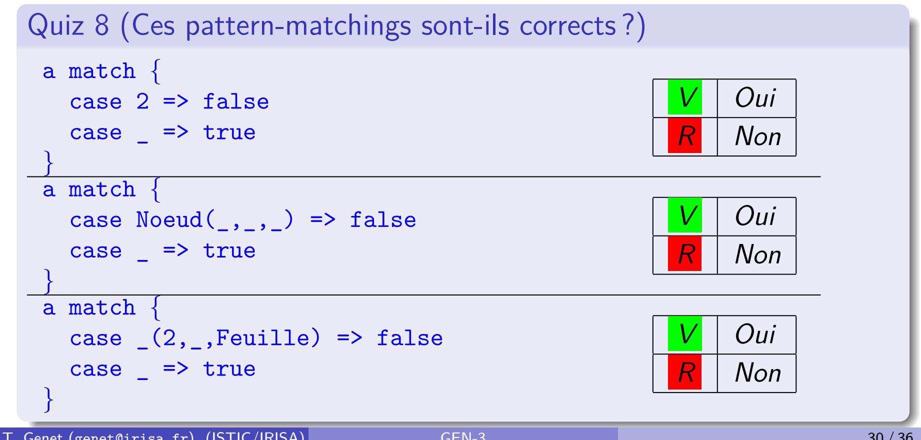
- `match-case` peut plonger dans plusieurs niveaux de case classes

```
def filsPositif(e:IntA):Boolean= {
  e match {
    case Noeud(_,Noeud(e,_,_),Feuille) => e>0 //fils gauche
    case Noeud(_,Feuille,Noeud(e,_,_)) => e>0 //fils droit
    case Noeud(_,Noeud(e1,_,_),Noeud(e2,_,_)) => e1>0 || e2>0
    case _ => false
  }
}
```


Case classes et le pattern matching par `match-case` (III)

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA

val a= Noeud(2,Noeud(1,Feuille,Feuille),Feuille)
```



Quiz 8 (Ces pattern-matchings sont-ils corrects ?)

```
a match {
  case 2 => false
  case _ => true
}
```

V	Oui
R	Non

```
a match {
  case Noeud(_,_,_) => false
  case _ => true
}
```

V	Oui
R	Non

```
a match {
  case _(2,_,Feuille) => false
  case _ => true
}
```

V	Oui
R	Non

Case classes et le pattern matching par `match-case` (IV)

```
sealed trait IntA
case class Noeud(i:Int,g:IntA,d:IntA) extends IntA
case object Feuille extends IntA
val a= Noeud(2,Noeud(1,Feuille,Feuille),Feuille)
```

Quiz 9 (Quels sont les résultats de ces pattern-matchings ?)

```
a match {
  case Noeud(_,_,Feuille) => 2
  case Noeud(2,_,_) => 3
}
```

V	2
R	3

```
a match {
  case Noeud(x,Noeud(y,Feuille,_),Feuille) => x+y
  case Noeud(x,Feuille,_) => x

}
```

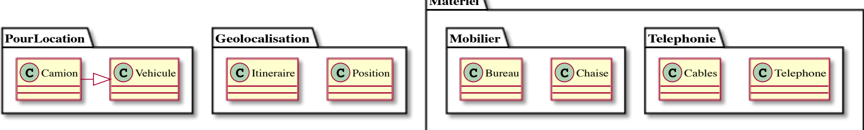
V	2
R	3

```
a match {
  case Noeud(_,x,_) => x
  case Noeud(_,_,x) => Feuille
}
```

V	*Noeud(1,Feuille,Feuille)*
R	*Feuille*

Packages

Pour structurer le code, on introduit des packages pour rassembler des classes/objets/traits ayant un lien logique.



Définition 10 (Notation pointée, les chemin d'accès pour les packages)

Comme pour les répertoires dans un système de fichiers, un package `p2` peut être inclus dans un package `p1`, etc. Dans ce cas, depuis le package `p1`, le chemin d'accès au package `p2` sera simplement : `p2`. Depuis le package *contenant* `p1`, le chemin d'accès sera `p1.p2`.

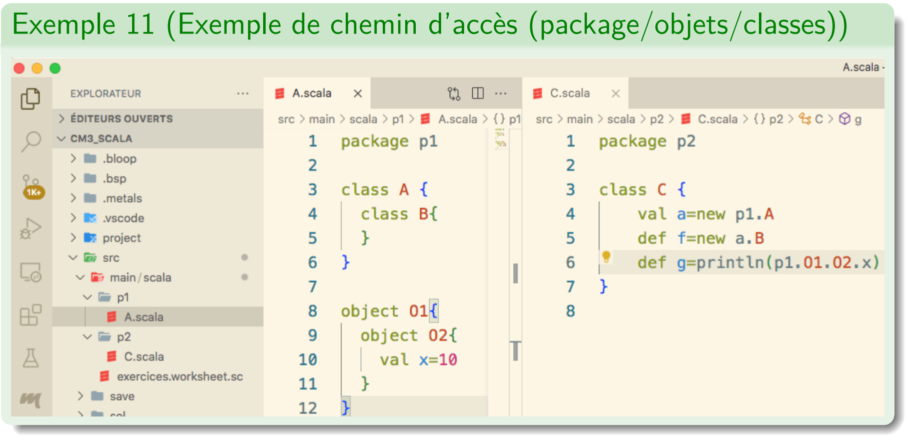
Remarque 10 (Définition du package d'un code Scala avec `package`)

Si `p` *est un chemin d'accès à un package, pour placer un code dans le package de chemin* `p`*, celui-ci doit débuter par* `package p`*.*

Packages et chemins d'accès aux objets/champs/méthodes

- On peut aussi inclure des classes/objets dans d'autres classes/objets
- Pour accéder à un objet/classe/champ/méthode `x` d'un objet `an` on utilise un chemin d'accès `a1.a2. ... an.x` (où `a1 a2 ... an` sont des noms de packages ou d'objets (pas de noms de classes !)

Exemple 11 (Exemple de chemin d'accès (package/objets/classes))



Packages et chemins d'accès (II)

Quiz 10 (Comment accéder au champ `x` ?)

```
package p0.p1.p2
object A{
  val x=10
}
```

```
package p0.p3

object C{
  val lx= ??? x
}
```

V	p1.p2.A.x	R	p0.p1.p2.A.x

```
package p0.p1
class A{
    val x=10
}
```

```
package p0.p2
object C{
val lx= ??? x
}
```

V	val lx= (new p0.p1.A).x)
R	val lx= p0.p1.A.x

Packages et importation de code

Les chemins sont simplifiés en important des classes/objets/champs/méthodes

Définition 12 (`import p.x`)

Importe la classe/objet/champ/méthode `x` de chemin d'accès `p`. Le chemin `p` peut contenir des noms de packages et d'objets mais pas de classes. Si `x` est '_', on importe tous les objets membres du package `p` ou tous les champs/méthodes de l'objet désigné par `p`.

- `import p.C` importe la classe/objet `C` du package de chemin `p`
- `import p._` importe toutes les classes/objets du package de chem. `p`
- `import p.O._` importe les champs/méthodes de l'objet `O` de chem. `p`

Remarque 11 (Classes/objets/méthodes importés par défaut)

```
import java.lang._     // tout le package java.lang
import scala._         // la librairie de base Scala: List,Set,...

import scala.Predef._ // des méthodes prédéfinies: println,...
```


Packages et importation de code (II)

Quiz 11 (Comment accéder au champ `x` ?)

```
package p0.p1.p2
object A{
  val x=10
}
```

```
package p0.p3
import p0.p1._
object C{
  val lx= ??? x
}
```

V	p1.p2.A.x	R	p2.A.x

```
package p0.p1
object A{
  object B{
    val x=10
  }
}
```

```
package p0.p2
import p0.p1.A._
object C{
  val lx= ??? x
}
```

V	x	R	B.x

Initiation au Génie Logiciel

Cours 4

–

Quelques outils pour le Génie Logiciel

Déroulement du TP345 par rapport au CMM ?

Capability Maturity Model = évaluation du processus de développement

Niveau 1, Initial/Chaotique/Heroïque

- Développement dans l'urgence, non reproductible
- Le succès du projet dépend d'un petit nombre d'individus
- Comportement du logiciel non prédictible (partiellement testé)

Pour réussir le second projet, vous **devez** passer le Niveau 2 :

Niveau 2, Reproductible/Discipliné

- Assurance qualité du logiciel (**test systématique** : CM4) comportement du logiciel prédictible, défauts détectés
- Sauvegarde, partage, restauration projet (**gestion de version** : CM4) versionnement du code, documentation, tests
- Tâches définies pour les développeurs, qui fait quoi ? (CM6)

Il reste 3 niveaux au dessus ☺ : défini, maîtrisé, optimisant

Plan

1. Gestion de version
2. Tests unitaires
3. Couverture et pertinence de tests
4. Génération de documentation

Plan

1. Gestion de version
2. Tests unitaires
3. Couverture et pertinence de tests
4. Génération de documentation

Pourquoi la gestion de version ?

1. Développer collaborativement un code commun, en évitant

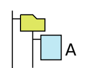
2. Sauvegarder le code, pour qu'il survive à

3. Revenir à une version antérieure du code, si la dernière version fait

Gestion de versions... quel est le problème ?

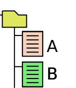
Il était une fois Jo et Mo qui travaillaient sur le même projet...

- Ils copient sur leur machine le projet qui ressemble à ça :



- Mo modifie le fichier A :
- Jo ajoute un fichier B :

- Comment font ils pour obtenir chacun une copie de ?



- Combien de copies entre Jo et Mo sont nécessaires (au min.) ?
  ... et s'il y avait eu N développeurs et N modifications ?

Gestion de versions... ça se complique, Bo est une truffe !

Bo s'est trompé et souhaite annuler sa modification de A

- A partir de
- Comment font ils pour obtenir chacun une copie de ?

- Combien de copies entre Jo et Mo sont nécessaires (au min.) ?
- ... et pour N développeurs ?

Remarque 1 (Ca n'est possible que si au moins un développeur...)

- *garde un historique des projets qu'il copie*
- *sait quels sont les fichiers/répertoires à ajouter/supprimer d'une version à l'autre*

C'est précisément ce que fait un gestionnaire de versions, y compris sur les fichiers eux-mêmes : (suppression, ajout, modification de ligne)

Gestion de versions avec Git, le principe

- Chaque développeur à un **dépôt Git local** qui enregistre toutes les modifications opérées sur **son** projet (toutes les versions)
- Les développeurs qui le souhaitent peuvent synchroniser leur **dépôt local** avec un **dépôt partagé** (sur le GitLab ISTIC pour nous)
- Les développeurs interagissent avec Git avec les commandes :
  - créer le dépôt local : `init`
  - signaler des modifications/ajouts sur des fichiers/répertoires : `add`
  - supprimer un fichier : `rm` , supprimer un répertoire : `rm -r`
  - propager les modif./ajouts/suppressions sur le dépôt Git local : `commit`
  - propager les modif./ajouts/suppr. du Git local vers le GitLab : `push`
  - obtenir une **copie locale** d'un dépôt GitLab : `clone`
  - obtenir les dernières modifications faites sur le GitLab : `pull`

Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo crée un nouveau projet (vide) sur GitLab

Jo

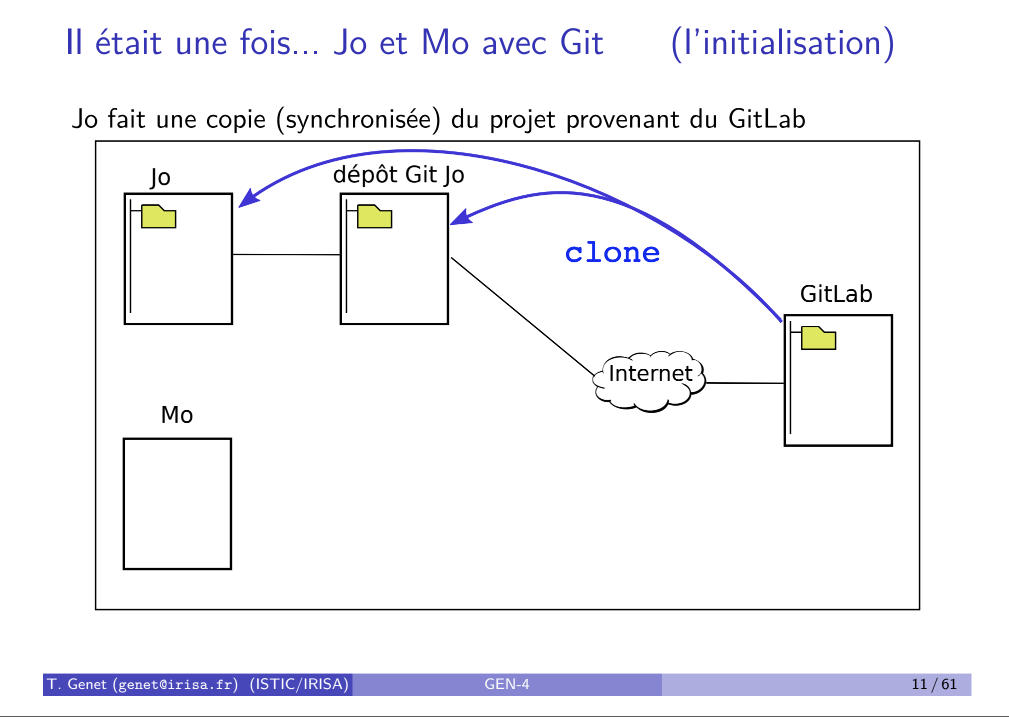
Mo

Internet

GitLab

Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo fait une copie (synchronisée) du projet provenant du GitLab



Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo ajoute un nouveau fichier A dans sa copie du projet

Jo

dépôt Git Jo

A

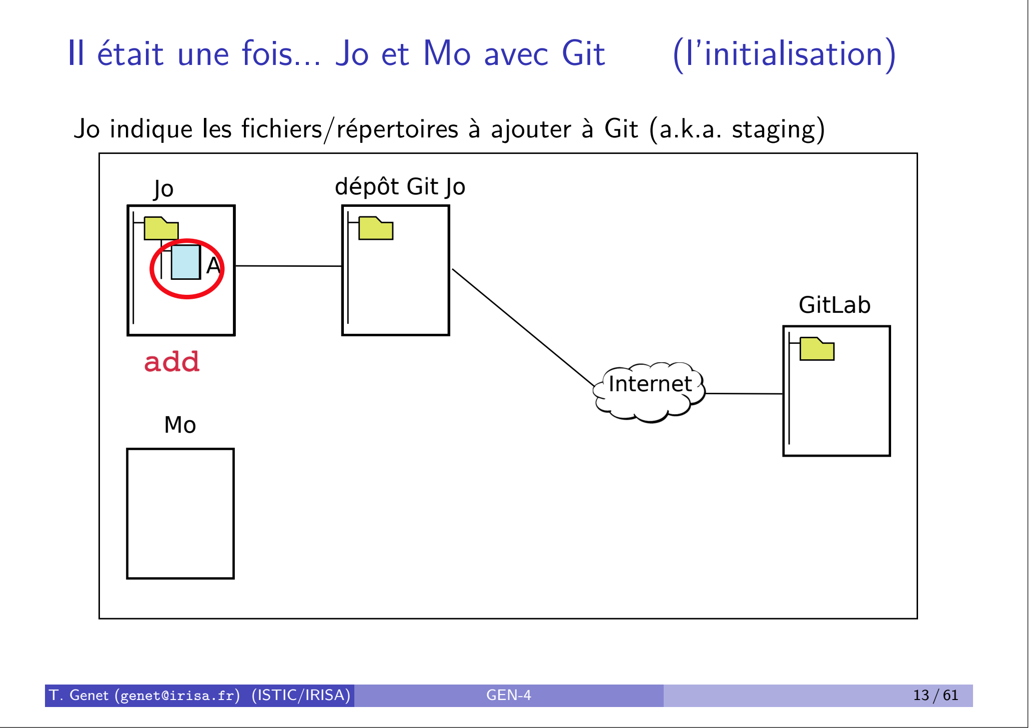
GitLab

Internet

Mo

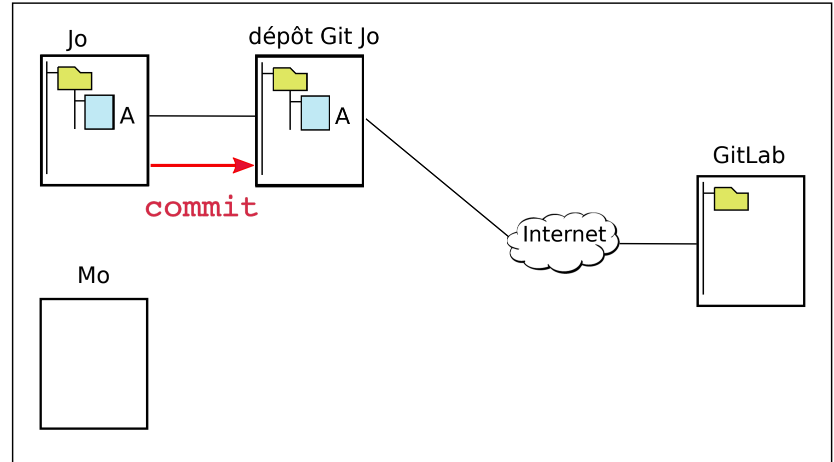
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo indique les fichiers/répertoires à ajouter à Git (a.k.a. staging)



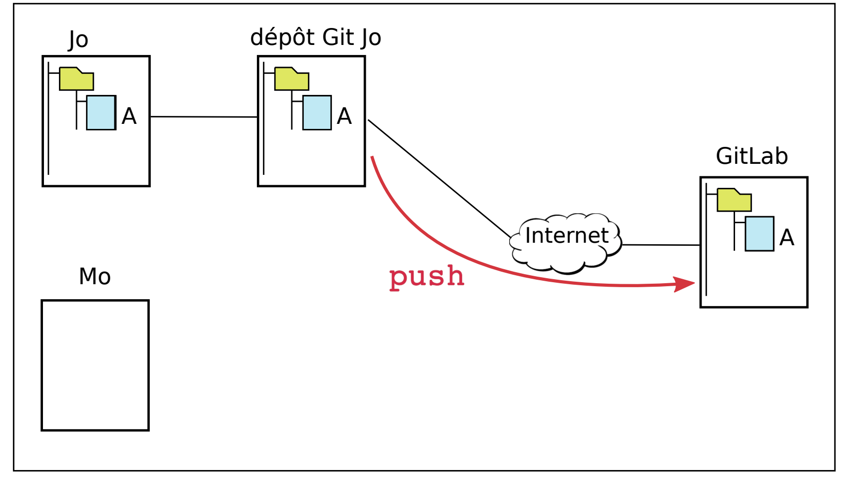
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo envoie les nouveaux fichiers/répertoires vers **son** dépôt Git



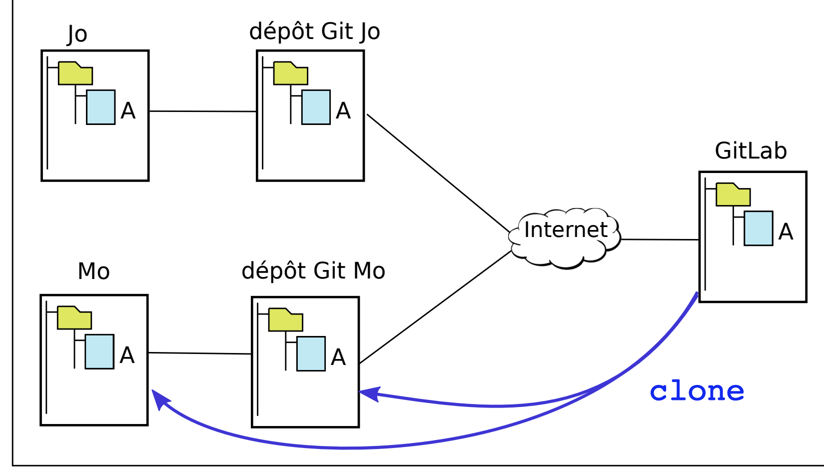
Il était une fois... Jo et Mo avec Git (l'initialisation)

Jo pousse ses modifications vers le GitLab



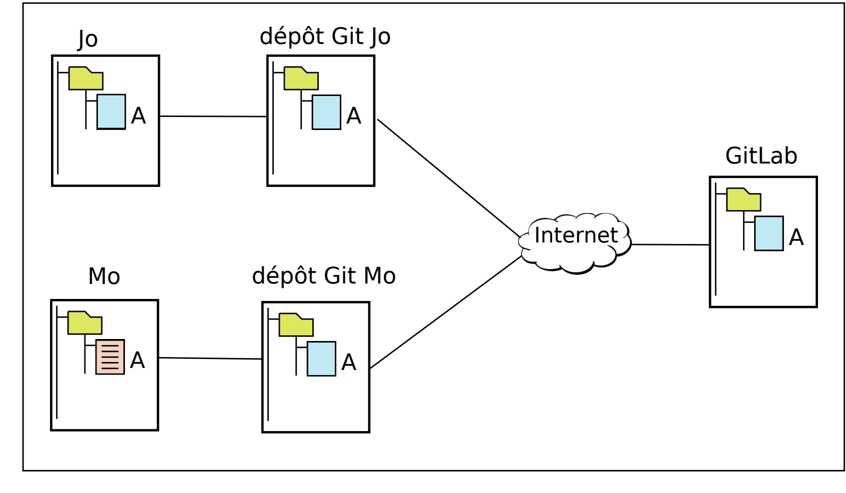
Il était une fois... Jo et Mo avec Git (l'initialisation)

Mo fait une copie (synchronisée) du dépôt sur le GitLab



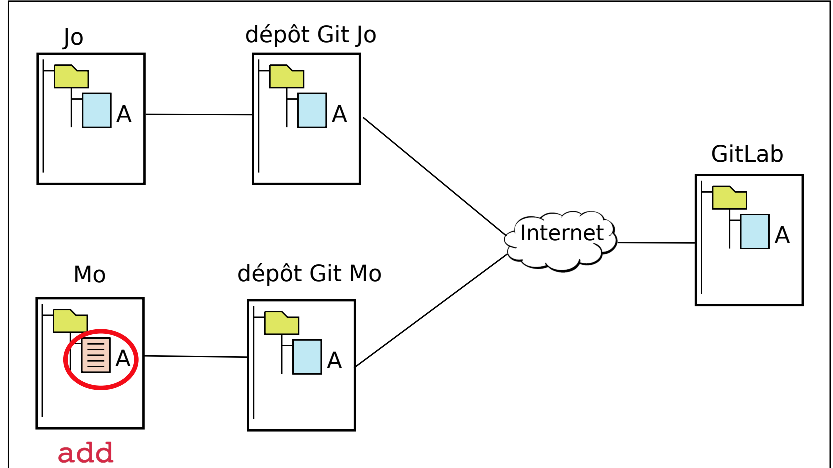
Il était une fois... Jo et Mo avec Git (utilisation)

Si Mo modifie le fichier A et veut envoyer la modification vers GitLab



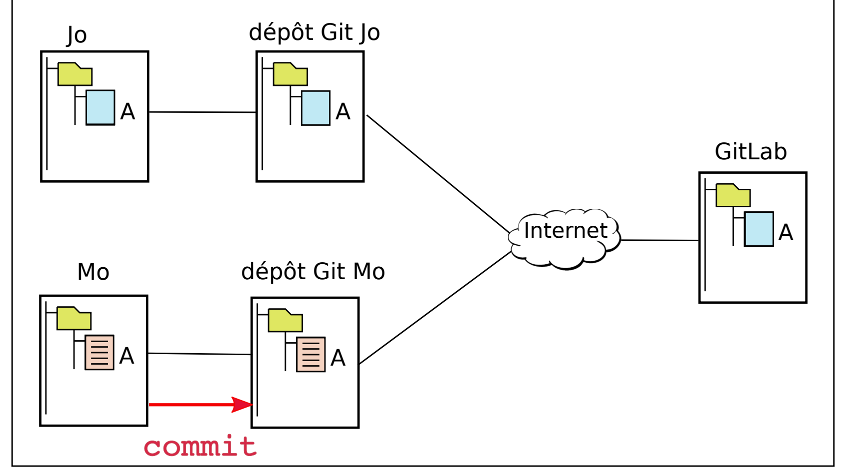
Il était une fois... Jo et Mo avec Git (utilisation)

Mo indique quels sont les fichiers à propager vers Git (a.k.a. staging)



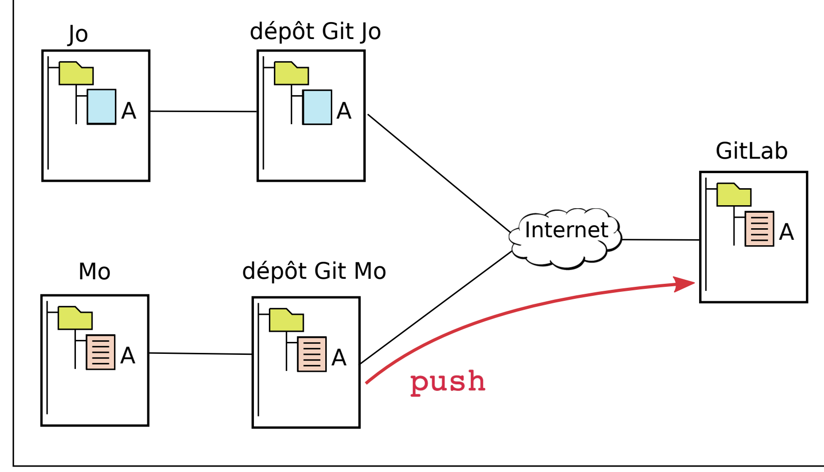
Il était une fois... Jo et Mo avec Git (utilisation)

Mo envoie les fichiers/répertoires modifiés vers **son** dépôt Git



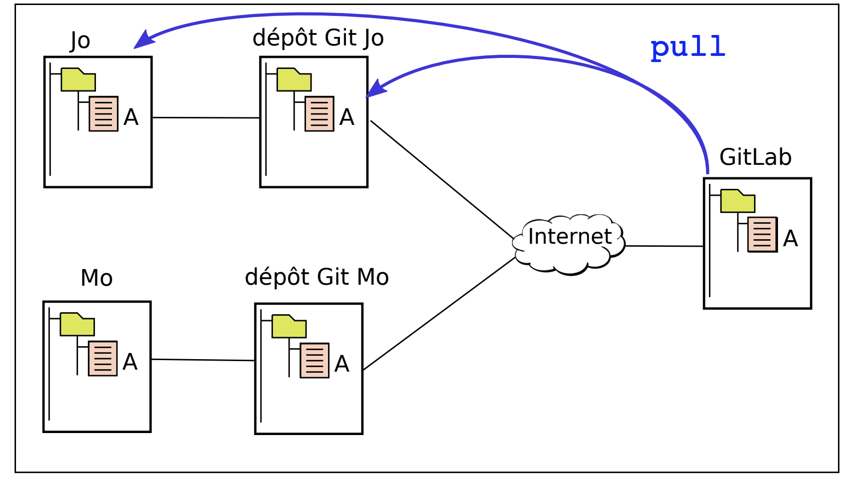
Il était une fois... Jo et Mo avec Git (utilisation)

Mo pousse ses modifications vers le GitLab



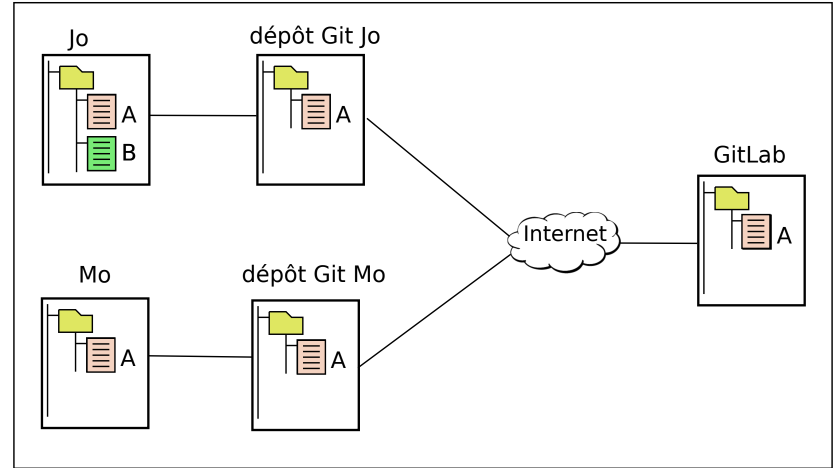
Il était une fois... Jo et Mo avec Git (utilisation)

Jo tire/applique les modifications du GitLab vers son dépôt et répertoire



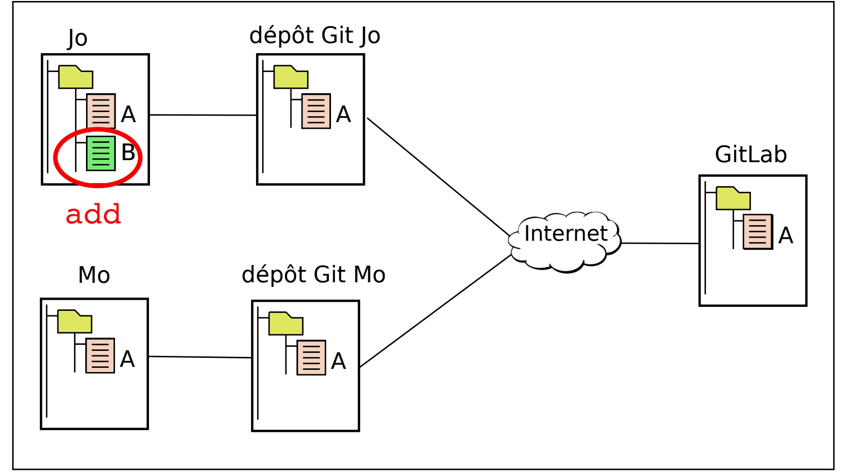
Il était une fois... Jo et Mo avec Git (utilisation)

Si Jo ajoute un fichier B et veut envoyer la modification vers GitLab



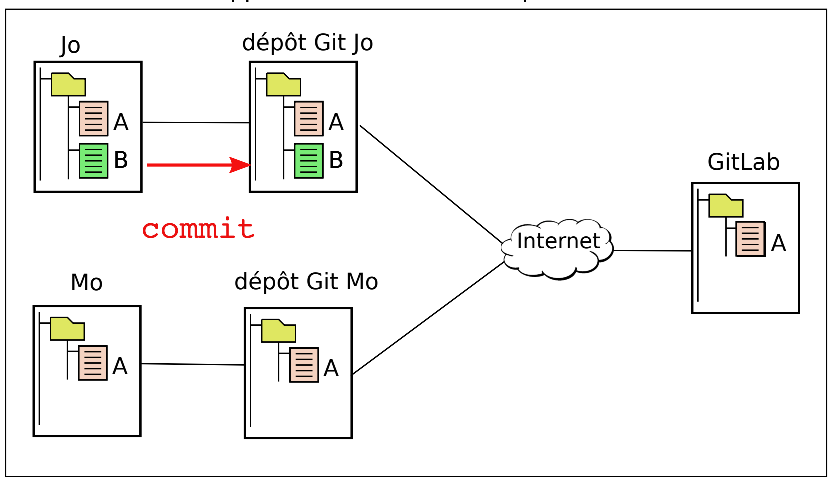
Il était une fois... Jo et Mo avec Git (utilisation)

Jo indique quels sont les fichiers à propager vers Git (a.k.a. staging)



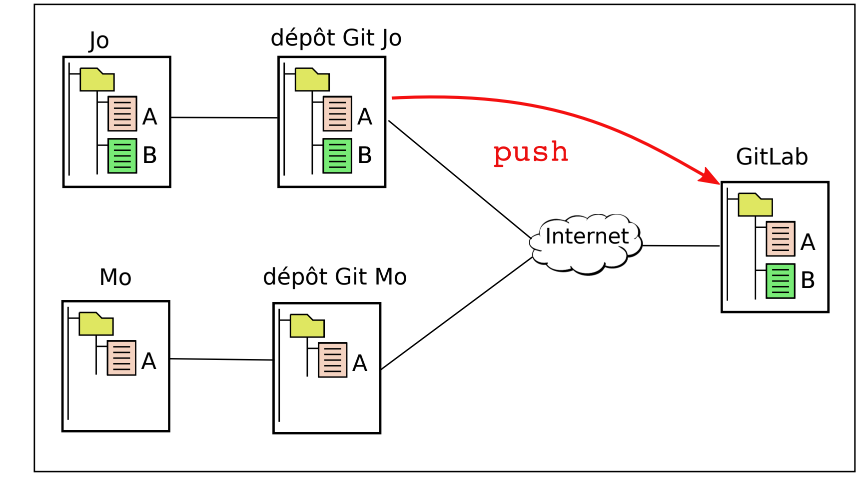
Il était une fois... Jo et Mo avec Git (utilisation)

Jo envoie le fichier supplémentaire vers **son** dépôt Git



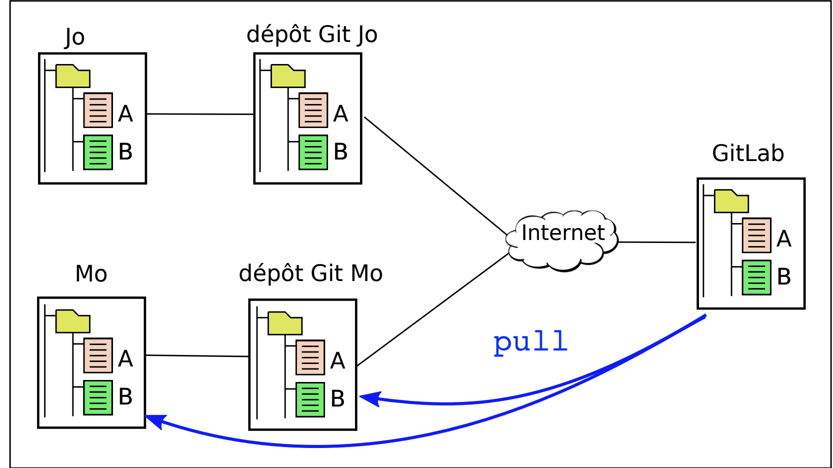
Il était une fois... Jo et Mo avec Git (utilisation)

Jo pousse ses modifications vers le GitLab



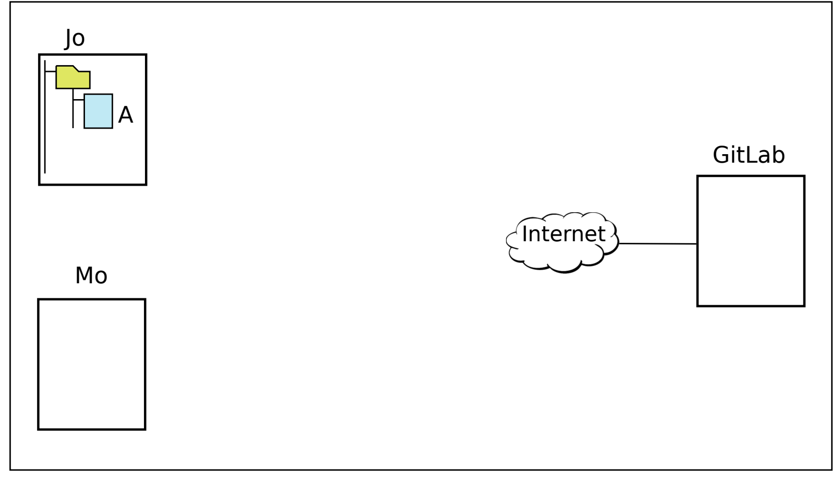
Il était une fois... Jo et Mo avec Git (utilisation)

Mo tire/applique les modifications du GitLab vers son dépôt et répertoire



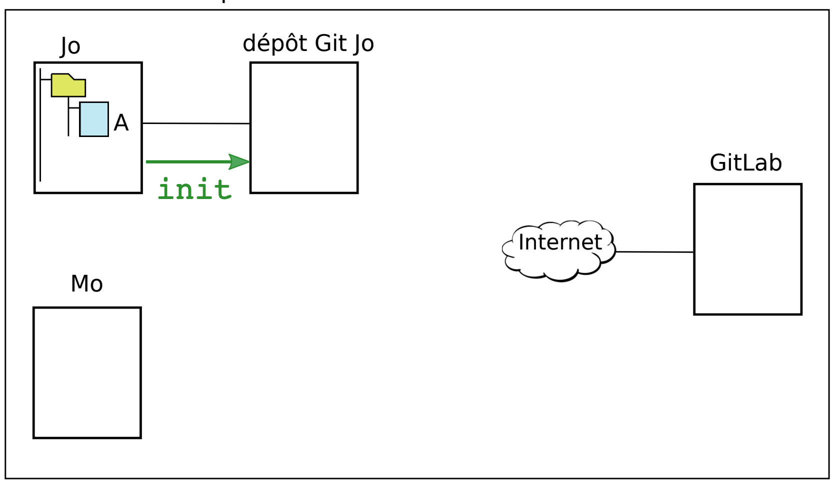
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo souhaite partager le projet contenu dans son répertoire



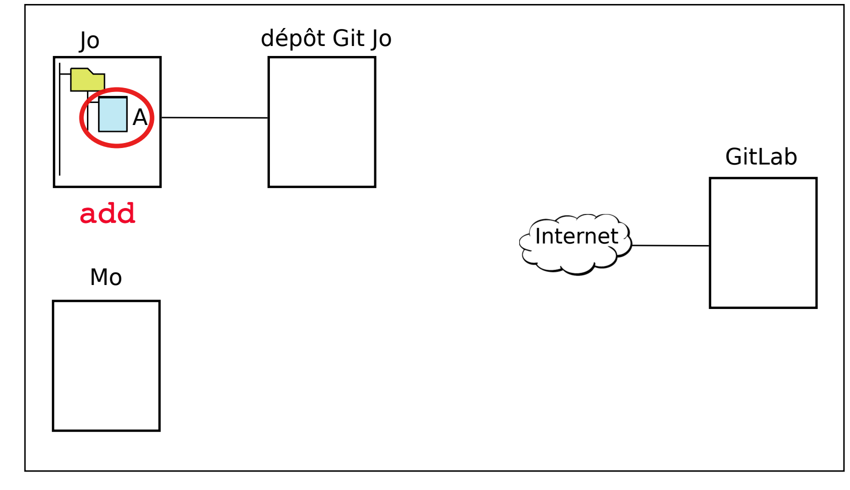
Jo et Mo avec Git (l'initialisation, une autre méthode)

L'initialisation du dépôt de Jo



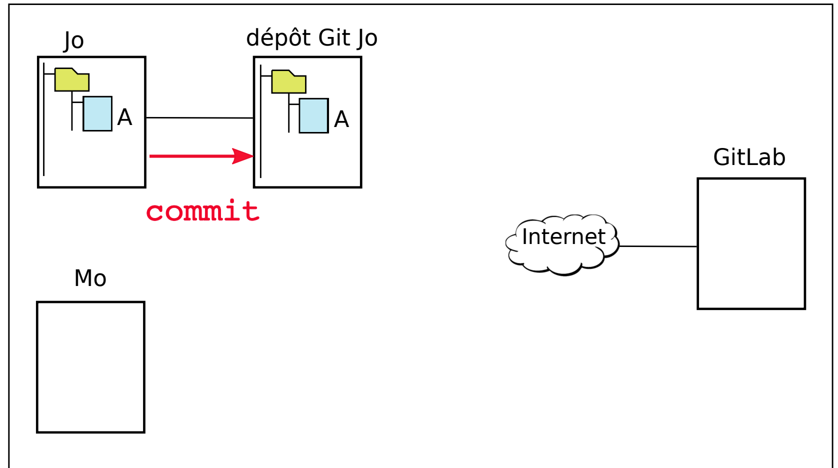
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo indique quels sont les fichiers/répertoires à ajouter Git (a.k.a. staging)



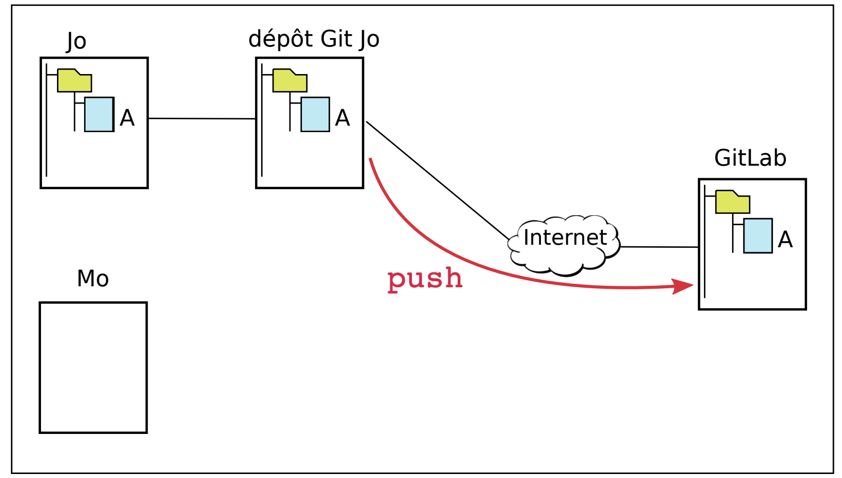
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo envoie les nouveaux fichiers/répertoires vers **son** dépôt Git



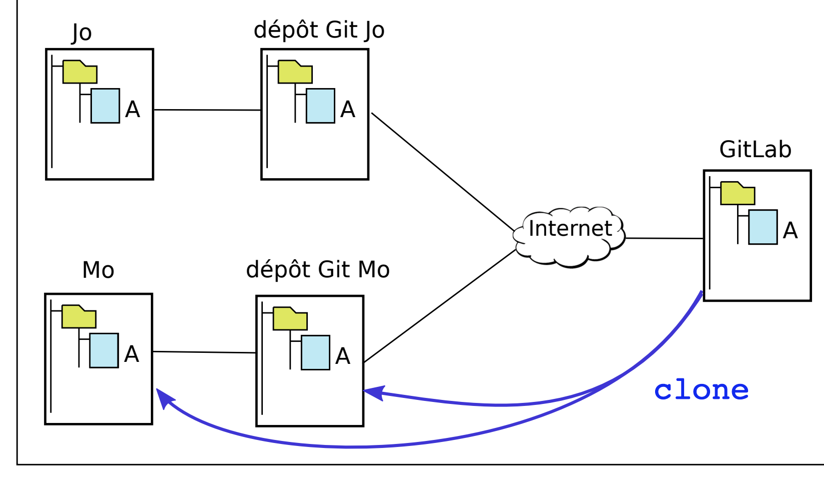
Jo et Mo avec Git (l'initialisation, une autre méthode)

Jo pousse ses modifications vers le GitLab



Jo et Mo avec Git (l'initialisation, une autre méthode)

Mo fait une copie (synchronisée) du dépôt sur le GitLab



Jo, Mo et Git, en vrai...

- Git s'utilise dans un terminal à l'aide de la commande `git`
- Il existe une multitude de clients graphiques pour Git...
- En TP nous utiliserons : Git en ligne de commande

Remarque 2 (Le GitLab ISTIC)

- `https://gitlab.istic.univ-rennes1.fr/`
- *Connectez-vous (au moins une fois) sur le GitLab istic avant le prochain TP !*
- *Créez un projet et tentez d'y déposer des fichiers*
- *Le code de vos projets GitLab sera consultables par un navigateur (Démo)*

Jo, Mo et Git, dans le terminal (initialisation)

Exemple 1 (Jo clone un projet gen1 auquel il a accès par le GitLab)

```
% git clone https://gitlab.istic.univ-rennes1.fr/jo/gen1.git
% cd gen1      # le répertoire de travail gen1 a été crée par git
```

Exemple 2 (Jo ajoute le fichier `lettre.txt` au projet (staging))

```
% git add lettre.txt
```

Exemple 3 (Jo envoie les modifications vers son dépôt Git local)

```
% git commit -m "ajout d'une lettre pour le père noël"
```

Exemple 4 (Jo pousse ses modifications vers le GitLab)

```
% git push
```

Exemple 5 (Mo clone le projet gen1 auquel il a accès par le GitLab)

```
% git clone https://gitlab.istic.univ-rennes1.fr/jo/gen1.git
% cd gen1
```


Jo, Mo et Git, dans le terminal (utilisation)

On suppose que le répertoire de travail est `gen1` (on a fait `cd gen1`)

Exemple 6 (Mo signale une modification sur `lettre.txt` (staging))

```
% git add lettre.txt
```

Exemple 7 (Mo envoie les modifications vers son dépôt Git local)

```
% git commit -m "ajout de remerciements"
```

Exemple 8 (Mo pousse ses modifications vers le GitLab)

```
% git push
```

Exemple 9 (Jo obtient les modifications faites sur le GitLab)

```
% git pull
```

Exercice 1 (Donnez-moi un Jo et un Mo !)

Corrigez la faute d'orthographe dans le fichier `temp1.txt` *de Mo. Envoyez la correction au GitLab et vérifiez que Jo peut réccupérer la modification.*

Jo, Mo et Git, en vrai... dans le terminal (III)

Quiz 1 (Cette suite de commande échoue ? V Oui R Non)

```
% git add monFichier
% git commit -m "modifications sur monFichier"
% git add monFichier
% git commit -m "on commite sans avoir modifié monFichier"
```

Quiz 2 (Cette suite de commande détruit définitivement `monFichier` ?)

```
% rm monFichier
% git commit -m "Suppression de monFichier"
% git push
```

V Oui / R Non

Quiz 3 (Cette suite de commande détruit définitivement `monFichier` ?)

```
% git rm monFichier
% git commit -m "Suppression de monFichier"
% git push
```

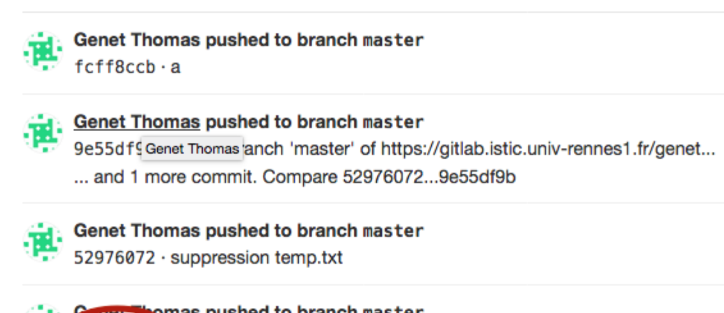
V Oui / R Non

Dans "gestionnaire de version", il y a le mot "version"

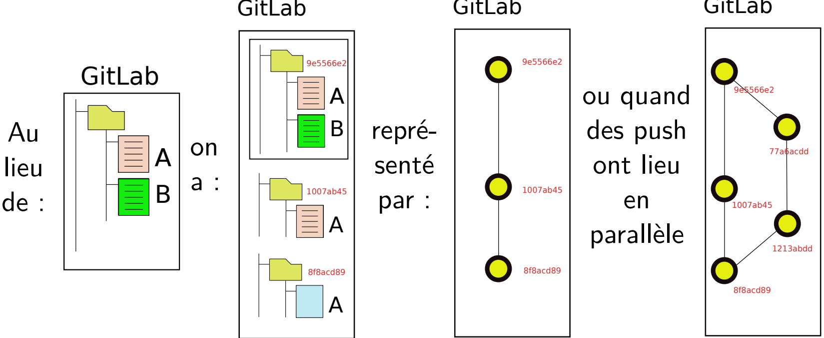
Remarque 3 (En Git chaque version a un identifiant unique)

Git conserve l'historique complet de toutes les versions du projet.

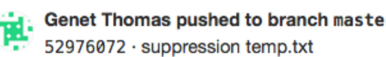
Dans GitLab, l'onglet "Activity" donne les identifiants des versions

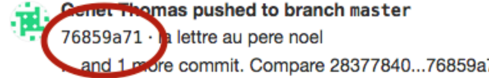


Dans "gestionnaire de version", il y a le mot "version"



Dans "gestionnaire de version", il y a le mot "version"





Exemple 10 (Jo replace tout son projet dans la version 76859a71)

```
% git reset --hard 76859a71
```

Attention ! on perd les modifications du projet non "commitées" dans Git !

Exemple 11 (Jo restaure le fichier temp1.txt de la version courante)

```
% git checkout temp1.txt
```

Attention ! on perd les modifications sur `temp1.txt` non "commitées" dans Git !

Dans "gestionnaire de version", il y a le mot "version"

Exercice 2

1. *Comment récupérer un fichier détruit dans le répertoire local ?*
2. *Comment récupérer un fichier supprimé de la dernière version du GitLab ?*

Quiz 4

Initialement, on suppose que le projet est dans la version 52976889 *et que l'on tape :*

```
% rm temp1.txt
% git reset --hard 76859a71
```

Comment remettre le projet dans sa forme initiale ?

% git reset --hard 52976889	% git checkout temp1.txt
V	R

La vie n'est pas un long fleuve tranquille : les conflits

Définition 12 (Conflit de version)

Si deux développeurs font des modifications contradictoires sur un même fichier, un conflit peut apparaître. Il touchera le dernier des deux faisant un push. Ce dernier sera le seul à pouvoir résoudre le conflit.

En pratique, les conflits de versions sont très rares, mais... (voir le titre)

Exemple 13 (Exemple de conflit de version)

Jo et Mo modifient différemment la ligne *i* du fichier B de la dernière version. Jo fait son push. Ensuite, Mo tente de faire son push. Celui-ci échoue car sa version n'est plus à jour. Mo fait un pull qui révèle un conflit sur la ligne *i* du fichier B.

Résolution de conflit Git sur un fichier B

1. Rechercher les annotations de conflits dans le texte du fichier B :

```
Le texte précédant la partie de texte en conflit.
<<<<<<< HEAD
C'est le texte que je voulais proposer.
=======
C'est le texte proposé par l'autre développeur.
>>>>>>> e4e1e956f6426c04ab80487f612265fe5c95ece8
La suite du texte qui n'est pas en conflit.
```

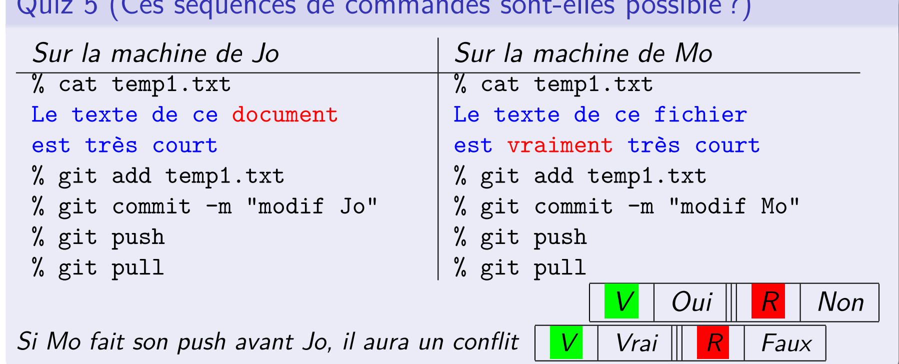
2. Résoudre le conflit dans le texte du fichier B :

```
Le texte précédant la partie de texte en conflit.
C'est le texte que je voulais proposer complété par le
texte proposé par l'autre développeur.
La suite du texte qui n'est pas en conflit.
```

3. Informer le dépôt de la résolution du conflit sur B :

```
% git add B
% git commit -m "Conflit sur 2eme ligne résolu"
% git push
```


Résolution de conflit Git sur un fichier B (II)



Exercice 3 (Jo et Mo modifient en même temps la ligne 4 du fichier `temp1.txt`)

Mo "push" sa modification. Que doit faire Jo pour envoyer, malgré tout, sa modification sur le dépôt sans perdre celle de Mo ?

Plan

1 Gestion de version

2 Tests unitaires

3 Couverture et pertinence de tests

4 Génération de documentation

Tester un logiciel pour quoi faire ?

Loi de Murphy : ≪ Le seul programme garanti sans bug est celui qui ne comporte aucune instruction ≫

Le test permet d'avoir des garanties sur la fiabilité de votre programme

- Vous n'aurez des garanties que sur ce qui a été effectivement testé !
- Que faut-il tester, comment choisir les tests ?
- Comment savoir si un programme est suffisamment testé ?

$\Longrightarrow$ Stratégies de tests + stratégie de sélection des tests

Remarque 4 (Incomplétude du test)

En général, il est impossible de tester complètement un programme car il possède un grand nombre (voire une infinité) de comportements possibles.

Par contre, on peut prouver complètement les programmes : voir cours de vérification formelle en L3 info et Master 1 info IL.

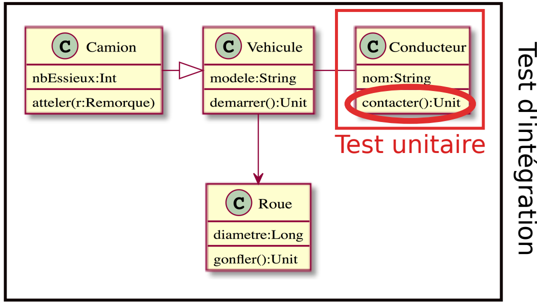
Stratégie de test : test unitaire, puis test d'intégration

Définition 14 (Test unitaire)

Un test unitaire porte sur une méthode d'une classe en isolation par rapport au reste de l'application.

Définition 15 (Test d'intégration)

Un test d'intégration porte sur l'intégralité de l'application. Il peut faire intervenir plusieurs classes de l'application.



Mise en oeuvre des tests unitaires/intégration avec JUnit 4

Une classe de test JUnit 4 (Java/Scala) peut comporter :

- Des définitions de variables (réinitialisées avant chaque test)
- Des tests : @Test

```
import org.junit.Assert._
import org.junit.Test
import D.Graphe

class TestGraphe{
// Init de l'env. de test
// Fait avant chaque test
  val g= new Graphe
  g.ajouteArc("A","B")
  g.ajouteArc("B","C")
```

```
// Un test
  @Test
  def testObjectCreation{
     assertNotNull(g)
  }

// Un autre test
  @Test
  def testEnsNoeuds{
     assertEquals(g.ensNoeuds,
                  Set("A","B","C"))
  }
}
```


Mise en oeuvre des tests unitaires avec JUnit 4 (II)

Les principales instructions de test sont :

```
assertNull(x:Any):Unit              assertNotNull(x:Any):Unit
assertEquals(x:Any,y:Any):Unit      assertTrue(x:Boolean):Unit
assertFalse(x:Boolean):Unit         ...
```

Remarque 5 (Créer une classe de test dans VisualStudio+SBT)

- *Vos classes de test sont à placer dans le répertoire* `src/test/scala/`
- *Le découpage en packages doit être identique à* `src/main/scala/`

Par exemple, pour tester un fichier `src/main/scala/p1/classeA.scala`
Placez votre fichier de test dans `src/test/scala/p1/`

Remarque 6 (Lancer les tests JUnit – voir tutoriel vidéo, site du cours)

- *Lancez vos tests à l'aide de* [icône], *ou*
- *Lancez vos tests à l'aide de* `sbt test`

Mise en oeuvre des tests unitaires avec JUnit 4 (III)

Quiz 6 (Le test suivant est-il réussi ?)

```
import org.junit.Assert._
import org.junit.Test

class Test1 {
    var s1=Set(1,3)
```

```
   @Test
   def test1={
     s1= s1.union(Set(2))
     assertEquals(s1,Set(1,2,3))
   }
}
```

V	*Oui*	R	*Non*

Quiz 7 (Qu'affiche le code suivant ?)

```
import org.junit.Assert._
import org.junit.Test

class Test1 {
    print("0")
```

```
   @Test
   def test1={print("1")}

   @Test
   def test2={print("2")}
}
```

V	0102	R	012

Mise en oeuvre des tests unitaires avec JUnit 4 (IV)

Quiz 8 (Les tests suivants sont-il tous réussis ?)

Le code de la classe `A` *:*

```
package cm4
class A(x:Int){ var c=x }
```

```
import org.junit.Assert._
import org.junit.Test
import cm4.A

class TestA {
   val a= new A(0)
```

```
  @Test
  def test1{
    assertEquals(0,a.c)
    a.c=10
  }

  @Test
  def test2{
    assertEquals(10,a.c)
  }
}
```

V	Oui	R	Non

Tests unitaires pour assurer la non-régression

JUnit est un outil de tests unitaires automatisés :

- Chaque test est écrit une seule fois, mais...
- Il peut-être relancé à volonté (après toute modification)
- Les tests sont sauvegardés/versionnés sur le GitLab du projet

$\Longrightarrow$ Un test est plus précieux (et pérenne) qu'une ligne de code !

Définition 16 (Test de non-régression)

Un test de non-régression vérifie que l'ajout de nouvelles fonctionnalités dans un programme ne perturbe pas les fonctions initiales.

Remarque 7 (JUnit pour le test automatique de non-régression)

En préparant la version $n+1$ du logiciel, en vérifiant que tous les tests de la version n passent encore, on assure la non régression de la version $n+1$ par rapport à la version n.

Plan

1. Gestion de version
2. Tests unitaires
3. Couverture et pertinence de tests
4. Génération de documentation

Quels tests choisir ?

Le choix des tests à effectuer est un exercice difficile :

(a) Que dois-je tester ? — Demandez à l'utilisateur !

(b) Ai-je suffisamment testé ma fonction ? — Jamais !

Stratégie de sélection de tests

On utilisera 2 stratégies (complémentaires) de sélection de tests :

- par couverture des fonctionnalités, pour répondre à (a)
- par couverture du code, pour tenter de répondre à (b)

Exercice 4

Donnez des tests pour vérifier que `delete(x:Int,l:List[Int]):List[Int]` *supprime bien toutes les occurences de* `x` *dans* `l`.

Couverture des fonctionnalités par les tests : TDD

TDD= Test Driven Development (Développement dirigé par les tests)

Principe du TDD

Pour ajouter une fonctionnalité X à un logiciel :

1. Ecrire les tests T montrant que X fonctionne
2. Vérifier que le test échoue (X n'est pas encore développée)
   $\Longrightarrow$ Permet de vérifier que le test est valide !
3. Ecrire le code nécessaire pour passer T (et pas plus)
4. Vérifier que les tests T passent, sinon retourner en 3
5. Si nécessaire, remanier le code pour le simplifier

Couverture des fonctionnalités par les tests : TDD (II)

Exercice 5

Développer une fonction `add(s:String):Int` *qui additionne les nombres contenus dans une chaîne de caractères. Fonctionnalités :*

1. *pour une chaîne vide, la fonction rend 0*
2. *pour une chaîne contenant un seul nombre, la fonction rend ce nombre*
3. *pour une chaîne contenant deux nombres séparés par une virgule, la fonction rend la somme*
4. *étendre la fonction pour traiter tous les nombres présents dans la chaîne*
5. ...

Couverture du code par les tests : les métriques

Définition 17 (Tests boîte noire/tests boîte blanche)

Le test "boîte noire" vérifie que les sorties d'un programme correspondent à la spécification. Dans le test "boîte blanche", c'est identique mais l'on dispose du code du programme pour sélectionner les tests à effectuer.

Il existe de nombreuses métriques pour les tests boîte blanche :

- % de couverture des instructions du programme
- % de couverture des branches, % de couverture des chemins, ...

Définition 18 (Métrique de couverture "instructions")

Un ensemble de tests couvre n% des instructions du programme si cet ensemble a exécuté, au moins une fois, n% des instructions du programme.

Remarque 8 (Les métriques sont juste une indication de robustesse)

Un programme dont 100% des instructions ont été testées peut encore comporter des bugs. Mais, il sera plus sûr que s'il avait été testé à 50%.

Plan

1. Gestion de version
2. Tests unitaires
3. Couverture et pertinence de tests
4. Génération de documentation

Génération de documentation : Scaladoc

Remarque 9 (Génération de la Scaladoc dans ScalaIDE)

Dans ScalaIDE, la génération de la documentation est automatique. Elle sera accessible simplement en passant le pointeur de la souris sur le nom de la classe/objet/champ/opération documenté.

Exemple 19 (La Scaladoc par défaut)

Dans le projet `CM4`, package `vehicules`, objet `Test`, consultez la scaladoc disponible pour `ajouter` et `vpossible`.

On peut compléter la Scaladoc en ajoutant dans le code des classes, objets, interfaces et membres des commentaires spécifiques `/** ... */`

Exemple 20 (Scaladoc pour du code annoté)

Dans le projet `CM4`, package `vehiculesAnnote`, objet `Test`, consultez la scaladoc disponible pour `ajouter` et `vpossible`.

Scaladoc : principales annotations

```
/** Les nombres rationnels
 *  @constructor Crée un nouveau rationnel à partir de son
                 numérateur et dénominateur
 *  @param n le numérateur
 *  @param d le dénominateur
 *  @throws DenominateurNul si le dénominateur est 0
 */
class Rational(n:Int,d:Int){
   if (d==0) throw DenominateurNul
   /** Le numérateur */
   val num=n
   /** Le dénominateur */
   val den=d
   /** Rend la somme de deux rationnels
    *  @param r le rationnel à ajouter
    *  @return la somme des deux rationnels
    */
   def +(r:Rational):Rational = {
     new Rational(this.num*r.den+this.den*r.num,this.den*r.den) }
```


Scaladoc : principales annotations (II)

Remarque 10 (Constructeurs auxiliaires documentés comme des méthodes)

```
/** Crée un nouveau rationnel à partir d'un entier. En utilisant
    ce constructeur, le dénominateur vaudra 1.
  *  @param x la valeur de l'entier */
   def this(x:Int)={this(x,1)}
```

Remarque 11 (Il n'est pas nécessaire de répéter la documentation des traits)

```
/** Les Transporteurs de passagers.*/
trait Transporteur{
  /** ajouter un passager au transporteur
   *  @param p le passager à ajouter */
  def ajouter(p: Passager):Unit
}

class Vehicule extends Transporteur
 // La scaladoc Vehicule contiendra celle de Transporteur
```


Conseils de rédaction Scaladoc : allez à l'essentiel

1. La 1ère phrase de la documentation classe/méthode/trait et les annotations doivent être synthétiques. Les détails seront dans le reste de la documentation de la classe/méthode/trait.

```
/** ~~Cette classe représente~~ Les véhicules routiers.
  * Les véhicules contiennent des références vers [...] */
  class Vehicule{ ... }
/** ~~Cette méthode~~ Rend le premier élément d'une liste.
* Si le premier élément n'existe pas [...]
* @return le premier élément de la liste ~~s'il existe, sinon
         rend [...]~~ */
def premier[T](l:List[T]):T={ ... }
```

2. Si une méthode rend un résultat XXX, commencez la première phrase de la documentation par "Rend XXX".
3. Si la documentation de la méthode se résume à une seule ligne, ne pas le répéter dans l'annotation `@return`

```
/** Teste si le rationnel est nul.
    ~~* @return true si le rationnel est égal à 0~~ */
  def isNull:Boolean=(num==0)
```

Initiation au Génie Logiciel

Cours 5

–

Concepts avancés de programmation fonctionnelle et de programmation objet

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Fonctions/méthodes d'ordre supérieur

Définition 1 (Fonction d'ordre supérieur)

Une fonction *d'ordre supérieur* prend en paramètre une ou plusieurs fonctions.

Exemple 2 (La fonction `map` sur les listes)

La fonction `map` applique une fonction (un traitement) à tous les éléments d'une liste.

```
val l= List(1,2,3)
def f(x:Int)= x+1
val l2= l.map (f)       // l2= List(2,3,4)

// ou plus simplement en utilisant l'application partielle sur +
val l3= l.map (_ + 1)   // l3= List(2,3,4)
```

Remarque 1 : A comparer avec la version impérative (avec for) !

Fonctions/méthodes d'ordre supérieur (II)

Exemple 3 (La fonction `filter` sur les listes)

La fonction `filter` filtre les éléments d'une liste en fonction d'un prédicat, *i.e.* une fonction dont le résultat est booléen. `filter` conserve les éléments pour lesquels le prédicat est vrai.

```
val l= List(1,2,3,4)
def f(x:Int)= x>2
val l2= l.filter (f)       // l2= List(3,4)

// ou plus simplement en utilisant l'application partielle sur >
val l3= l.filter (_ > 2)   // l3= List(3,4)
```

Nous verrons comment **définir** des fonctions d'ordre supérieur dans l'exercice 11.

Fonctions/méthodes d'ordre supérieur (III)

Exemple 4 (La fonction `reduce` sur les listes)

La fonction `reduce` réduit une liste d'éléments à une valeur à l'aide d'une fonction à 2 arguments. La liste doit comporter au moins 1 élément !

```
val l= List(1,2,3,4)
def f(x:Int,y:Int)= x+y
val i= l.reduce(f)        // i= 10

// ou plus simplement en utilisant l'application partielle sur +
val l3= l.reduce (_ + _)   // i= 10
```

Remarque 1 : toutes ces fonctions existent pour les `Set`, `Array`, `Map`, ...

Remarque 2 : il existe beaucoup d'autres fonctions d'ordre supérieur : `exists`, `forAll`, `foldLeft`, `foldRight`, ...

Fonctions/méthodes d'ordre supérieur (IV)

Exercice 1

A partir de l'ensemble d'entiers $\{1, 2, 3\}$, *construisez l'ensemble de chaînes* $\{"1", "2", "3"\}$.

Exercice 2

Définissez la fonction `remove(x:String,l:List[String]):List[String]` *qui supprime toutes les occurrences de l'élément* `x` *dans* `l`.

Exercice 3

Calculez la somme des éléments d'un tableau. Calculez la factorielle de 10.

Exercice 4

Calculez l'ensemble des carrés de l'ensemble `Set(1,2,3,4)`.

Fonctions anonymes

Notation de fonction classique	Lambda-notation
$f : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$	$f : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
$f(x, y) = x + y$	$f = \lambda x\ y.\ x + y$

$\lambda x\ y.\ x + y$ est une fonction anonyme ajoutant deux naturels

C'est le composant de base de ce qu'on appelle le "lambda-calcul"

- La fonction anonyme Scala ajoutant deux entiers s'écrit :
  `((x:Int,y:Int) => x + y)`
  Le type inféré par Scala pour cette fonction est : `(Int,Int) => Int`

Exercice 5

Refaire l'exercice 4 à l'aide de `map` *et d'une fonction anonyme.*

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Abstraction de type et polymorphisme

Quel est le problème ? Comment factoriser ce code ?

```
// Fonction retournant le premier élément d'un couple d'entiers
def first(p:(Int,Int)):Int=  p match case (x,y) => x
val e=  first((10,20))    // e=10
// Fonction retournant le premier élément d'un couple de chaînes
def first(p:(String,String)):String=  p match case (x,y) => x
```

On peut paramétrer les fonctions par des variables de type

```
// Fonction retournant le premier élément d'un couple
def first[T1,T2](p:(T1,T2)):T1=  p match case (x,y) => x
```

- pour appeler la fonction on doit donner les valeurs des types

```
val e=  first[String,String](("titi","toto"))// e="titi"
val e2= first[Int,String]((10,"toto"))       // e2=10
```

- ... mais ceux-ci peuvent généralement être inférés ! !

```
val e3= first((10,"toto"))                   // e3=10
```


Abstraction de type et polymorphisme (II)

Quel est le problème ? Comment factoriser ce code ?

```
trait IntQueue{//files d'entiers     trait StringQueue{//files
   def get:Int                           def get:String //de chaînes
   def put(x:Int):Unit}                  def put(x:String):Unit}
```

On peut paramétrer les classes/traits par des types

```
trait Queue[T]{  // Interface des files génériques
      def get:T
      def put(x:T):Unit}
                                        // Classe de files génériques
class MyQueue[T](init:List[T]) extends Queue[T]{
      private var b= init
      def get={val h=b(0); b=b.drop(1); h}
      def put(x:T):Unit= {b=b:+x}}

val f= new MyQueue[String](List())  // valeur de T non inferée
val f2= new MyQueue(List("toto"))   // valeur de T EST inferée
```


Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Quel est le problème avec les traits ?

Exemple 5 (Rappel : Trait pour les files d'entiers)

Utilisateurs	Interface	Implanteurs
Une équipe réalise un logiciel *utilisant* des `IntQueue`	`trait IntQueue { def get:Int def put(x:Int):Unit def empty:Boolean }`	Une équipe *implante* le trait `IntQueue` dans une classe `MyQueue`

```
def vider(q:IntQueue):Unit=
  while(!q.empty) q.get

val f:IntQueue= new MyQueue()

// On doit connaitre MyQueue
// pour obtenir un objet
// de type IntQueue!
```

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0)
            b=b.drop(1)
            h}
  def put(x:Int)= {b=b:+x}
  def empty=b.isEmpty
}
```


Plus d'abstraction avec les traits : les fabriques d'objets

Connaître le trait `X` ne permet pas de créer un objet de type `X`

- Il faut connaître une classe `C` implantant `X`
- Dans le code, `new C` apparaît partout où un nouvel objet de type `X` est nécessaire
- $\Longrightarrow$ On ne peut pas facilement changer d'implantation de `X` sans reprendre tout le code

Un patron de conception : les fabriques d'objets

- En complément du trait `X`, fournir une fabrique d'objets

```
object Xfactory{
  def get:X= new C
}
```

- Dans le reste du code, remplacer `new C` par `Xfactory.get`

Voir example 4 dans le projet CM5_Scala du cours

Plus d'abstraction avec les traits : les fabriques d'objets (II)

Exemple 6 (Changer d'implantation de `X` sans fabrique)

Dans le code, remplacer **toute** occurrence de `C` par `C2`

```
val q1= new C        | val q1= new C2
val q2= new C        | val q2= new C2
 ...                 |  ...
val qn= new C        | val qn= new C2
```

Exemple 7 (Changer d'implantation de `X` avec une fabrique)

Dans le code, remplacer **une** occurrence de `C` par `C2`

```
object Xfactory{        | object Xfactory{
  def get:X= new C}     |   def get:X= new C2}
                        |
val q1= Xfactory.get    | val q1= Xfactory.get  // ne change pas
val q2= Xfactory.get    | val q2= Xfactory.get
 ...                    |  ...
val qn= Xfactory.get    | val qn= Xfactory.get
```


Polymorphisme et fabriques

Un **multi-ensemble** est un ensemble où chaque valeur peut apparaître plusieurs fois.

Exercice 6

Définir le trait pour un multi-ensemble polymorphe (de valeurs de type `T`*). Donnez le type des opérations d'ajout d'un élément et de test d'appartenance qui donne le nombre d'occurrences de l'élément dans le multi-ensemble.*

Exercice 7

Définir une classe polymorphe implantant le trait multi-ensemble.

Exercice 8

Définir une fabrique pour le trait des multi-ensembles.

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Héritage et visibilité

Si la classe A hérite de B, tous les *membres* de B...

- déclarés `public` sont accessibles aux objets de classe A

```
class B {
  val x=0
  def f(x:Int)=x+1}
class A extends B {
  val y=18}
```

```
scala> val a= new A
scala> a.y
Int = 18
```

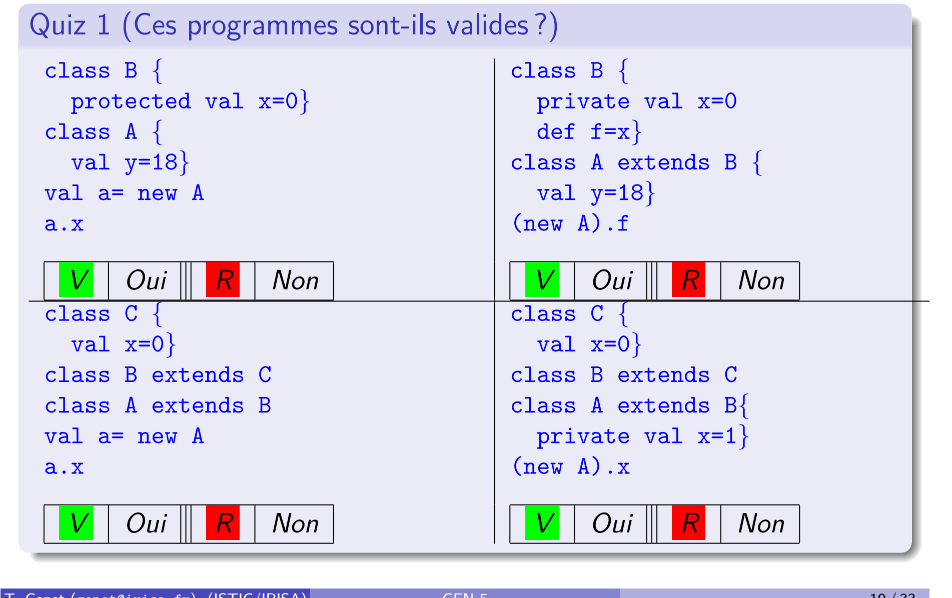
```
scala> a.x
Int = 0
scala> a.f(10)
Int = 11
```

- déclarés `private` sont invisibles pour tous les objets (y compris A)
- déclarés `protected` sont accessibles aux objets des classes héritant de B (dont A), **depuis le code de ces classes**.

Remarque 1 (Liaison dynamique et superclasse)

Si `o` est un objet de classe A, `o.x` fait référence au membre `x` de la classe A, s'il existe. Sinon, `x` est recherché dans la classe dont hérite A. Ce processus est itéré jusqu'à trouver une classe implantant `x`. Dans la classe A, `super.x` fait référence à l'implantation de `x` dans la classe dont hérite A. De la même façon, ceci est itéré jusqu'à trouver une classe implantant `x`.

Héritage et visibilité (II)



Héritage et redéfinition

Tous les champs/méthodes hérités peuvent être redéfinis avec `override`

Exemple 8 (Redéfinition de champs/méthodes avec `override`)

```
class B {
  val x=0
  def f(y:Int)=y+1
}
class A extends B{
  override val x=18
  override def f(z:Int)=20
}
```

```
scala> val a=new A
scala> a.x
Int = 18
scala> a.f(18)
Int = 20
```

Héritage et redéfinition (II)

Quiz 2 (Quels sont les résultats attendus pour ces programmes ?)

```
class B {
  def f(y:Int)=y+1
}
class A extends B {
  override def f(z:Int)=18
}
val a= new A
val b= new B
println(a.f(10)+b.f(10))
```

V	29	R	36

```
class B {
  val x=0
  def f(y:Int)=x
}
class A extends B {
  override val x=18
}
val a= new A
println(a.f(10))
```

V	18	R	0

Quiz 3 (Ce programme rend V 2 R "11" ?)

```
class C {
 def f(x:Int)=x+1
}
```

```
class B extends C{
 def f(x:String)=x+1
}
```

```
class A extends B{}
val a= new A
println(a.f(1))
```


Héritage et redéfinition... le cas de toString et equals

Tout objet hérite de la classe `Any` de Scala qui définit un certain nombre de méthodes par défaut : `toString`, `equals`, `hashCode`, ...

Pourquoi redéfinir toString ?

Par défaut, `toString` affiche la référence de l'objet.

Exemple 9

Dans la classe `Rational` du CM1, on a redéfinit `toString` (CM5.scala).

Pourquoi redéfinir equals dans les classes que **vous** définissez ?

Par défaut, `equals` (et donc `==`) compare les références sur les objets.

Exemple 10

A partir du code de la classe `Rational`, construire des ensembles de rationnels. Que se passe-t-il si on ajoute des objets rationnels différents mais avec de même valeur ?

Redéfinition de equals

Comment redéfinir equals, == et != ?

- Toute égalité de 2 objets Scala (`==` et `!=`) est réalisée par `equals`
- Par défaut, `equals` compare les références des objets (sauf `case class` où il compare les valeurs de tous les champs du constructeur)
- La méthode à redéfinir est `equals(o:Any):Boolean`

```
class Rational(x:Int,y:Int){ val n=x; val d=y ; ...
   override def equals(o:Any)=
     o match {
        case e:Rational => e.n==this.n && e.d==this.d
        case _ => false }}
```

Exercice 9

Redéfinissez `equals` dans `Rational` et observez l'impact sur les ensembles de nombres rationnels.

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. Exceptions

Match-case revisité pour les List

- Toute liste peut être construite à partir de Nil (liste vide) et ::

```
scala> val l=1::(2::(3::Nil))  // ou simplement 1::2::3::Nil
List[Int] = List(1, 2, 3)
```

- On peut aussi effectuer du pattern matching sur n'importe quelle liste à l'aide de :: et Nil

```
l match {
  case Nil => false
  case e::r => (e==1)
}
```

Dans le pattern matching, e=1 et r=2::3::Nil, car le filtrage s'opère de la façon suivante :

1	::	(2::(3::Nil))
e	::	r

Match-case revisité pour les List (II)

Quiz 4 (Que vaut cette expression ?)

```
val l=List()
l match {
  case Nil => false
  case e::r => true
}
```

V	false
R	true

Quiz 5 (Que vaut cette expression ?)

```
val l=List(1,2,3)
l match {
  case Nil => List(0)
  case _ :: Nil => List(1)
  case e::e2::e3 => e3
}
```

V	List(3)
R	List(1)

Exercice 10

Ecrivez un match-case *qui vérifie si une liste a exactement 4 éléments.*

Fonctions récursives

- On peut définir des fonctions *récursives* comme en SI2
- Pour les fonctions récursives, le type du résultat *doit* être donné
- Les fonctions récursives doivent terminer, par exemple : chaque appel récursif doit faire décroître un paramètre de la fonction

Exemple 11 (fonctions récursives)

```
def sum(i:Int):Int=    // La somme des i premiers entiers
  if (i<=0) 0 else i+sum(i-1)

def length(l: List[String]):Int=  // Longueur d'une liste
  l match {
    case Nil => 0
    case _::r => 1+length(r)
}
```


Fonctions récursives (II)

Quiz 6 (Cette fonction termine-elle toujours ?)

```
def revaux(l:List[Int],l2:List[Int]):List[Int]=
  l match {
    case Nil => l2
    case e::r => revaux(r,e::e::r)
}
```

V	*Oui*
R	*Non*

Exercice 11 (Définissez les fonctions append et map)

- append *La fonction qui concatène deux listes d'entiers* la *et* lb.
- map *La fonction qui applique une fonction* f *à tous les éléments d'une liste* l

Exercice 12 (Bonus)

Définissez le toString *pour la classe des multi-ensembles.*

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. **Traits étendus**
7. Exceptions

Traits étendus

- Les traits ne sont pas nécessairement *abstraits* et peuvent contenir une implantation
- Dans les traits, les déclarations de méthodes peuvent contenir du code et les champs peuvent contenir des valeurs

Exercice 13

Définissez un trait `Aire` avec deux champs `unite:String` et `taille:Int` ainsi qu'une méthode `toString`. Ensuite, définissez une classe `Rectangle` et une classe `Cercle` implantant `Aire`.

Remarque 2 (Implantation multiple)

Si une classe `A` implante deux traits `B` et `C` : `A extends B with C`. Si `B` et `C` définissent un même champ/méthode `f`, c'est la définition de `C` qui l'emporte.

Traits étendus (II)

Quiz 7 (Qu'affichent les programmes suivants ?)

```
trait A{
  override def toString="toto"
}
trait B{
  override def toString="tutu"
}
```

```
object C extends A with B
println(C)
```

V	"tutu"	R	"toto"

```
trait A{
  def f(x:Int)="toto"
}
trait B{
  def f(x:Any)="tutu"
}
```

```
class C extends A with B
val c=new C
println(c.f(10))
```

V	"tutu"	R	"toto"

Plan

1. Fonctions d'ordre supérieur et fonctions anonymes
2. Polymorphisme
3. Traits et fabriques d'objets
4. Héritage, visibilité et redéfinition
5. Modèle fonctionnel
   - Pattern matching
   - Fonctions récursives
6. Traits étendus
7. **Exceptions**

Des objets spécifiques : les exceptions

- Les exceptions de Scala se comportent comme celles de Java
- Déclenchement par `throw` sur un objet de type exception
  e.g. `throw new IllegalArgumentException` (Exception Java)
- Tant qu'elle n'est pas capturée une exception interrompt l'exécution de la méthode courante et remonte la pile des méthodes appellantes
- La capture se fait par une construction `try-catch-case` :

```
try (m(3)=="ok") catch {
    case e:RuntimeException => throw e
    case _:NoSuchElementException => false }
```

- On définit de nouvelles exceptions en étendant la classe `Throwable` :

```
class monException(x:String) extends Throwable{
    val content=x
}
```

Initiation au Génie Logiciel

Cours 6

–

Introduction au « développement agile »

Plan

1. Cycles de développement
2. Méthodes agiles, principes généraux
3. Comment se passe un Sprint ?
4. Principes de développement agiles : TDD, YAGNI, KISS

Bibliographie

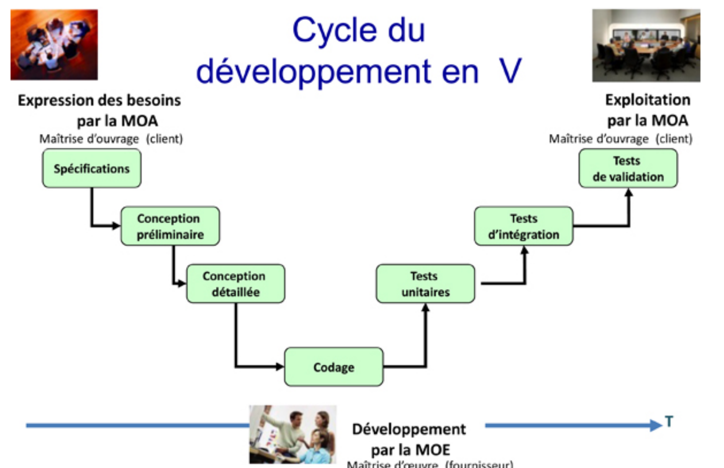
- *L'informatique agile, c'est quoi ?*. D. Certain. Interstices. 2013. `https://interstices.info/jcms/int_71726/linformatique-agile-cest-quoi`
- *XP : Extreme programming*. Wikipédia.
- *TDD : Test Driven Development*. Wikipédia.

Plan

1. Cycles de développement
2. Méthodes agiles, principes généraux
3. Comment se passe un Sprint ?
4. Principes de développement agiles : TDD, YAGNI, KISS

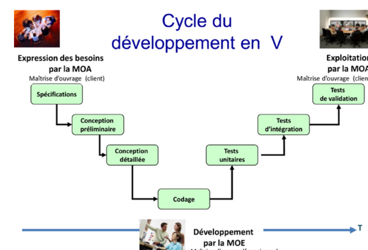
Cycles de développement : le cycle en V

80% des développements logiciels sont organisés selon un cycle en V



Crédits : Didier Certain

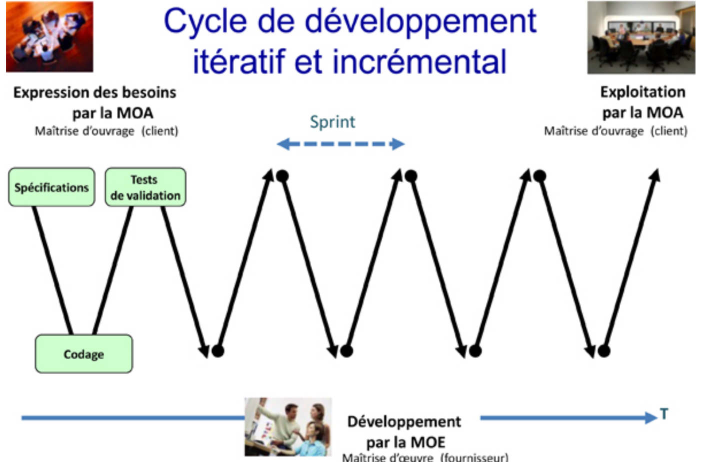
Cycles de développement : le cycle en V (II)



+ Découpage du développement rationnel, logique, stable
+ Identification claire des documents à produire, des rôles des équipes
+ Convient aux grosses équipes, développement de logiciels "critiques"

– Manque de souplesse : le client ne peut pas changer ses objectifs en cours de route !
– Découverte tardive des erreurs de spéc./conception par le client
– Intégration finale risquée : elle s'effectue à la fin du cycle (1 à 2 ans)

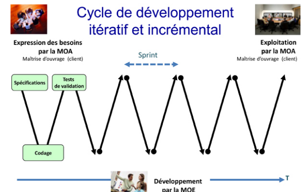
Cycles de développement : le cycle itératif



Un sprint = intégration d'un ensemble de nouvelles fonctionnalités

Crédits : Didier Certain

Cycles de développement : le cycle itératif (II)



+ Intégration continue : dispose en *permanence* d'un logiciel fonctionnel (cycles courts : 2 semaines)
+ Validation rapide et fréquente du logiciel par le client (au moins à chaque fin de sprint)
+ Le client peut changer ses objectifs (pas de spécification initiale figée)

– Inadapté au développement de logiciels critiques (qui ont besoin de spécifications détaillées et stables)
– Mise en place complexe pour les grosses équipes (> 12 p.) et équipes multi-localisées

Plan

1. Cycles de développement
2. Méthodes agiles, principes généraux
3. Comment se passe un Sprint ?
4. Principes de développement agiles : TDD, YAGNI, KISS

Méthodes agiles : *une* implantation du cycle itératif

Manifeste agile

≪ Nous découvrons comment mieux développer des logiciels par la pratique et en aidant les autres à le faire. Ces expériences nous ont amenés à valoriser :

- Les individus et leurs interactions plus que les processus et les outils
- Des logiciels opérationnels plus qu'une documentation exhaustive
- La collaboration avec les clients plus que la négociation contractuelle
- L'adaptation au changement plus que le suivi d'un plan

Nous reconnaissons la valeur des seconds éléments, mais privilégions les premiers. ≫

Méthodes "agiles" : le bestiaire

- RAD, ASD, FDD, Kanban, Lean, . . .
- Scrum
  - User Stories
  - Sprint
  - Sprint planning
  - Daily Scrum
  - Démo de fin de Sprint
- XP : Extreme Programming
  - Intégration continue
  - TDD : Test Driven Development
  - Pair programming
  - Principes YAGNI et KISS

Nous allons piocher des éléments dans Scrum et XP pour organiser le travail en TP

Préliminaire au développement agile : les user-stories

Définition 1 (User-story)

Une user-story (récit utilisateur) est une phrase simple décrivant une fonctionnalité du logiciel à développer. Idéalement, la phrase contient trois éléments descriptifs : qui ?, quoi ? et pourquoi ?.

En tant que <qui>, je veux <quoi> afin de <pourquoi>

Exemple 2 (Une User-story pour le robotWeb)

En tant qu'utilisateur, je veux pouvoir taper un mot clé afin d'obtenir une liste de pages web contenant ce mot clé.

Exemple 3 (Une User-story pour avatar)

En tant qu'utilisateur, je veux pouvoir taper la phrase "Je cherche la gare" et appuyer sur entrée. Deux bulles apparaissent dans l'interface. L'une contient ma requête et l'autre contient "L'adresse de la gare est : place de la gare".

Le développement agile commence : la version 0 du logiciel

- Les user-stories vont être intégrées au fur et à mesure dans le logiciel
- La version 0 (V0) est quasiment une coquille vide : elle doit être rapidement opérationnelle
- Idéalement, la V0 doit faire *intervenir* tous les composants du logiciel final : **périphériques, interf. graphiques, syst. de fichiers, réseau, etc.**

Exemple 4 (Version 0 du logiciel pour le robot web)

- Lit une chaîne de caractère au clavier
- Lit une URL sur internet
- Ecrit un contenu quelconque dans un fichier

Remarque 1

Si la V0 fait bien intervenir tous les éléments de la chaîne, on pourra détecter très tôt tout problème technique majeur : par exemple des incompatibilités matérielles ou logicielles.

Le point de départ : la version 0 du logiciel (II)

Exercice 1

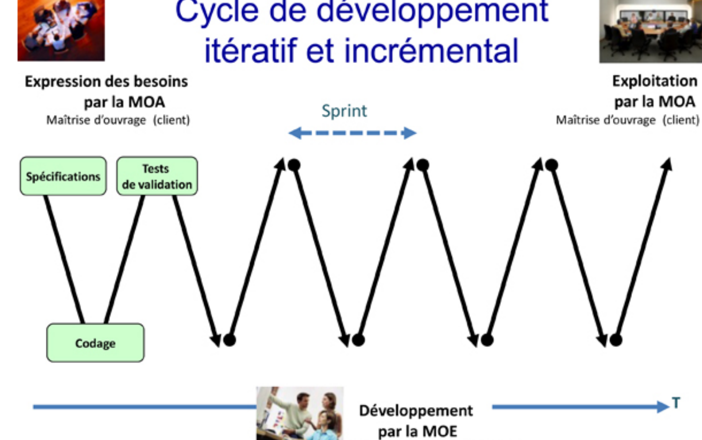
Quelle seraient de bonnes V0 pour les logiciels suivants :

1. *Navigateur web*
2. *Eclipse*
3. *Git*

Plan

1. Cycles de développement
2. Méthodes agiles, principes généraux
3. Comment se passe un Sprint ?
4. Principes de développement agiles : TDD, YAGNI, KISS

Cycles de développement : le cycle itératif



Un sprint = intégration d'un ensemble de nouvelles fonctionnalités

Crédits : Didier Certain

Comment se passe un Sprint ?

Client + Développeurs

1. Le client définit la liste de user-stories à intégrer pendant le Sprint

Développeurs

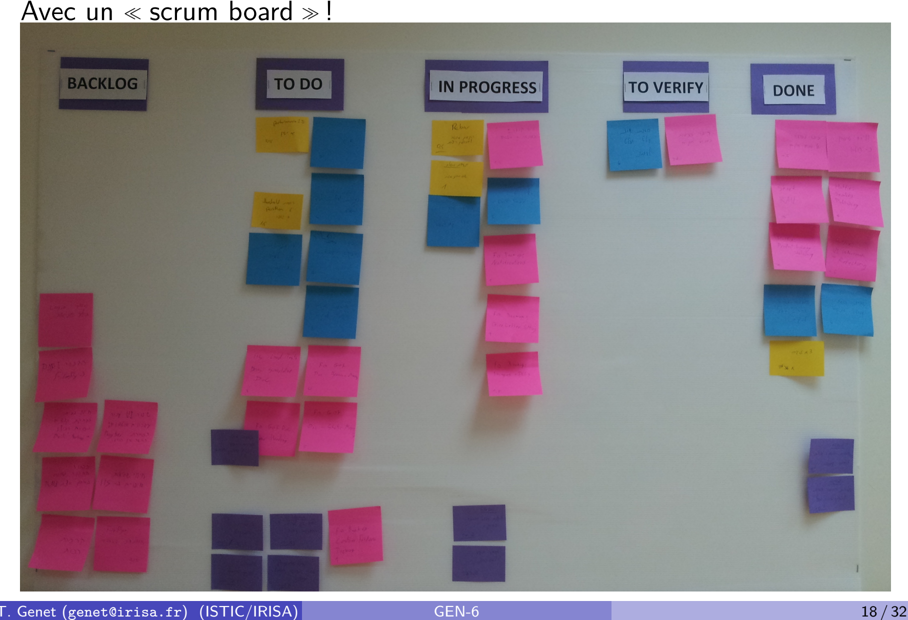
2. Affecter les user-stories à des équipes (Scrum Planning)
3. Développer, intégrer, tester le code du Sprint
4. Si nécessaire, pendant le Sprint, les équipes se réorganisent

Client + Développeurs

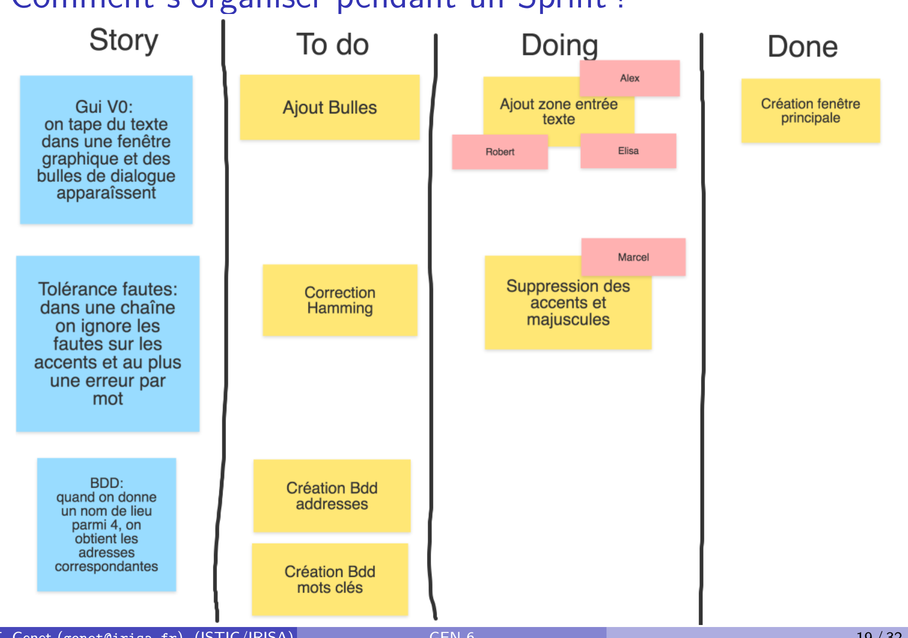
5. A la fin du Sprint : démo avec le client qui vérifie que les user-stories sont couvertes par la version courante du logiciel

Comment s'organiser pendant un Sprint ?

Avec un ≪ scrum board ≫ !



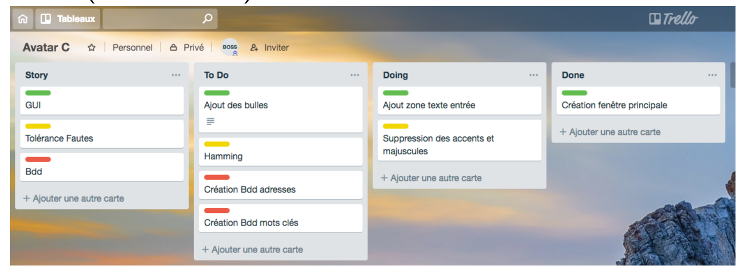
Comment s'organiser pendant un Sprint ?



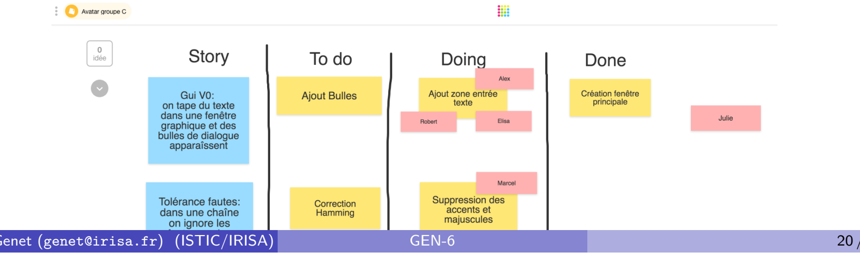
Comment s'organiser pendant un Sprint ?

Et si on ne partage pas de lieu pour laisser un tableau de post-it ?

- Trello (trello.com)



- Klaxoon (www.klaxoon.fr)



Comment se passe un Sprint ?

Client + Développeurs

1. Le client définit la liste de user-stories à intégrer pendant le Sprint

Développeurs

2. Affecter les user-stories à des équipes (Scrum Planning)
3. Développer, intégrer, tester le code du Sprint
4. Si nécessaire, pendant le Sprint, les équipes se réorganisent

Client + Développeurs

5. A la fin du Sprint : démo avec le client qui vérifie que les user-stories sont couvertes par la version courante du logiciel

Comment développer du code pendant un Sprint ?

- Méthode de dével. du code : TDD + KISS + YAGNI
- Méthode de validation du code : Intégration continue (Git + JUnit)
  - Le code correspondant à la "user-story" est intégré à la version courante du logiciel
  - On teste que la modification ne fait pas régresser le logiciel !
  - On teste que la modification satisfait bien la "user-story"
- Méthode de travail Pair programming (programmation en binômes)
  - Le pilote a le clavier. C'est lui qui code.
  - Le co-pilote, l'aide en suggérant de nouvelles solutions ou en décelant d'éventuels problèmes
  - Ils échangent leurs rôles régulièrement
  - Les binômes changent d'une séance à l'autre pour améliorer la communication et la connaissance collective de l'application

Plan

1. Cycles de développement
2. Méthodes agiles, principes généraux
3. Comment se passe un Sprint ?
4. Principes de développement agiles : TDD, YAGNI, KISS

Principe KISS (Keep It Simple, Stupid)

Eviter toute complexité non nécessaire

- Rechercher la simplicité dans la conception
- Simplifier le code de la version courante si nécessaire (Refactoring)

Exemple 5 (On veut représenter un ensemble de villes)

Pas KISS

```
class Ville(n:String){
  var nom=n
}
class MutableVilleSet {
  def ajouterVille(v:Ville):Unit={ ... }
}
val s= new MutableVilleSet
s.ajouterVille(new Ville("Rennes"))
s.ajouterVille(new Ville("Milan"))
```

KISS

```
case class Ville(n:String)
val s= Set(Ville("Rennes"),Ville("Milan"))
```

Principe YAGNI (You Ain't Gonna Need It)

Ne pas prévoir ce qui, de toute façon, ne sera jamais utilisé par la suite !

Exemple 6 (Dans le premier avatar, on veut stocker 5 adresses)

Pas YAGNI

```
object BDD{
  val id: Option[Long] =
    SQL("insert into Ad(name, address)
            values (name, address)").
        on('name -> "Gare",
           'address -> "19, Place de la Gare")
     .executeInsert() [...] }
```

YAGNI

```
object Bdd{
  val adr=
     Map("Gare" -> "19, Place de la Gare"), [...]) }
```

- Raccourcit le temps de développement
- Evite l'alourdissement de l'architecture
- Elimine des bugs potentiels... dans du code inutile !

TDD : Test Driven Development

Pour ajouter une fonctionnalité X à un logiciel :

1. Ecrire les tests T montrant que X fonctionne
2. Vérifier que le test échoue (X n'est pas encore développée)
   $\Longrightarrow$ Permet de vérifier que le test est valide !
3. Ecrire le code nécessaire pour passer T (et pas plus)
4. Vérifier que les tests T passent, sinon retourner en 3
5. Si nécessaire, remanier le code pour le simplifier (KISS)

Quiz 1 (f existe, on ajoute un test pour vérifier qu'elle est commutative)

```
@Test
def test1{
  assertEquals(f(11,1),f(11,1))
}
```

```
def f(x:Int,y:Int)= x
```

Quelle est l'étape du TDD qui a été négligée | V | 2 | R | 4 |

TDD : Test Driven Development (II)

Exemple 7

On développe un logiciel dans lequel on peut cliquer sur un bouton pour modifier une liste d'éléments.

- L'équipe A développe l'interface graphique
- L'équipe B développe le traitement des listes
- L'équipe C développe l'affichage graphique des listes

La première "user-story" à traiter est : « L'utilisateur clique sur un bouton, l'application supprime 2 dans la liste [1, 2, 3] et affiche le résultat. »

TDD : un exemple de code de l'équipe B

(1) On écrit le test

```
import org.junit.Assert._
import org.junit.Test
import ListTools._

class TestListTools {

  // delete supprime 2 dans
  // la liste [1,2,3]
  @Test
  def test_supplelement{
    assertEquals(
      List(1,3),
      delete(2,List(1,2,3)))
  }
}
```

```
object ListTools {
 def delete(x:Int,l:List[Int])=
     ???
}
```

(2) on vérifie que le test échoue !

TDD : un exemple de code de l'équipe B

(3) On écrit le code passant le test

```
import org.junit.Assert._
import org.junit.Test
import ListTools._

class TestListTools {

  @Test
  def test_supp1element{
    assertEquals(
      List(1,3),
      delete(2,List(1,2,3)))
  }
}
```

```
object ListTools{
 def delete(x:Int,l:List[Int])=
     List(1,3)
} // KISS + YAGNI :-)
```

(4) on vérifie que le test passe

... et c'est tout, pour ce test !

B peut déjà envoyer son code qui valide la première "user-story" ! !

⟹ L'avancement de A et C n'est pas bloqué par l'attente du code de B

TDD : l'exemple, seconde "user-story"

Exemple 8 (B étudie la seconde "user-story" à intégrer)

L'utilisateur clique sur un bouton cela supprime 1 dans la liste $[1, 2, 1, 1]$.

(1) On écrit le test

```
[...]
  @Test
  def test_supp1element{
    assertEquals(
      List(1,3),
      delete(2,List(1,2,3)))
  }

  @Test
  def test_suppToutes_occ{
    assertEquals(
      List(2),
      delete(1,List(1,2,1,1)))
  }
```

```
object ListTools {
 def delete(x:Int,l:List[Int])=
     List(2,3)
}
```

(2) on vérifie que le test échoue !

TDD : l'exemple, seconde "user-story"

(3) On écrit le code passant le test

```
[...]

  @Test
  def test_supp1element{
    assertEquals(
      List(1,3),
      delete(2,List(1,2,3)))
  }

  @Test
  def test_suppToutes_occ{
    assertEquals(
      List(2),
      delete(1,List(1,2,1,1)))
  }
}
```

```
object ListTools{
 def delete(x:Int,l:List[Int])=
     if (l==List(1,2,3))
     List(1,3)
     else List(2)
}  // Pas KISS :/
```

(4) on vérifie que le test passe

(5) On remanie le code

```
object ListTools{
 def delete(x:Int,l:List[Int])=
   l.filter(_!=x)
}  // KISS :-)
```


Déroulement des séances de TPs : Daily Scrum

Chaque séance de TP (en mode projet) commencera par : un Daily Scrum

Définition 9 (Daily Scrum – Mélée journalière)

C'est une réunion courte (max 10 minutes) où chaque membre du groupe dit, à son tour :

1. Ce qu'il a fait à la dernière séance
2. Ce qu'il doit faire à cette séance
3. Les problèmes qu'il a pour atteindre cet objectif

Quelques règles :

- Tout le monde est debout
- Le but est de signaler les problèmes, pas de les résoudre
  Ils seront résolus plus tard, après le daily scrum.

Initiation au Génie Logiciel

Cours 7

–

Propriétés logiques et programmation par contrats

Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Invariants, pré et post conditions

- Ce sont des principes formels de raisonnement sur les programmes
- Définis pour la logique de Hoare (Hoare 1969)
- Utilisé pour la programmation par contrats (Meyer 1985)
- Trois grands types d'assertions/propriétés ($\approx$ conditions booléennes)

Définition 1 (Invariant)

Un invariant est une propriété qui est (constamment) vraie pendant l'exécution d'un programme/fonction.

Définition 2 (Précondition)

Une précondition est une propriété qui est vraie avant l'exécution d'un programme/fonction.

Définition 3 (Postcondition)

Une postcondition est une propriété qui est vraie après l'exécution d'un programme/fonction.

Invariants, pré et post conditions en Scala

Exemple 4 (Précondition en Scala – require)

```
var credit= 1000

def retrait(x:Int):Int={
  require(x <= credit)
  credit= credit - x
  credit
}
```

Exemple 5 (Invariant en Scala – assert)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  assert(credit >= 0)
  credit
}
```


Invariants, pré et post conditions en Scala (II)

Exemple 6 (Postcondition en Scala – ensuring)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  credit
} ensuring (credit >= 0)
```

Exemple 7 (Postcondition sur le résultat : fonction anonyme)

```
var credit= 1000

def retrait(x:Int):Int={
  credit= credit - x
  credit
} ensuring (res => res >= 0)
```


Invariants, pré et post conditions en Scala (III)

Exemple 8 (Postcondition et valeurs antérieures : variables fantômes)

```
var credit= 1000


def retrait(x:Int):Int={

  credit= credit - x
  credit
} ensuring (res => (res== ???))

var credit= 1000
var creditPre=credit // Variable "fantôme" utilisée pour
                     // exprimer la propriété
def retrait(x:Int):Int={
  creditPre=credit
  credit= credit - x
  credit
} ensuring (res => (res== creditPre - x) && credit==res)
```


Invariants, pré, post-condition – Le quizz

Quiz 1 (Sur les programmes suivant quelle propriété va être violée ?)

```
def plus(x:Int,y:Int):Int={
  require(x+y>0)
  x
} ensuring (res =>(res==x+y))

plus(10,20)
```

V	precond.	R	postcond.

```
def moins(x:Int,y:Int):Int={
  require(x>y)
  x - x
} ensuring (res => res>0)

moins(10,20)
```

V	precond.	R	postcond.

```
def f(a:Int):Int={
  require(a match {
    case 18 | 19 => true
    case _ => false})
  18}
}
f(10)
```

V	precond.	R	aucune

```
def max(a:Int,b:Int):Int={
  30
} ensuring (res => (res>=a
                && res>=b))

max(10,20)
```

V	aucune	R	postcond.

Utilisation des assertions en Scala sous Eclipse

Les assertions sont vérifiées pendant l'exécution du programme

- Elles permettent de détecter et de localiser plus facilement les erreurs
- Pour une fonction f donnée, elles déterminent si l'erreur provient :
  - d'un mauvais appel de f (exception sur `require`)
  - d'une erreur dans le corps de f (exception sur `assert`)
  - d'une erreur dans le résultat produit par f (exception sur `ensuring`)

Démo sur `Compte.scala`

Désactivation des assertions

- Avant de livrer le logiciel, il est possible de désactiver les assertions
- Utile si elles diminuent **significativement** les performances
- Dans Eclipse : Clic droit sur le projet > Properties > Scala Compiler > Advanced > Use Project Settings > Xdisable-assertions

Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Décrire les propriétés par des formules logiques ?

En GEN, formule logique ≈ condition booléenne

Exercice 1 (Quelle formule pourrait définir le résultat de `max` ?)

```
def max(a:Int,b:Int):Int={
  [...]
} ensuring (???)
```

Remarque 1 (Comment définir un résultat de fonction par une formule ?)

- *Ne pas se limiter à un cas particulier : généraliser la formule*
- *Si nécessaire, utiliser des fonctions auxiliaires pour la vérification*
- *Si nécessaire, définir des fonctions auxiliaires pour la vérification*

Exercice 2 (Quelle formule pourrait définir le résultat de `delete` ?)

```
def delete(x:Int,l:List[Int]):List[Int]={
  l.filter(_ != x)
} ensuring (???)
```


Décrire les propriétés par des formules logiques ? (II)

Quiz 2 (Les postconditions définissent elles correctement ce qui est attendu ?)

```
def divisionEntiere(n:Int,d:Int):(Int,Int)={
  require(d>0 && n>=0)
  [...]
} ensuring (res => n== (res._1 * d + res._2) &&
                        res._2<= d)
```

V Oui / R Non

```
def plus(x:Int,y:Int):Int={
  [...]
} ensuring (res => res==plus(x,y))
```

V Oui / R Non

```
def intersect(a:Set[Int],b:Set[Int]):Set[Int]={
  [...]
} ensuring (res => res.forall(a.contains(_)))
```

V Oui / R Non

Décrire les propriétés par des formules logiques ? (III)

Exercice 3 (Quelle formule pourrait définir `intersect` ?)

```
def intersect(a:Set[Int],b:Set[Int]):Set[Int]={
  [...]
} ensuring (???)
```

Exercice 4 (Quelle formule pourrait définir `triCroissant` ?)

```
def triCroissant(l:List[Int]):List[Int]={
  [...]
} ensuring (???)
```


Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Des formules pour factoriser les tests – Property based testing

On peut (parfois) automatiser l'écriture des tests unitaires par :

- écriture de la post-condition de la fonction (a.k.a. oracle)
- des tests automatiques exhaustifs/aléatoires (a.k.a. QuickCheck)

Exemple 10 (Sur `intersect(a:Set[Int],b:Set[Int]):Set[Int]`)

- au lieu de

```
assertEquals(Set(),intersec(Set(0),Set(1,2)))
assertEquals(Set(2),intersec(Set(1,2),Set(2,3)))
// 2 tests en 2 lignes
```

- écrire la post-condition de la fonction `intersec`
  définir un générateur aléatoire d'ensembles d'Int : `genSet:Set[Int]`

```
for (i<-1 to 100000) insersec(genSet,genSet)
// 100.000 tests en 1 ligne!
```

Démo sur le package testers.

Des formules pour factoriser les tests (II)

Exemple 11 (La post-condition d'`intersec`)

```
def intersec(a:Set[Int],b:Set[Int]):Set[Int]={
 [...]
} ensuring (inter =>{
     var res=true
     // tous les éléments de l'intersection sont dans a ET b
     for (e <- inter) res = res && a.contains(e) && b.contains(e)
     // les éléments dans a ET dans b sont dans l'intersection
     for (e <- b) if (a.contains(e)) res=res && inter.contains(e)
     res
  })
```

Exemple 12 (La post-condition d'`intersec`, version ordre supérieur)

```
def intersec(a:Set[Int],b:Set[Int]):Set[Int]={
 [...]
} ensuring (inter =>
            inter.forall (e => a.contains(e) && b.contains(e))
            && a.forall(e => !b.contains(e) || res.contains(e)))
```

Des formules pour factoriser les tests (III)

Exemple 13 (Testeur aléatoire)

```
import scala.util.Random
class TestInter {
  val rand= new Random

  // Générer un entier aléatoire entre i et j
  def genInt(i:Int,j:Int)= rand.nextInt(j-i+1)+i

  //Générer un ensemble, de taille i, d'entiers aléatoires
  def genSetL(i:Int):Set[Int]=
    if (i<=0) Set[Int]() else genSetL(i-1)+genInt(0,10)

  //Générer un ensemble, de taille aléatoire, d'entiers aléatoires
  def genSet= genSetL(genInt(0,5))

  @Test
  def test1{                        //100.000 tests, hop!
    for (i<-1 to 100000) intersec(genSet,genSet)
  }
```


Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Programmation défensive

En programmation défensive, autour d'une fonction `f` :

- l'utilisateur se prémunit d'une implantation erronée/malveillante de `f`
- l'implanteur se prémunit d'une utilisation erronée/malveillante de `f`

Utilisateur de `f`	Interface `def f(x:T1):T2`	Implanteur de `f`
vérifie : • le résultat de `f` • que `f` n'a pas modifié `x` • que l'appel de `f` n'a pas modifié l'état global • ...		vérifie que les entrées (`x`) de `f` sont correctes

Se protéger en tant qu'implanteur

Exemple : `max(x:Array[Int]):Int`, maximum d'un tableau d'entiers

L'implanteur de `max` doit vérifier que :

- `x` n'est pas la référence `null`
- le tableau `x` comporte au moins un élément

```
def max(x:Array[Int]):Int={
    require(x!=null)    // rejette le pointeur null
    require(x.length>0) // rejette les tableaux vides
    var max= x(0)
    for (i <- x) if (i>max) max=i
    max
}
```

Se protéger en tant qu'utilisateur

Exemple : `max(x:Array[Int]):Int`, maximum d'un tableau d'entiers

Exercice 5 (Utilisez `max` pour calculer le maximum d'un tableau)

```
val t=Array(1,2,3)
...
var res= max(t)
...
```

Quel code faut-il placer autour de l'appel à `max` pour se protéger d'implantations erronées ? Des exemples d'implantations erronées :

1. ```
   def max(x:Array[Int])=
     Int.MaxValue
   ```
2. ```
   def max(x:Array[Int])= x(0)
   ```
3. ```
   def max(x:Array[Int])= {
     var max= x(0)
     for (i<-x) if(i>max) max=i
     x(0)=10
     max
   }
   ```


Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

La programmation par contrats

La programmation par contrats définit les propriétés attendues sur les fonctions dès leur conception (avant leur implantation)

« de la programmation défensive prévisionnelle »

L'utilisateur et l'implanteur se protègent à l'aide d'un contrat

Le contrat définit l'utilisation/implantation des fonctions partagées

La programmation par contrats (exemple)

La programmation par contrat raffine la notion de trait/interface

- Un trait/interface donne le nom et le types des opérations
- Un contrat le complète par des propriétés attendues sur ces opérations

Exemple 14 (Reprise de l'exemple `IntQueue` du Cours 3)

Utilisateurs	Interface	Implanteurs

Une équipe réalise un logiciel <u>utilisant</u> des `IntQueue`	`trait IntQueue {`
	`def get:Int`
	require(!empty)
	`def put(x:Int):Unit`
	ensuring(!empty)
	`def empty:Boolean}`
		Une équipe <u>implante</u> le trait `IntQueue` dans une classe `MyQueue`

Remarque 2 (Choix et syntaxe des contrats dans l'Exemple 14)

Dans l'exemple, les contrats sont volontairement simples, pour illustrer. Nous verrons la syntaxe Scala pour définir ces contrats dans la suite.
   ```

La programmation par contrats (II)

Le contrat définit les responsabilités de l'utilisateur (ou client) et de l'implanteur (ou fournisseur)

- L'utilisateur doit s'assurer qu'il appelle les opérations en respectant les préconditions
- L'implanteur doit s'assurer que les opérations qu'il développe satisfont les postconditions

Quiz 3 (L'utilisateur et l'implanteur ont-ils respecté le contrat ?)

```
def retarder(q:IntQueue):Unit=
  q.put(q.get)

def vider(q:IntQueue):Unit=
  while(!q.empty) q.get
```

V	Oui	R	Non

```
class MyQueue extends IntQueue{
  private var b= List[Int]()
  def get= {val h=b(0)
            b=b.drop(1)
            h}
  def put(x:Int)= {b=b:+x}
  def empty=b.isEmpty
 }
```

V	Oui	R	Non

Une façon d'implanter les contrats en Scala

Le contrat à définir	Le code Scala correspondant
`trait IntQueue {` `  def empty:Boolean`	`trait IntQueue{` `  def empty:Boolean`

`  def get:Int`
`    require( !empty)`
	`  def get:Int ={`
	`    require(!empty)`
	`    getIMP`
	`  }`
	
	`  protected def getIMP:Int`

`  def put(x:Int)`
`    ensuring( !empty)`
`}`
	`  def put(x:Int)={`
	`    putIMP(x:Int)`
	`  } ensuring (!empty)`
	
	`  protected def putIMP(x:Int):Unit`
	`}`

Une façon d'implanter les contrats en Scala (II)

Le code de la classe MyIntQueue implantant IntQueue devient :

Version sans contrats

```
class MyQueue extends IntQueue{
  private var b= List[Int]()

  def get= {
    val h=b(0)
    b=b.drop(1)
    h}

  def put(x:Int)= {
    b=b:+x
  }

  def empty=b.isEmpty
 }
```

Version avec contrats

```
class MyQueue extends IntQueue{
  private var b= List[Int]()

  protected def getIMP= {
    val h=b(0)
    b=b.drop(1)
    h}

  protected def putIMP(x:Int)= {
    b=b:+x
  }

  def empty=b.isEmpty
 }
```


Une façon d'implanter les contrats en Scala (III)

Exercice 6

Définir un contrat `ArrayCopy` *qui propose une fonction* `copy(t1:Array[Char],t2:Array[Char]):Unit` *qui copie tous les caractères d'un tableau* `t1` *dans un tableau* `t2`*. Dans le contrat, on se protègera de tous les cas de figure suivants :*

- *les tableaux peuvent être* `null`
- `t2` *est trop petit pour recevoir tous les éléments de* `t1`
- *la copie n'a pas été réalisée convenablement*
- `t1` *a été modifié par la copie*
- ...

Remarque 3 (Invariants de classes)

En Scala il est également possible de définir des invariants de classes mais la construction est plus complexe.

Plan

1. Invariants, pré et post conditions
2. Des formules logiques pour exprimer des propriétés
3. Application en Génie Logiciel : factorisation des tests
4. Application en Génie Logiciel : programmation défensive
5. Application en Génie Logiciel : programmation par contrats
6. Parenthèse culturelle : au delà des tests et des contrats

Des formules logiques prouvées sur des programmes

- Au lieu de tester une formule sur un programme, on peut la prouver
- Démo de l'assistant de preuve Isabelle/HOL
- Une preuve remplace une infinité de tests !
- Principe émergent dans le dommaine du logiciel critique
  - CompCert : compilateur C certifié pour Airbus
  - METEOR : logiciel embarqué de la Ligne 14 du métro pour la RATP
- Vous en entendrez parler dans d'autres UE de
  - L3 : LOG, ProgC (programmation de confiance)
  - M1 : ACF, MVFA

Bases de Scala pour GEN

1 Définition de val/var

1.1 Typage

val *i*:**Int** = 0
val *c*:**Char** = *'a'*
val *u*:**Unit** = ()
val *s1*:**Set**[*Int*] = **Set**()
val *table*:**Map**[*String, Int*] = **Map**()
val *d*:**Double** = 1.45
val *s*:**String** = *"toto"*

1.2 Différence entre val et var

val *ne peut pas être réassigné*
var *peut être réassigné*

2 Conditionnelles

2.1 Opérateurs booléens

Pour tous les objets : ==, !=
Spécifique aux numériques : >, >=, <, <=
Connecteurs logiques : && (et), || (ou), ! (non)

2.2 Expressions *if*

if (*condition*) *e1* **else** *e2*

2.3 Expressions *match-case*

```
x match {
        case motif1 => e1
        case motif2 => e2
        ...
}
```

2.4 Expressions *while* et *for*

Le type d'un while/for est Unit

while ($x < 10$) { ... }

for (x <- s) { ... }
Pour tout s de type iterable: List, Set, Map, etc.

(ou)
for (x <- 1 to 10) { ... }
(ou)
for (x <- 10 to $(1, -2)$) { ... }
pour aller de 10 à 1 par pas de -2

3 Définition de fonction

```
def f(x1:Type1,...,xn:Typen) : Type_résultat = {
    Corps de la fonction
}
```

La dernière expression du corps de la fonction doit être de type Type_résultat.

4 Classes, objets, traits

4.1 Classes

Une classe permet de produire une série d'objets ayant des caractéristiques communes.

```
class C(x1:Type1,...,xn:Typen) {
        val champ1 : Type = ...      Champ
        def fonction1(...) : Type = ...      Fonction
        def this(...) = ...      Constructeur auxiliaire
}
```

Création d'un objet : **val** x = **new** C(v1,...,vn)
Accès aux champs : x.champ1
Appel des méthodes/fonctions de l'objet : x.fonction1(...)

4.2 Objets singletons

Un objet singleton est unique.

```
object O {
        val champ1 : Type = ...      Champ
        def fonction1(...) : Type = ...      Fonction
}
```

L'objet est déjà crée et il est unique, il se nomme O
Accès aux champs de l'objet : O.champ1
Appel des méthodes/fonctions de l'objet : O.fonction1(...)

4.3 Case classes

On définit des objets à l'aide de **case class** si **toutes** les données de l'objet apparaissent dans le constructeur.
case class CC(c1:Type1,...,cn:Typen)
Création d'un objet : **val** x = CC(v1,...,vn)
Accès aux champs de l'objet : x.c1, x.c2, ... ou par :
    x **match** {**case** CC(y,z,_,...) => ... y ... z ...}

4.4 Trait

```
trait T {
        val champ1 : Type
        def fonction1(...) : Type
}
```

4.5 Héritage/implantation

Classe C héritant d'une classe C2 : **class** C **extends** C2 {...}
Classe C implantant un trait T : **class** C **extends** T {...}
Case Classe C implantant un trait T : **case class** C **extends** T {...}
Objet O implantant un trait T: **object** O **extends** T {...}

5 Tuples

val x : (Int, String, Int) = (1,"Test",5)
Accès au ième élément: x._i (ex: x._3 est égal à 5) ou par :
    x **match** {**case** (_,_,z) => ... z ...}

6 Listes (immutables)

val L = **List**(1, 2, 3, 4, 5)
var L2:List[Int] = **List**()
Ajout en tête de liste : val L1 = 0::L
Ajout en queue de liste : val L3 = L: +6
Concaténation : val L4 = L1++L3
Renverser : val L5 = L4.**reverse**
Consulter une valeur à une position : val i= L(2)
Itérer sur une liste : **for**(e <- L) {...}
Convertir en ensemble : val s:Set[Int] = L4.**toSet**

7 Ensembles (immutables)

val s = **Set**(1, 2, 3, 4, 5)
var sa : **Set[Int]** = **Set**()

Teste si l'ensemble n'est pas vide: s.**nonEmpty**
Existence d'une valeur: s.**contains**(2)
Ajout d'un élément: val s5 = s + 6
Suppression d'un élément: val s6 = s − 4
Intersection: val s2 = s.**intersect**(s2)
Union: val s3 = s ++ s2
Différence: val s4 = s −− s2
Test d'égalité: s == s2
Maximun: s.**max**
Taille: s.**size**
Itérer sur un ensemble : **for**(e <- s) {...}
Convertir en liste : val L5 = s.**toList**

8 Map (immutables)

val m = **Map**('C'− >"Carbon",'H'− >"Hydrogen")
var m2:**Map[Char,String]** = **Map**()
Existence d'une clé : m.**contains**('C')
Recherche avec valeur par défaut: m.**getOrElse**('K',"None")
Ajout d'une association : val m3 = m + ('O'-> "Oxygen")
Taille : m.**size**
Liste des clés : m.**keys**
Liste des valeurs : m.**values**
Itérer sur une table : **for**((e1,e2) <- m) {...}

9 Array (mutables)

val t = **Array**("abc","def","ghijkl")
val t2:**Array**[String] = **new Array**(3)
Tableau de booléens, à 2 dimensions, initialisé à false:
val t3 = **Array**.fill(2,3)(false)
Consultation : t(1), t3(1)(2)
Modification : t(1)="hhh", t3(1)(2)=true
Taille : t.**size**
Itérer sur un tableau : **for**(e <- t) {...} ou **for**(0 <- t.size-1) {...}

10 Comment accumuler des valeurs dans une structure immutable

Pour les structures immutables (List, Set, Map, ...)
Par exemple: construire l'ensemble des prénoms à partir d'une liste L de couples nom-prénom.

```
def tousPrenoms(L:List[(String,String)]):Set[String]={
    var res : Set[String] = Set()
    for ((n,p) <- L) res = res + p
    res
}
```

11 Polymorphisme

11.1 Fonctions polymorphes

```
def first[T1,T2](p:(T1,T2)): T1 =
    p match {case (x,y) => x}
def remove[T](x:T,L:List[T]) : List[T] = L.filter(_ != x)
```

11.2 Traits et classes polymorphes

```
trait Queue[T]{
  def get:T
  def put(x:T):Unit}

class MyQueue[T] extends Queue[T]{
  private var b:List[T]=List()
  def get={val h=b(0); b=b.drop(1); h}
  def put(x:T):Unit= {b=b:+x}}
```

12 Fonctions d'ordre supérieur

Pour toutes les structures (List, Set, Map, Array, ...)
val s = Set(1,2,3,4,5,6)
Application d'une fonction à tous les éléments : s.**map**

```
val s2 : Set[Int] = s.map(_ + 1)
val s3 : Set[String] = s.map(_ .toString)
val s4 : Set[Int] = s.map(x => x * x)
```

Réduction de l'ensemble de valeurs 2 à 2 : s.**reduce**

```
val i1 : Int = s.reduce(_ + _)
val maximum : Int = s.reduce((x,y) => if (x>y) x else y)
```

Filtrage de l'ensemble de valeurs : s.**filter**

```
val s5 : Set[Int] = s.filter(_ > 2)
val s6 : Set[Int] = s.filter(x => (x % 2) == 0)
```

Toutes les valeurs vérifient une propriété : s.**forall**

```
val b1 : Boolean = s.forall(_ ! = 0)
val b2 : Boolean = s.forall(x => (x * x) > 4)
```

Au moins une valeur vérifie une propriété : s.**exists**

```
val b3 : Boolean = s.exists(_ == 2)
val b4 : Boolean = s.exists(x => (x % 2) == 0)
```

Comment définir une fonction d'ordre supérieur, l'exemple de **map**

```
def map[T1,T2](f:T1=>T2,l:List[T1]):List[T2]={
  l match {
    case Nil => Nil
    case e::r => f(e)::map(f,r)
  }
}
```

13 Visibilité des membres

Membre d'un objet = champ/fonction/classe interne/etc.
Par défaut tout membre est **public**
Un membre **public** est visible en dehors de l'objet
Un membre **private** n'est pas visible en dehors de l'objet
Un membre **protected** n'est pas visible en dehors de l'objet, sauf depuis un objet en héritant.

```
class A{
  val x:Int=1
  private val y:Int=2
  protected val z:Int=3}

class B extends A{
  val v= z}

val o= new B
```

Les champs o.x, o.v sont accessibles (et o.v contient la valeur de o.z). Les champs o.y et o.z ne sont pas accessibles. La valeur du champ z est accessible depuis le code de la classe B.