R2.01 - Développement Orienté Objets

Quelques "trucs" en Kotlin

Arnaud Lanoix Brauer
Arnaud.Lanoix@univ-nantes.fr

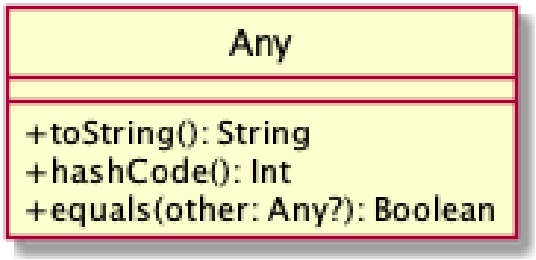
Département informatique

Sommaire

IUT Nantes

La classe `Any`

De manière **implicite**, toute classe Kotlin hérite de `Any`



- Utile quand on ne connait pas le type des objets à manipuler :
  `Any` = n'importe quel type
- Permet de redéfinir certaines méthodes :
  1. `toString(): String` : représentation d'un objet sous la forme d'une chaîne de caractères
  2. `hashCode(): Int` : donne un entier (unique) représentant l'objet
  3. `equals(other: Any?): Boolean` : teste l'égalité de l'objet avec un autre

`https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/`

IUT Nantes

Redéfinition de `toString()`

La méthode `toString()` est implicitement appelée dès qu'on doit convertir un objet en chaine de caractères.

```
class Chien(...) {
  var nom :String
  private var age : Int
  val race : String
  var poids : Double

  init {
    ...
  }
  override fun toString(): String {
    return ("$nom est un chien"
        + " de la race $race"
        + " age de $age"
        + " et pesant $poids")
  }
```

- Sans redéfinition de `tostring()` `println(rogue)` affiche `Chien@4d405ef7`
- Avec la redéfinition `println(rogue)` affiche `Rogue est un chien de la race Berger Australien age de 15 et pesant 30.2`

```
    var rogue = Chien("Rogue", 15, "Berger Australien", 30.2)
    var potter = Chien("Other", 40, "Beauceron", 39.4)
    println(rogue)
    println("son copain est : " + potter)
```

IUT Nantes

Redéfinition de `toString()`

La méthode `toString()` est implicitement appelée dès qu'on doit convertir un objet en chaine de caractères.

```
class Chien(...) {
  var nom :String
  private var age : Int
  val race : String
  var poids : Double

  init {
    ...
  }
  override fun toString(): String {
    return ("$nom est un chien"
        + " de la race $race"
        + " age de $age"
        + " et pesant $poids")
  }
```

- Sans redéfinition de `tostring()`
  `println(rogue)` affiche
  `Chien@4d405ef7`
- Avec la redéfinition
  `println(rogue)` affiche
  `Rogue est un chien de la race Berger Australien age de 15 et pesant 30.2`

```
    var rogue = Chien("Rogue", 15, "Berger Australien", 30.2)
    var potter = Chien("Other", 40, "Beauceron", 39.4)
    println(rogue)
    println("son copain est : " + potter)
```

IUT Nantes

Redéfinition de `hashCode()`

```
class Chien(...) {
  ...

  override fun hashCode() : Int {
    var result = nom.hashCode()
    result = result * 31 + age + race.hashCode()
    return result
  }
```

La méthode `hashCode()` est utilisée par certaines `Collections` pour optimiser l'insertion, la recherche et l'accès aux éléments dans la collection.

IUT Nantes

Redéfinition de `equals(...)`

```
class Chien(...) {
  ...

override fun equals(other: Any?): Boolean {
  if (this === other)      // se sont les memes references
    return true
  if (other === null)      // l'autre est null
    return false
  if other !is Chien)      // l'autre n'est pas du meme type
    return false
  // on regarde l'egalite des attributs
  if (nom != other.nom || age != other.age || race != other.race)
    return false
  return true
}
```

```
var rogue = Chien("Rogue", 15, "Berger Australien", 30.2)
var potter = Chien("Other", 40, "Beauceron", 39.4)

println(rogue.equals(potter))        // expected : false
println(rogue.equals(rogue))         // expected : true

var cloneDeRogue = Chien("Rogue", 15, "Berger Australien", 0.0)
println(rogue.equals(cloneDeRogue)) // expected : true
```

IUT Nantes

Operateurs `==` et `!=`

Redéfinir `equals(...)` permet également d'utiliser directement les opérateurs d'égalité et ou de différence de Kotlin pour la classe considérée.
En effet

- `a == b` est (automatiquement) traduit par
  `a?.equals(b) ?: (b === null)`
- `a != b` est (automatiquement) traduit par
  `!(a?.equals(b) ?: (b === null))`

```
println(rogue == potter)                // expected : false
println(rogue == cloneDeRogue)          // expected : true
println(rogue != rogue)                 // expected : false
println(cloneDeRogue != potter)         // expected : true
```

IUT Nantes

Redéfinition des opérateurs Kotlin

Comme pour `==` et `!=`, la plupart des opérateurs de Kotlin peuvent être utilisés dans une nouvelle classe donnée en alertredéfinissant (via `operator`) dans la classe la méthode correspondante.

Opérateurs mathématiques	
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
Opérateurs d'incrément	
`a++`	`a.inc()`
`a--`	`a.dec()`

IUT Nantes

Redéfinition des opérateurs Kotlin (2)

Opérateurs d'appartenance	
`a in b` `a !in b`	`b.contains(a)`
Accès indicé	
`a[i]`	`a.get(i)`
`a[i] = b`	`a.set(i,b)`
Opérateurs de comparaison	

`a > b`
`a < b`
`a >= b`
`a <= b`
	`a.compareTo(b)`

Voir également :
https ://kotlinlang.org/docs/operator-overloading.html

IUT Nantes

Exemple de redéfinition d'opérateur : comparaison

```
class Chien(...) {
  ...

  // on compare les poids, puis les ages
  operator fun compareTo(other : Chien) : Int {
    val comp = this.poids.compareTo(other.poids)
    if (comp == 0)
      return this.age.compareTo(other.age)
    return comp
  }
```

```
  println("### comparaisons ###")
  println(rogue <= potter)              // expected : true
  println(potter < cloneDeRogue)        // expected : false
  println(potter < potter)              // expected : false
```

IUT Nantes

Exemple de redéfinition d'opérateur : incrément

```
class Chien(...) {
  ...

  //on fait vieillir le chien de 12 mois
  operator fun inc() : Chien {
    var nouveau = Chien(this.nom, this.age,
                        this.race, this.poids)
    nouveau.age += 12
    return nouveau
  }
```

```
  println(rogue)
  // Rogue est un chien age de 15 mois
  rogue++
  println(rogue)
  // Rogue est un chien age de 27 mois
```

IUT Nantes

Exemple de redéfinition d'opérateur : appartenance

```
class Proprietaire {
  private val compagnons : Array<Chien?>

  fun adopte(compagnon : Chien) {...}

  operator fun contains(chienRecherche : Chien) : Boolean {
    for (chien in compagnons) {
      if (chienRecherche == chien)
        return true
    }
    return false
  }
```

```
  var rogue = Chien("Rogue", 15, "Berger Australien", 30.2)
  var potter = Chien("Other", 40, "Beauceron", 39.4)
  var cloneDeRogue = Chien("Rogue", 15, "Berger Australien", 0.0)
  arnaud.adopte(rogue)
  println(rogue in arnaud)          // expected : true
  println(potter in arnaud)         // expected : false
  println(cloneDeRogue in arnaud) // expected : true
```

IUT Nantes

Sommaire

IUT Nantes

Les packages en Kotlin

Package

Un package ("paquetage" ne français) est une unité logique d'organisation regroupant un ensemble de classes, d'interfaces et de sous-packages

- Les packages sont organisés sous forme de hiérarchie, c-à-d qu'il est possible de les imbriquer.
- Les packages évitent des conflits de noms, i.e. il peut y avoir deux classes avec le même nom dans deux packages différents.

En Kotlin

- on déclare qu'une classe est dans un package via
```
package nom.du.paquetage
```
- on doit importer une classe présente dans un package pour y avoir accès
```
import nom.du.paquetage.MaClasse
```
- on peut également importer tout le contenu d'un package
```
import nom.du.paquetage.*
```

IUT Nantes

Exemple de packages

```
package iut.animals

import iut.Appelable
import iut.personnes.Personne

abstract class Animal(...)
    : Appelable {
var maitre : Personne?
...
}
```

```
package iut.animals

class Chien(nom : String, age : Int)
    : Animal(nom, age), Joueur {
...
}
```

```
package iut.animals

class Chat(nom : String, age : Int)
    : Animal(nom, age) {
...
}
```

```
package iut.personnes

import iut.Appelable
import iut.animals.*

class Personne (nom:String)
    : Appelable {
  val compagnons : Array<Animal?>
  ...

}
```

```
import iut.animals.*
import iut.personnes.Personne

fun main() {
    var rogue = Chien("Rogue", 15)
    var gaga = Chat("Gaga", 36)
    rogue.aboyer()
    gaga.miauler()
    val arnaud = Personne("Arnaud")
    arnaud.adopte(rogue)
}
```

IUT Nantes

Diagramme de paquetages UML

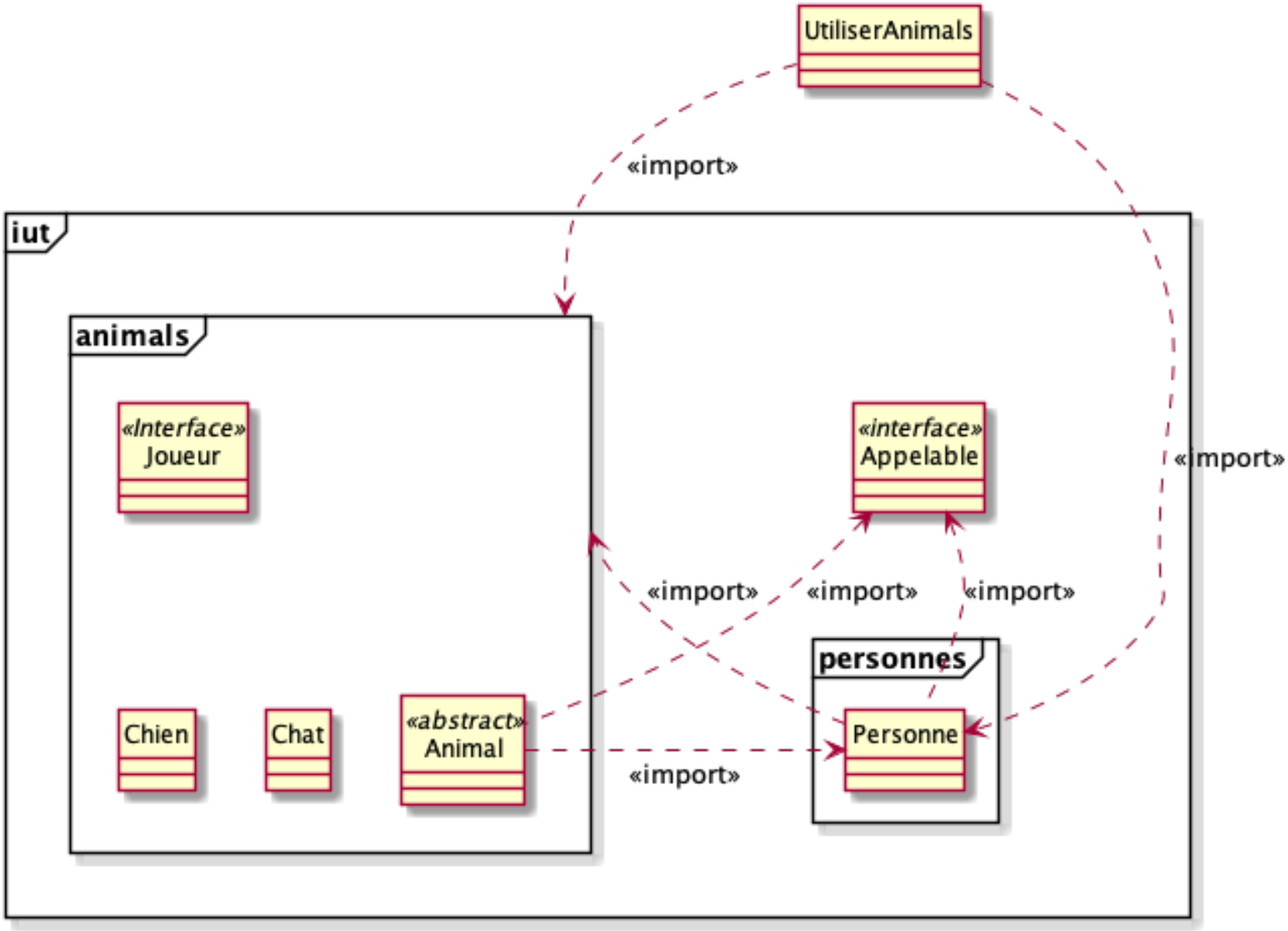
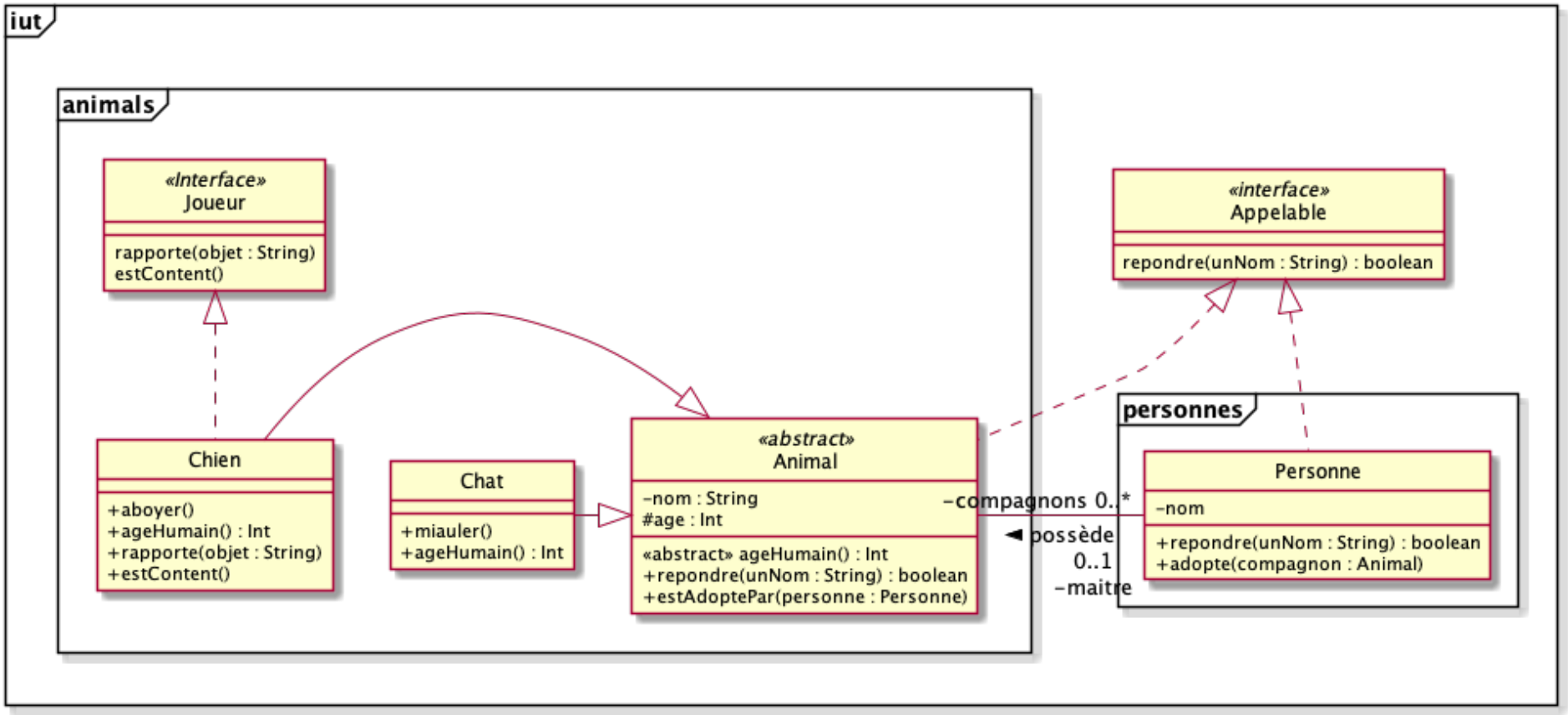


Diagramme de classes + paquetages UML



IUT Nantes

Packages et fichiers sources/compilés

```
src/
  + UtiliserAnimals.kt
  + iut/
      + Appelable.kt
      + animals/
      |    + Animal.kt
      |    + Chat.kt
      |    + Chien.kt
      |    + Joueur.kt
      + personnes/
           + Personne.kt
```

Les fichiers source `.kt` correspondant à des classes déclarées dans un package doivent reprendre le nom du package en terme de structuration en dossiers/sous-dossiers.

Les fichiers bytecode `.class` sont automatiquement compilés dans des dossiers/sous-dossiers reprenant le nom du package

```
bin/
  + UtiliserAnimalsKt.class
  + iut/
      + Appelable.class
      + animals/
      |    + Animal.class
      |    + Chat.class
      |    + Chien.class
      |    + Joueur.class
      + personnes/
           + Personne.class
```

IUT Nantes

Sommaire

IUT Nantes

Tableaux multidimensionnels

Un tableau à plusieurs dimensions (=une matrice) aura le type `Array< Array<X> >` : il s'agira en réalité d'un tableau de tableaux.

Exemple (tableau de 4 lignes x 3 colonnes)

```
var mat : Array< Array<Int?>>
// tab vide
mat = Array(4) { arrayOfNulls(3) }
```

ligne \ colonne	0	1	2
0	null	null	null
1	null	null	null
2	null	null	null
3	null	null	null

Comme pour un tableau simple,

- le nombre de lignes et de colonnes est fixe et ne peut pas être modifié
- Lignes et colonnes sont indicés de 0 à `size-1`
- un tableau vide contient `null` dans toutes les cases

IUT Nantes

Tableaux multidimensionnels (2)

L'accès à un élément se fait par `[..][..]`

```
mat = Array(4) { arrayOfNulls(3) }
mat[0][0] = 5
mat[2][1] = -4
mat[1][2] = 9
mat[3][2] = -1
mat[1][1] = mat[3][2]
```

ligne \ colonne	0	1	2
0	5	null	null
1	null	-1	9
2	null	-4	null
3	null	null	-1

Comme pour les tableaux simples, on peut déclarer des tableaux préremplis grâce à `arrayOf(...`

```
mat = arrayOf( arrayOf(2,3),
               arrayOf(-3,6),
               arrayOf(7,0) )
```

ligne \ colonne	0	1
0	2	3
1	-3	6
2	7	0

IUT Nantes

Parcours d'un tableau multidimensionnel

En utilisant un accès indicé

```
fun affiche(tab:Array<Array<Int?>>) {
  for (i in 0 until tab.size) {
      for (j in 0 until tab[i].size) {
          val element = tab[i][j]
          if (element != null)
            print("$element\t")
      else
          print("_\t")
      }
      println()
  }
}
```

Ou, plus simplement

```
fun affiche(tab:Array<Array<Int?>>) {
  for (ligne in tab) {
      for (element in ligne) {
        if (element != null)
          print("$element\t")
      else
          print("_\t")
      }
      println()
  }
}
```

IUT Nantes