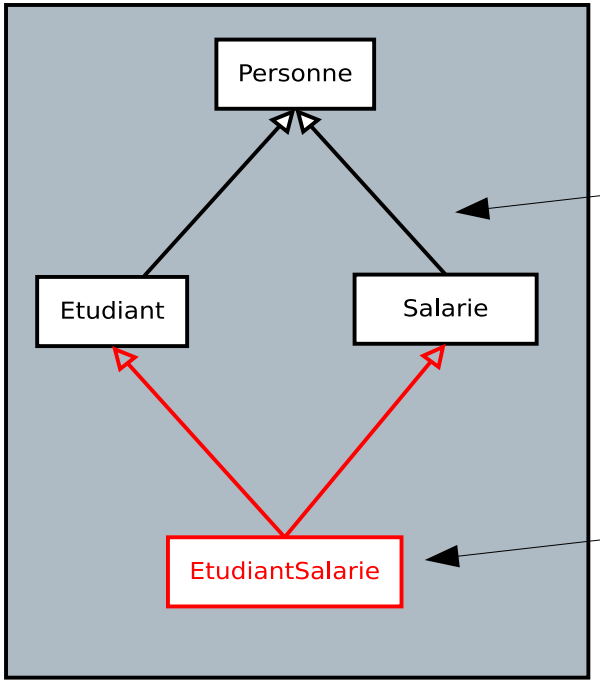
Introduction à Scala 3

Héritage, case Class, Trait, Object

Hanene AZZAG, Dominique BOUTHINON et Mustapha LEBBAH

Créer des sous classes

En scala l'héritage est simple :-)



Plusieurs classes peuvent hériter d'une même classe

mais

une même classe ne peut pas hériter de plusieurs classes

Créer un classe fille (1/2)

```
class Point(val x    : Double,
            val y    : Double)
```

ne pas mettre ***val*** devant ces arguments ! Ainsi, ils sont ***private[this]*** et ne serviront qu'à instancier les arguments *x* et *y* hérites de la classe mère

extends *Point* signale que cette classe hérite de la classe *Point*

Point(x1,y1) appelle le constructeur principal de la classe *Point* avec les arguments *x1* et *y1*

```
class PointColore(x1           : Double,
                  y1           : Double,
                  val couleur : String) extends Point(x1,y1)
```

nouvel attribut

Création d'un *PointColore*

appelle l'accesseur automatique héritée de *Point*

appelle l'accesseur automatique de *PointColore*

```
val pc = PointColore(5,10,"rouge")
println(pc.x)                    // affiche 5
println(pc.y)                    // affiche 10
println(pc.couleur)              // affiche "rouge"
```

Créer un classe fille (2/2)

```
class Point(val x    : Double,
            val y    : Double)
```

On peut utilise le même nom que pour les paramètres de la classe mère. Mais ne pas mettre ***val*** devant ces arguments ! Sinon on surcharge les arguments de la classe mère => erreur de compilation

```
class PointColore(val x           : Double,       // erreur compilation !!
                  val y           : Double,       // erreur compilation !!
                  val couleur : String) extends Point(x,y)
```

pas de **val** => correct : on a créé 2 valeurs **private[this]** différentes de celles héritées de la classe mère

```
class PointColore(x           : Double,       // ok
                  y           : Double,       // ok
                  val couleur : String) extends Point(x,y)
```

Classe fille avec constructeurs secondaires

```
class Point(val x    : Double,
            val y    : Double)
```

```
class PointColore(x           : Double,
                  y           : Double,
                  val couleur : String) extends Point(x,y) :

    def this(x : Double, y : Double) = this(x,y,"noir")
    def this() = this(0, 0)
```

appelle le constructeur secondaire précédent

appelle le constructeur principal

```
val PointColore pc1 = PointColore(5,10)
println(pc1)         // affiche : (5, 10) noire

val PointColore pc2 = PointColore()
println(pc2)         // affiche :  (0, 0) noire
```

Redéfinir une méthode de la classe mère

La méthode redéfinie masque la méthode héritée

Point redéfinissait déjà *toString* et *equals* héritée de *Any*

```
class Point(val x    : Double,
            val y    : Double) :

  override def toString = s"($x, $y)"
```

toString est redéfinie et masque *toString* héritée de *Point*

```
class PointColore(x           : Double,
                  y           : Double,
                  val couleur : String) extends Point(x,y) :

  override def toString = super.toString + " $couleur"
```

super pour appeler la méthode héritée masquée (sinon on réappelle *toString* de *PointColore*

super ne doit être utilisée que pour démasquer une méthode redéfinie
(pas pour appeler une méthode héritée non redéfinie)

Redéfinir la méthode équals de la classe mère (1/2)

Rappel de la méthode *equals* de la classe *Point*

```
class Point(val x    : Double,
            val y    : Double) :

    override def equals(arg: Any) =

        arg match

            case p:Point =>
                        this.x == p.x &&
                        this.y == p.y &&
                        this.getClass == p.getClass

            case _      =>
                        false
```

redéfinit le *equals* hérité de *Any*

si *arg* est un *Point* ou descendant
(*p* désigne le même objet que *arg*
mais en tant que *Point*)

retourne *true* s'il y a égalité des coordonnées...

...*p* et *this (objet courant)*
désignent des objets de la même classe

retourne *false* si *arg* n'est pas du type *Point*

Redéfinir la méthode *equals* de la classe mère (2/2)

Méthode *equals* de la classe *PointColore*

```
class PointColore(x    : Double,
                  y    : Double,
                  val couleur : String) extends Point(x,y) :

    override def equals(arg: Any) =

        arg match

            case p:PointColore =>
                        super.equals(p) &&
                        this.couleur == p.couleur


            case _      =>
                        false
```

si *arg* est un *PointColore* ou descendant (*p* désigne le même objet que *arg* mais en tant que *PointColore*)

appelle *equals* de la classe mère (pour vérifier l'égalité en tant que *Point*)

égalité des couleurs

inutile de vérifier si ***this*** et *p* désignent des objets de même classe, le test est fait par la classe mère

retourne *false* si *arg* n'est pas du type *PointColore*

Polymorphisme

Méthode polymorphe

Toute méthode redéfinie est polymorphe

```
class Point(val x    : Double,
            val y    : Double) :

    override def toString = s"($x, $y)"
```

toString est redéfinie donc polymorphe

```
class PointColore(x           : Double,
                  y           : Double,
                  val couleur : String) extends Point(x,y) :

  override def toString = super.toString + " $couleur"
```

tableau de *Point*

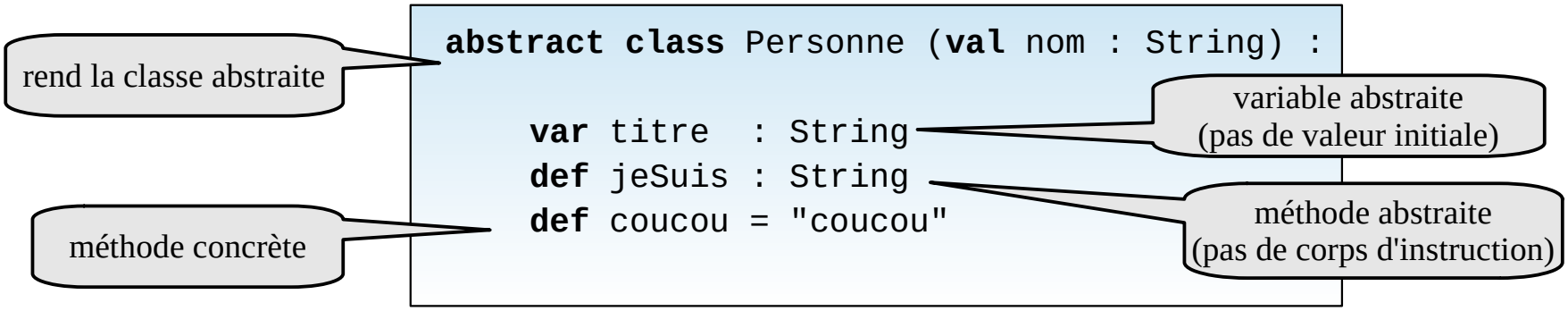
```
tPoints : Array[Point] = Array(Point(5,2), PointColore(10,3,"vert"))

for i <- 0 until tPoints.length do
    println(tPoints(i).toString())    // i = 0 affiche (5,2)
                                      // i = 1 affiche (10,3) vert
```

place un *PointColore*

Classe abstraite

Classe abstraite

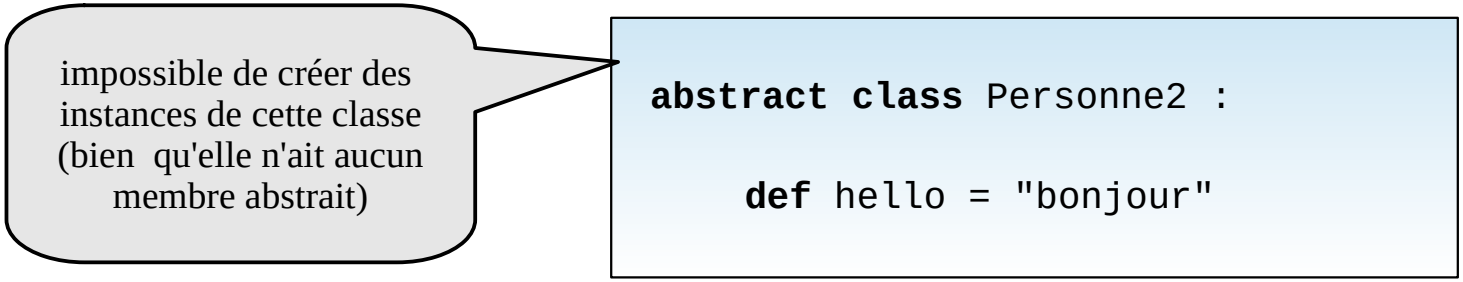
Une classe est obligatoirement abstraite dès qu'une variable ou constante est non initialisée ou qu'une méthode n'est pas définie



Impossible de créer des instances d'une classe abstraite

```
val p = Personne ("toto")          // !! erreur de compilation
```

On peut rendre une classe abstraite même si elle n'a aucun membre abstrait



Utilité des classes abstraites

Détenir des attributs et/ou des méthodes qui seront hérités (et si nécessaire instanciés) par des classes filles

```
class Personne(val nom : String) :

    var titre : String
    def jeSuis : String
```

les membres abstraits ont été instanciés

```
class Medecin(nom : String)
      extends Personne(nom) :

    var titre  = "Dr."
    def jeSuis = s"$nom, médecin"
```

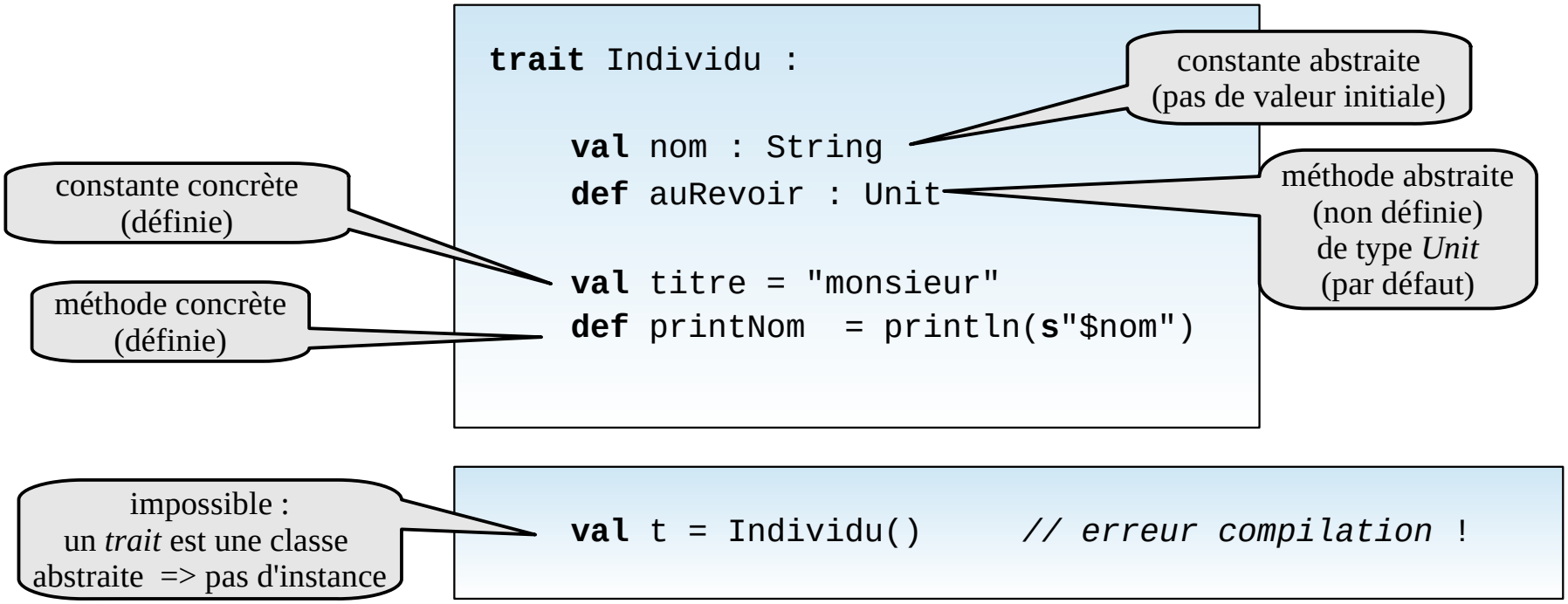
```
class Professeur(nom : String)
      extends Personne(nom) :

    var titre = "Pr."
    def jeSuis = s"$nom, professeur"
```

```
t: new Array[Personne] = Array(Medecin("jean"), Professeur("Fred"))

for i <- 0 until t.length do      // i = 0 : "Jean, médecine"
    t(i).jeSuis                   // i = 1 : "Fred, professeur"
```

Polymorphisme sur la méthode *jeSuis*

Trait

Un trait est une classe abstraite sans membre private



Un trait est semblable à une interface java (en moins restrictif)

Ajouter un trait à une classe

signale que la classe *Personne* hérite du trait *Individu*
Si la classe n'est pas **abstract** elle doit définir tous les membres abstraits dont elle hérite

```
trait Individu :

    val nom : String
    def auRevoir : Unit
    val titre     = "Monsieur"
    def printNom  = println(s"$nom")
}
```

```
class Personne extends Individu :


    val nom = "Pierre"


    def auRevoir = println("au revoir")
```

définition de la constante abstraite héritée *nom*

définition de la méthode abstraite héritée *auRevoir*

```
val p = Personne()
p.printNom       // affiche "Pierre"
p.auRevoir       // affiche "au revoir"
println(p.titre) // affiche "Monsieur"
```

Initialiser une variable ou constante d'un trait

```
trait Individu :

    val nom : String
```

définition de la constante abstraite héritée **val** *nom* par un paramètre

définition de la constante abstraite héritée **val** *nom* par une constante dans le corps de la classe

```
class Personne(val nom:String)
               extends Individu
```

```
class Personne extends Individu :

    val nom = "Jean"
```

```
val p1 = Personne("Jean")
println(p1.nom)  // affiche "Jean"

val p2 = Personne("Pierre")
println(p2.nom)  // affiche "Pierre"
```

```
val p1 = Personne()
println(p1.nom)  // affiche "Jean"

val p2 = Personne()
println(p2.nom)  // affiche "Jean" !!
```

attention ! chaque objet de type *Personne* aura le même nom

Ajouter un trait à un objet

On choisit les objets qui bénéficient du trait

```
trait Nourriture :

    def manger = println("je mange)
```

méthode définie (implicitement de type *Unit*)

```
class Personne(val nom:String)
```

seul l'objet *Personne* créé hérite du trait *Nourriture*
(pour les objets on utilise toujours **with**)

```
val p1 = Personne("Jean") with Nourriture
p1.manger                          // affiche "je mange"

val p2 = Personne("Pierre")
p2.manger                          // erreur compilation
```

la *Personne* créée n'hérite pas du trait *Nourriture*

Trait avec paramètres

```
trait commande(val quantite : Int)
```

```
class Achat(q : Int) extends commande(q) :

    def afficheAchat =
        println("achat : $quantite")
```

la classe *Achat* hérite de l'attribut *quantite*

```
val a =  Achat(50)

a.afficheAchat            // affiche : "achat : 50 "
```

Pourquoi faire des traits ?

Créer un interface (style java)

2 implémentations différentes du trait

```
trait LecteurDVD :
    def demarrer()
    def stopper()
```

```
class Sony extends LecteurDVD :

  def demarrer() =
      print("je demarre vite")

  def stopper() =
      print("je stoppe, j'éteins")
```

```
class  Samsung extends LecteurDVD :

  def demarrer() =
      print("je demarre doucement")

  def stopper() =
      print("je stoppe")
```

méthode ayant un *LecteurDVD* (trait) en argument
On peut donc lui fournir n'importe quelle implémentation du trait

```
def f(lec : LecteurDVD) = lec.demarrer()

lec1 = Sony()
lec2 = Samsung()
f(lec1)      // affiche "je demarre vite"
f(lec2)      // affiche "je demarre doucement"
```

Faire du pseudo héritage multiple

Une classe ou un objet peut utiliser plusieurs traits

```
trait Nourriture :

    def manger = print("je mange")
```

```
trait Sommeil :

    def dormir = print("je dors")
```

```
trait Respiration :

    def respirer = print("je respire")
```

Le premier trait est précédé de **extends**
les autres de **with**

```
class Personne(val nom:String)
        extends Nourriture
        with Sommeil
        with Respiration
```

```
val p = Personne("Jean")
p.manger     // affiche "je mange"
p.dormir     // affiche "je dors"
p.respirer   // affiche "je respire"
```

Une classe (ou objet) utilisant des traits est du type de chacun de ses traits

```
class Personne(val nom:String)
                extends Nourriture
                with Sommeil
                with Respiration
```

```
def f(x : Nourriture) = ...

def g(y : Sommeil) = ...

def h(z : Respiration) = ...
```

3 méthodes, chacune ayant un argument de type différent

```
val p = Personne("Jean")
f(p)          // ok
g(p)          // ok
h(p)          // ok
```

La *Personne p* est aussi du type *Nourriture*, *Sommeil* et *Respiration*

Conflit sur un membre commun à plusieurs traits

C'est le dernier trait listé qui l'emporte

```
trait A :

    def  salut = print("bonjour")
```

```
trait B :

    def salut = print("hello ")
```

```
val p = Personne("Jean") with A with B

p.salut      // affiche "hello"
```

salut de B est choisi

Héritage multiple entre traits

Un trait peut hériter de plusieurs traits

```
trait A :

    def ok     = print("ok")
    def  salut = print("bonjour")
```

```
trait B :

    def ouf   = print("ouf")
    def salut = print("hello ")
```

```
trait C extends A with B :

    override salut = super[A].salut
```

le trait *C* hérite de *A* et *B*

la méthode *salut* est redéfinit
on appelle explicitement *salut* du trait *A*

Dans une méthode redéfinit on peut spécifier le trait parent choisi

case class

case Class

Une *case Class* est une classe immutable :
les objets qu'elle créé ne peuvent être modifiés

les attributs sont par défaut publics et immutables (constants)
pas besoin de mettre **val**

```
case class Personne (nom : String, age : Int)
```

```
val p = Personne ("Jean", 15)
p.nom = "olivier"                    // erreur de compilation
                                     // l'attribut nom n'est pas modifiable
```

génération automatique de méthodes

lors de la création d'un objet d'une case Class scala génère automatiquement :

- une méthode *copy* (pour copier un objet)
- une méthode *equals* (pour comparer les objets)
- une méthode *toString* (pour décrire l'objet sous forme textuel)

```
val p1 = Personne("Jean", 15)

println(p1)                              // affiche Personne("jean", 15)

val p2 = Personne("Olivier", 20)

if p1 == p2                              // p1 == p2 <=> p1.equals(p2)
    then println("vrai")
    else println("faux")                 // affiche faux

val p3 = p1.copy
```

case Class et pattern matching

On peut récupérer les arguments des constructeurs dans un match..case

```
case class Personne (nom : String, age : Int)
```

```
val p = Personne ("Jean", 15)

p match
    case Personne(n, a) => println(s"nom : $n, age : $a)
                                        // affiche "nom : jean, age : 15
```

n et *a* sont les arguments données au constructeur lors de la création de la personne pointée par *p*
(si *p* ne pointe pas une *Personne* le case n'est pas activé)

Remarques

Une case class est

- très facile à créer
- sûre
- particulièrement bien adaptée au pattern matching
- très utile en programmation fonctionnelle (vue plus tard)

Dans la mesure du possible privilégier les case class

**(attention : il n'y a pas d'héritage entre case class,
mais une classe peut hériter d'une case class)**

Object

Créer un singleton (l'instance unique d'un classe)

Un *object* (singleton) est utilisé pour détenir des attributs et méthodes (statiques en java) qui ne concernent que cet objet

object signale qu'on définit un singleton (un objet unique contenant des variables, constantes et/ou méthodes)

un **object** n'a pas de constructeur (il est initialisé par ses déclarations internes)

méthode pour afficher un tableau de *Int*

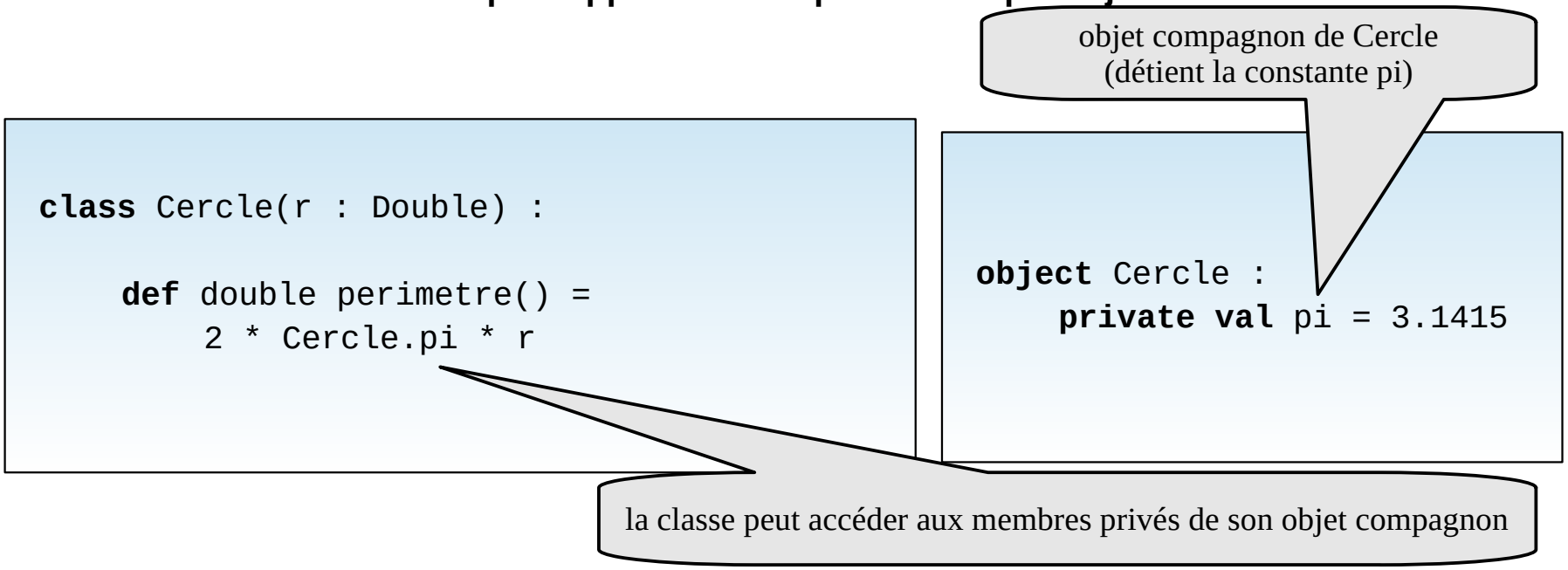
```
object tableau :

    def printTab(t : Array[Int]) =
        for elt ← t do
            println(elt)
```

```
t1 = Array[Int](5,7,9)

tableau.printTab(t1)
```

appelle de la méthode *printTab* de l'**object** tableau

Objet compagnon

Un *object* du même nom qu'une classe, est conçu pour détenir des attributs et méthodes (statiques en java) qui n'ont pas besoin des attributs de la classe, et qui n'appartiennent qu'à cet unique objet



L'objet compagnon et la classe doivent être dans le même fichier