R3.04 : Qualité de développement

Rappels Kotlin (1)

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

Département informatique

Kotlin

- Langage de programmation orienté objet **et** fonctionnel
- Développé à partir de 2010 par JetBrains et de nombreux autres contributeurs (complètement open-source)
- 100 % interopérable avec Java
  - Langage compilé : le bytecode (entre autre)
  - Machine virtuelle : la JVM – Java Virtual Machine
  - Multiplatforme
- Philosophie : *"plus concis, plus pragmatique, plus sûr que Java"*
- Langage "fortement recommandé" par Google pour le développement Android à partir de 2019
- Kotlin également compatible avec *Javascript (JS)*[1], du code natif, etc.
- `https://kotlinlang.org/`

vs.

IUT Nantes

1. rien à voir avec Java

Sommaire

IUT Nantes

Les types primitifs

Type	occ. mém. (bits)	min	max
Les nombres entiers			
`Byte`	8	-128	127
`Short`	16	-32_768	32_767
`Int`	32	-2_147_483_648	2_147_483_647
`Long`	64	-9_223_372_036_854_775_808	9_223_372_036_854_775_807
Les nombres flottants			
`Float`	32		
`Double`	64		
Les caractères			
`Char`	16		
`String`	variable	= séquence de caractères [2]	
Les booléens			
`Boolean`	8	`true` (vrai) ou `false` (faux)	

IUT Nantes

2. On y reviendra

Les variables `var` ou `val`

```
val monNom : "Arnaud Lanoix"
var monAge : Int = 42
```

En Kotlin, on manipule deux sortes de variables :

- Des variables classiques, dite muables, grâce à `var` pour "variable"
- Des variables immuables, grâce à `val` pour "valeur", c-à-d des variables non-modifiables, une fois initialisées (= "en lecture seulement")

Indiquer le type d'une variable n'est pas forcément nécessaire : le compilateur déduit automatiquement le type des variables, quand c'est possible.

IUT Nantes

La condition `if... else...`

```
if (cptAbs >= 5) {
  println("Echec($cptAbs abs)")
}
else if (cptAbs == 4) {
  println("Alerte rouge($cptAbs Abs)")
  println("* alerter tuteur *")
}
else if (cptAbs in 1..3)
  println("Attention($cptAbs abs)")
else
  println("Pas d'absence")
```

```
var max = if (a >= b) {
    println("$a plus grand que $b")
    a // la derniere instruction
    //du bloc est retournee
}
else if (a <b) {
    println("$a plus petit que $b")
    b
}
else b
```

- On peut imbriquer les `if... else...`
- On peut se passer des `{...}` si le bloc d'instructions ne contient qu'une instruction
- `if... else...` avec retour de valeur

IUT Nantes

La condition `when...`

= pour simplifier l'imbrication des `if... else...`

```
    when {
      cptAbs >= 5 -> println("Echec ($cptAbs abs)")
      cptAbs == 4 -> {
            println("Alerte rouge ($cptAbs abs)")
            println("* alerter tuteur *")
      }
      cptAbs in 1..3 -> println("Attention ($cptAbs abs)")
      else -> println("Pas d'absence")
    }
```

On peut préciser sur quelle variable porte le `when`

\+ `when...` avec retour de valeur

```
  msg = when (cptAbs) {
    in 5.. Int.MAX_VALUE -> "Echec ($cptAbs abs)"
    4 -> {
          println("Alerte rouge ($cptAbs abs)")
          "* alerter tuteur *"
    }
    in 1..3 -> "Attention ($cptAbs abs)"
    else -> "Pas d'absence"
  }
```

IUT Nantes

La boucle `while`

```
var cptRebourd = 10
println("Depart dans...")

while (cptRebourd >= 0) {
  println(cptRebourd)
  cptRebourd--
}

println("Go !!!")
```

- Attention aux boucles infinies
- les boucles `while` sont à utiliser quand on ne peut pas "prévoir" le nombre d'itérations
- dans l'exemple, on devrait (plutôt) utiliser une boucle `for`

IUT Nantes

La boucle `for`

```
println("Depart dans...")
for (cpt in 10 downTo 0) {
  println(cpt)
}
```

```
println("Depart a 10...")
for (cpt in 0 until 10  step 2) {
  println(cpt)
}
```

On doit préciser

- la variable d'itération ; ici `cpt`
- la valeur "initiale"
- l'ordre d'itération : décrémental `downTo` ou incrémental `until`
- la valeur "limite" incluse si `downTo`, exclue si `until`
- le pas d'itération : `step`, facultatif si `step = 1`

```
println("Depart a 10...")
for (cpt in 0..10 step 1) {
  println(cpt)
}
```

- Dans ce cas, la valeur "limite" est incluse

IUT Nantes

Les fonctions

Une fonction est définie par :

- le mot-clef `fun`
- un nom
- (éventuellement) des paramètres et leurs types
- (éventuellement) un résultat typé et renvoyé (`return`)

```
fun mult(a : Int, b : Double = 1.5, c : Double) : Double {
  var resultat = a * b * c
  return resultat
}
```

- Les paramètres sont immuables
- Les paramètres peuvent avoir des valeurs par défaut
- A l'appel d'une fonction, on peut nommer les paramètres et modifier l'ordre d'appel
- Ecriture raccourcie :

```
fun mult2(a : Int, b : Double, c : Double) = a * b * c
```

IUT Nantes

Sommaire

IUT Nantes

Qu'est-ce qu'une classe ?

= sorte de "moule" [3] pour définir des objets, qui précise

- les propriétés qui définissent la structure interne des objets (= les attributs)
- Les interactions qu'offrent les objets, les comportements possible pour les objets (= les méthodes)
- Les (éventuels) liens d'héritage
- ...

La classe **Citoyen**

- nom, prénom, date de naissance,
- numéro carte d'identité,
- photo,
- signature,
- ...

3. "patron", "plan"", "schéma de construction", ...

Instancier une classe = créer un objet

1. Créer des objets à partir d'une classe (= instanciation)
   = valuer les attributs définis dans la classe
2. Interagir avec les objets créés
   = appeler les méthodes définies dans la classe dans le contexte de l'objet

IUT Nantes

Instancier une classe = créer un objet

1. Créer des objets à partir d'une classe (= instanciation)
   = valuer les attributs définis dans la classe
2. Interagir avec les objets créés
   = appeler les méthodes définies dans la classe dans le contexte de l'objet

Une citoyenne précise : Corinne B.

Un objet est une **instance** d'une classe

Déclarer une classe en Kotlin

```kotlin
class Chien {
  var nom :String = ""
  var age : Int = 0          // en mois
  var race : String = ""
  var poids : Double = 0.0 // en kg

  fun aboyer() {
    println("$nom dit : ouaf !!!")
  }

  fun renommer(nouveauNom : String) {
    nom  = nouveauNom
  }

  // @param distance en m
  fun courir(distance : Int) {
    // le chien perd 1 g / km
    poids -= (distance / 1000.0) / 1000
  }

  fun ageEnAnnee() = age / 12.0
}
```

- Une classe est déclarée grâce au mot-clef `class`
- Les attributs sont déclarés comme des variables internes à la classe
  - Les attributs doivent être initialisés
- Les méthodes sont déclarées comme des fonctions internes à la classe
  - On peut consulter ou modifier les valeurs des attributs via les méthodes

IUT Nantes

Constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {
  var nom :String
  var age : Int = 1       // en nb de mois
  val race : String
  var poids : Double  // en kg

  init {
      nom = monNom
      this.race = race
      this.poids = if (poids > 0.0) poids else 0.1
  }
```

- les paramètres du constructeur sont déclarés après le nom de la classe
- On peut définir des valeurs par défaut pour les paramètres
- tous les attributs ne sont pas forcément présents comme paramètres
- les paramètres du constructeur servent à initialiser les attributs
- tous les attributs doivent être initialisés

Attribut immuable

Notez qu'ici l'attribut `race` est `val` : pour un chien donné ne doit plus pouvoir être changé : il est impossible de changer sa race après sa création

Constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {
  var nom :String
  var age : Int = 1       // en nb de mois
  val race : String
  var poids : Double  // en kg

  init {
      nom = monNom
      this.race = race
      this.poids = if (poids > 0.0) poids else 0.1
  }
```

- les paramètres du constructeur sont déclarés après le nom de la classe
- On peut définir des valeurs par défaut pour les paramètres
- tous les attributs ne sont pas forcément présents comme paramètres
- les paramètres du constructeur servent à initialiser les attributs
- tous les attributs doivent être initialisés

Attribut immuable

Notez qu'ici l'attribut `race` est `val` : pour un chien donné ne doit plus pouvoir être changé : il est impossible de changer sa race après sa création

Visibilités en Kotlin : exemple

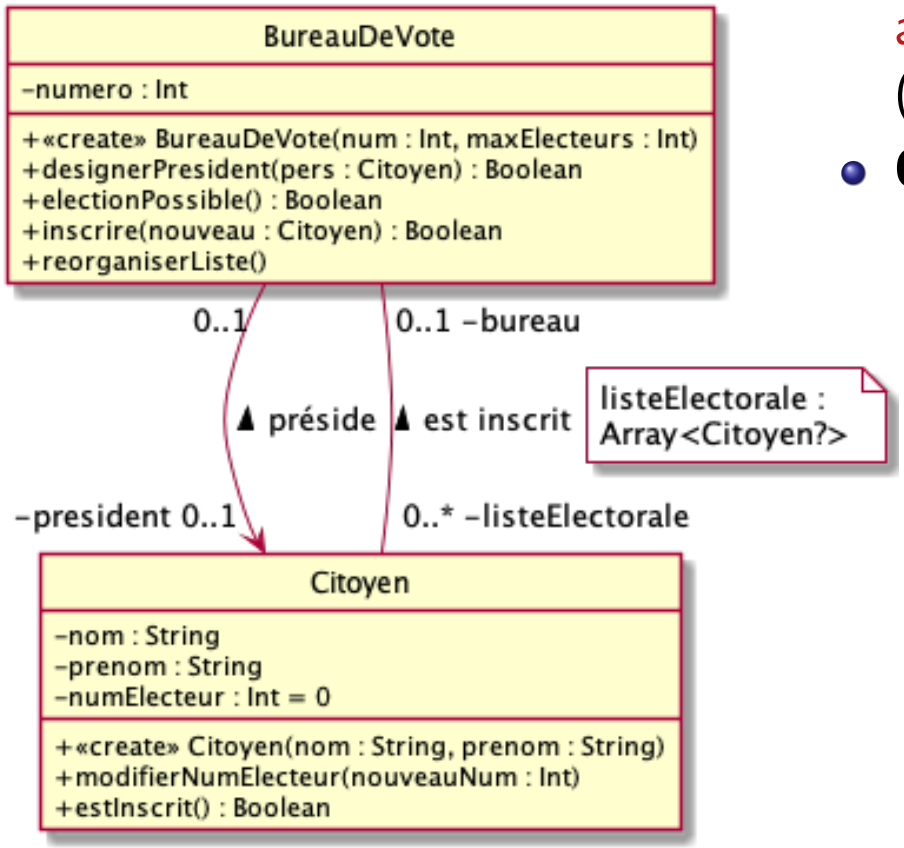
```kotlin
class Chien (...) {
  var nom :String
  private var age : Int
  val race : String
  var poids : Double
    private set

  fun courir(dist : Int) {
    poids -= poidsEnMoins(dist)
  }

  private fun poidsEnMoins(d : Int)
    = (d / 1000.0) / 1000
}
```

```kotlin
println("${rogue.nom}")
rogue.nom = "Severus"
println("${rogue.age}")      // error
  // it is private in 'Chien'
rogue.age = 10               // error
  // it is private in 'Chien'
println("${rogue.race}")
rogue.race = "Serpentard"    // error
  // val cannot be reassigned
println("${rogue.poids}")
rogue.poids = 42.0           // error
  // the setter is private in 'Chien'
rogue.courir(100)
rogue.poidsEnMoins(100)      // error
  // it is private in 'Chien'
```

- L'attribut `nom` est `public` (par défaut) : accessible en lecture/écriture
- L'attribut `age` est `private` : aucun accès possible
- L'attribut `race` est `public`, mais immuable : accessible en lecture
- L'attribut `poids` est restreint en écriture : accessible en lecture
- La fonction `courir()` est `public` (par défaut) : accessible
- La fonction `poidsEnMoins()` est `private` : aucun accès possible

IUT Nantes

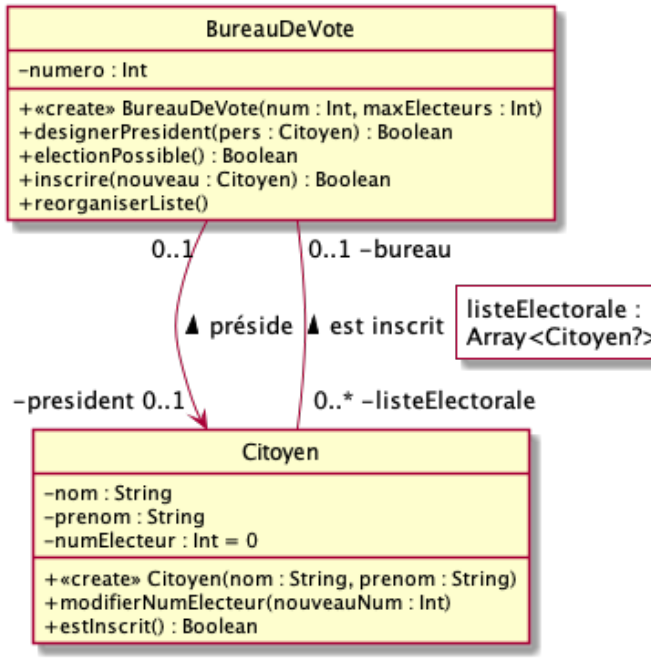
d'UML à Kotlin : processus systématique



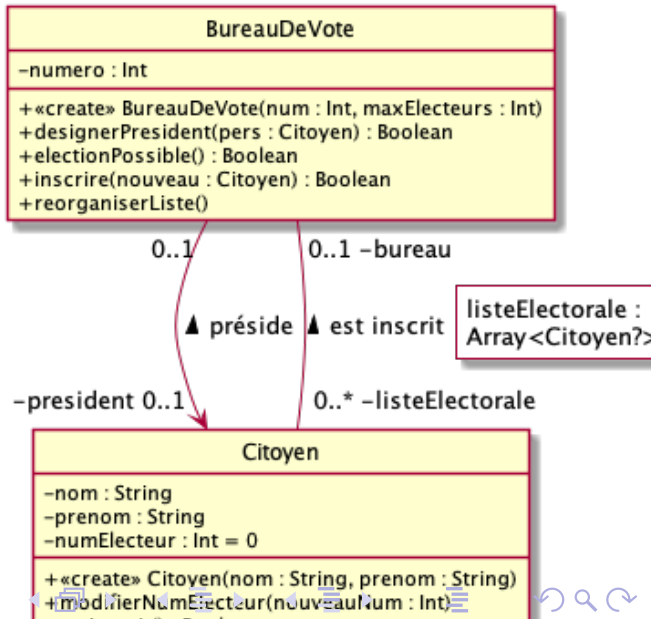
- Simple **ré-écriture** pour les classes, les attributs (type, visiblité), les méthodes (signature, visibilité)
- **Quid des associations ?**
  - Une association unidirectionnelle devient un attribut dans la classe "source"
  - Une association bidirectionnelle devient deux attributs, un de chaque coté de l'association
  - Les rôles deviennent les noms des attributs
  - (ajouter des méthodes pour mettre à jour les nouveaux attributs)
  - Les cardinalités 0..1 donnent des variables nullable ?
  - Les cardinalités O..*, 1..* ou * donnent des `Array?<X>` (ou d'autres collections)

IUT Nantes

```
class Citoyen (nom : String, prenom : String) {

    private var nom : String
    private var prenom : String
    private var numElecteur : Int
    private var bureau : BureauDeVote?

    init {
        this.nom = nom
        this.prenom = prenom
        this.numElecteur = 0
        this.bureau = null
    }

    fun modifierNumElecteur(nouveauNum : Int) {
        numElecteur = nouveauNum
    }

    fun modifierBureauDeVote(nouveauBureau : BureauDeVote) {
        bureau = nouveauBureau
    }

    fun estInscrit() = (bureau != null)
```



```
class BureauDeVote (num : Int, maxElecteurs : Int) {

    private val numero : Int
    private var president : Citoyen?
    private val listeElectorale : Array<Citoyen?>

    init {
      numero   = num
      president = null
      listeElectorale = arrayOfNulls<Citoyen>(maxElecteurs)
    }

    fun designerPresident(pers : Citoyen) =
      if (pers.estInscrit()) {
        president = pers
        true
      }
      else false

    fun electionPossible() =
      (president != null  // PAS CORRECT
       && listeElectorale.isNotEmpty())
```



Sommaire

IUT Nantes

Les variables sont des références

- En Kotlin, toutes les variables sont des références (dans la pile mémoire) qui "pointent" vers leur valeur (dans le tas mémoire)
- Une référence en Kotlin correspond à un pointeur en C/C++, avec une gestion simplifiée de l'allocation mémoire :
  - On ne s'occupe pas de réserver de l'espace mémoire
  - On ne gère pas non plus la libération de cet espace : le *Garbage Collector* (=ramasse-miette) s'occupe de libérer l'espace occupé par des objets déréférencés

IUT Nantes

Les variables sont des références

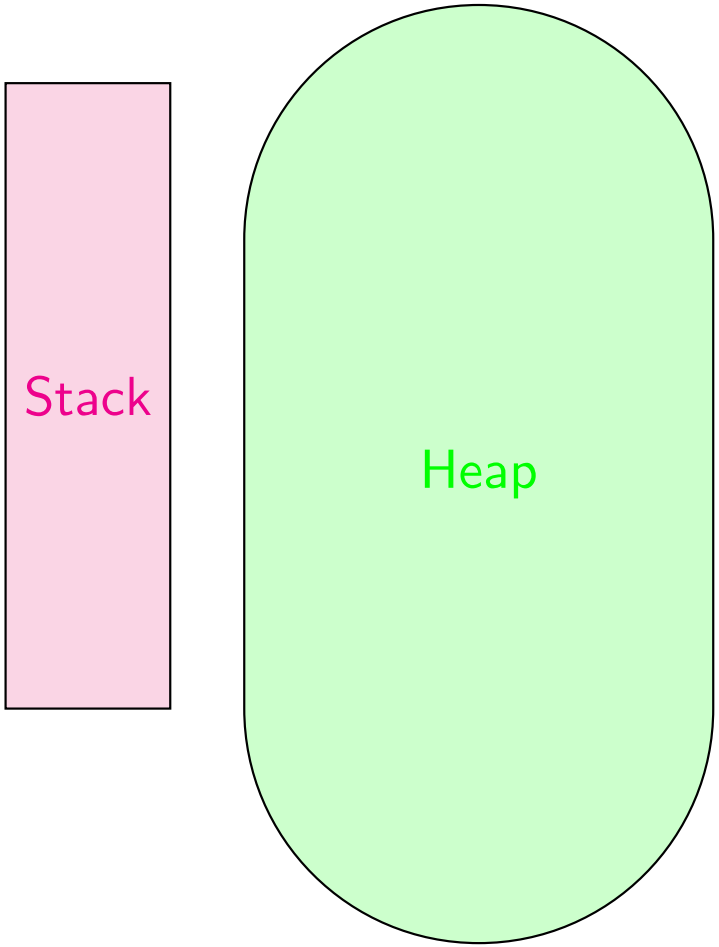
- En Kotlin, toutes les variables sont des références (dans la pile mémoire) qui "pointent" vers leur valeur (dans le tas mémoire)
- Une référence en Kotlin correspond à un pointeur en C/C++, avec une gestion simplifiée de l'allocation mémoire :
  - On ne s'occupe pas de réserver de l'espace mémoire
  - On ne gère pas non plus la libération de cet espace : le *Garbage Collector* (=ramasse-miette) s'occupe de libérer l'espace occupé par des objets déréférencés

L'opérateur d'identité `===`

L'opérateur `===` (3 x `=`) permet de vérifier que deux objets ont la même référence

- L'opérateur d'égalité `==` (2 x `=`) regarde l'égalité (des "valeurs")
- `===` implique `==` mais la réciproque n'est pas vraie

Les variables sont des références : exemple de `String`

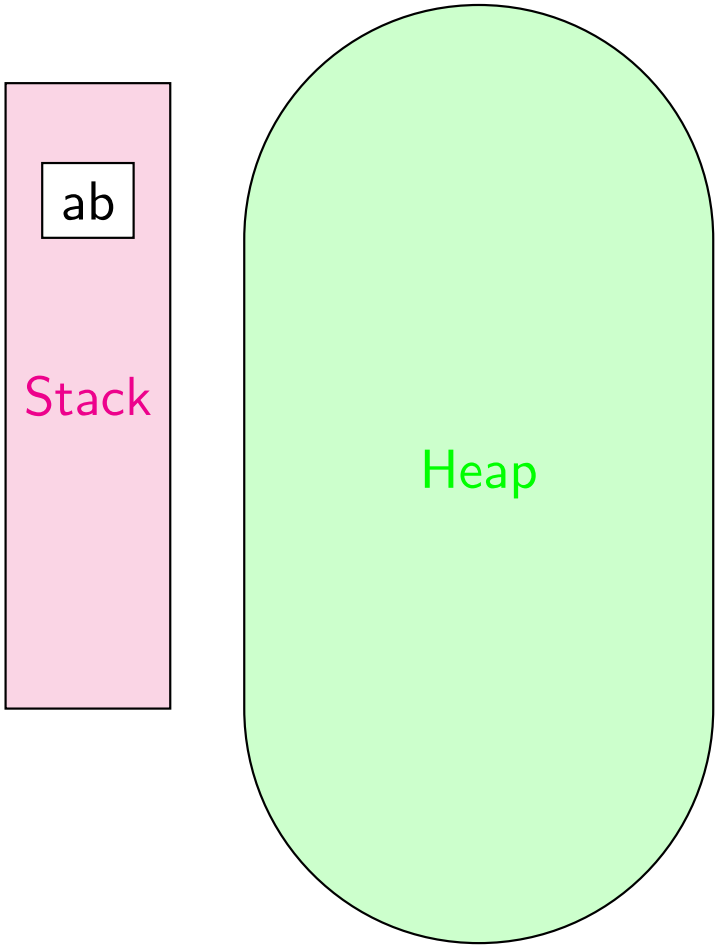



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Schématisation de la mémoire de la JVM

IUT Nantes

Les variables sont des références : exemple de `String`

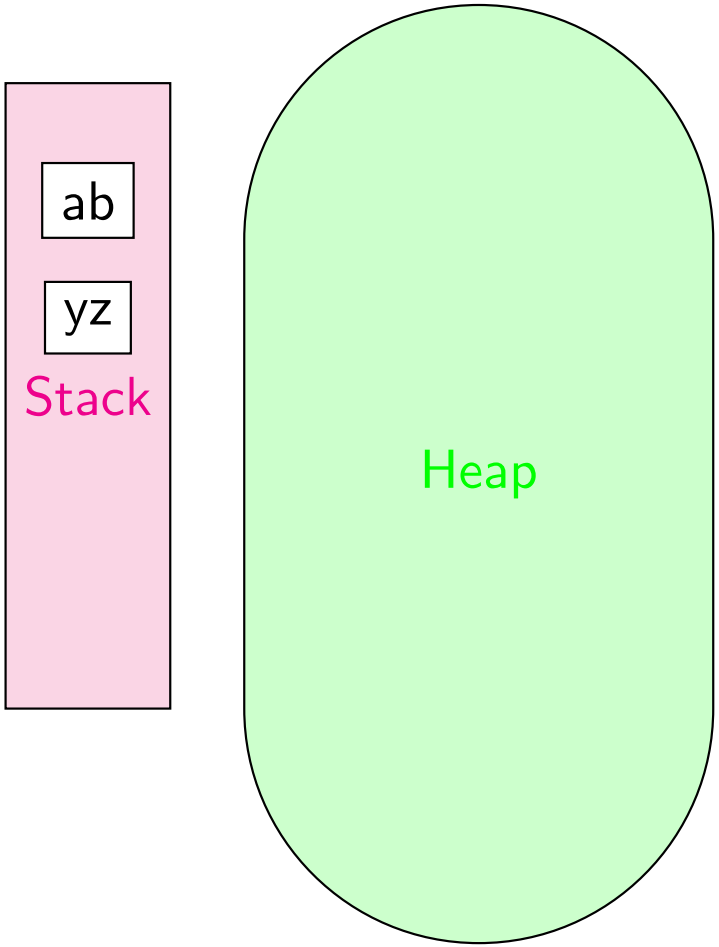



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`ab` créé dans la pile mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

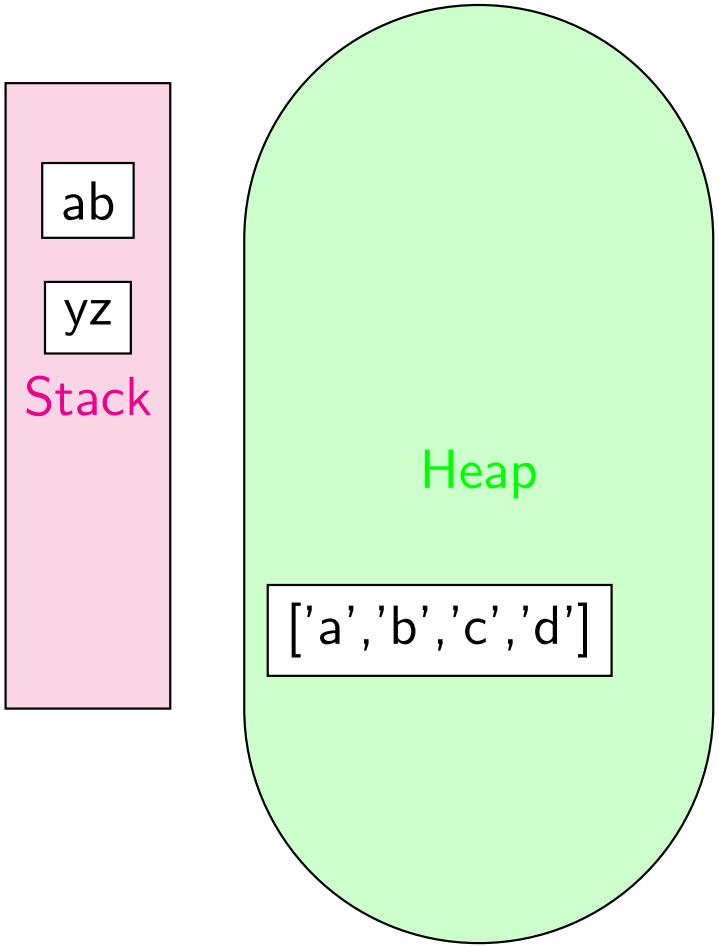



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`yz` créé dans la pile mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

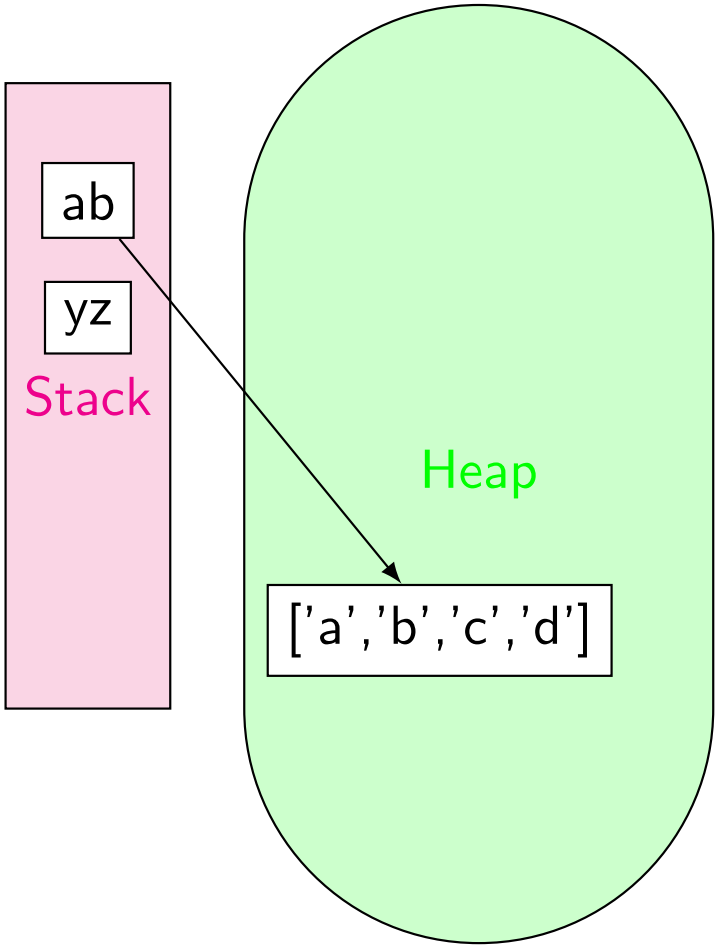



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`"abcd"` créé dans le tas mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

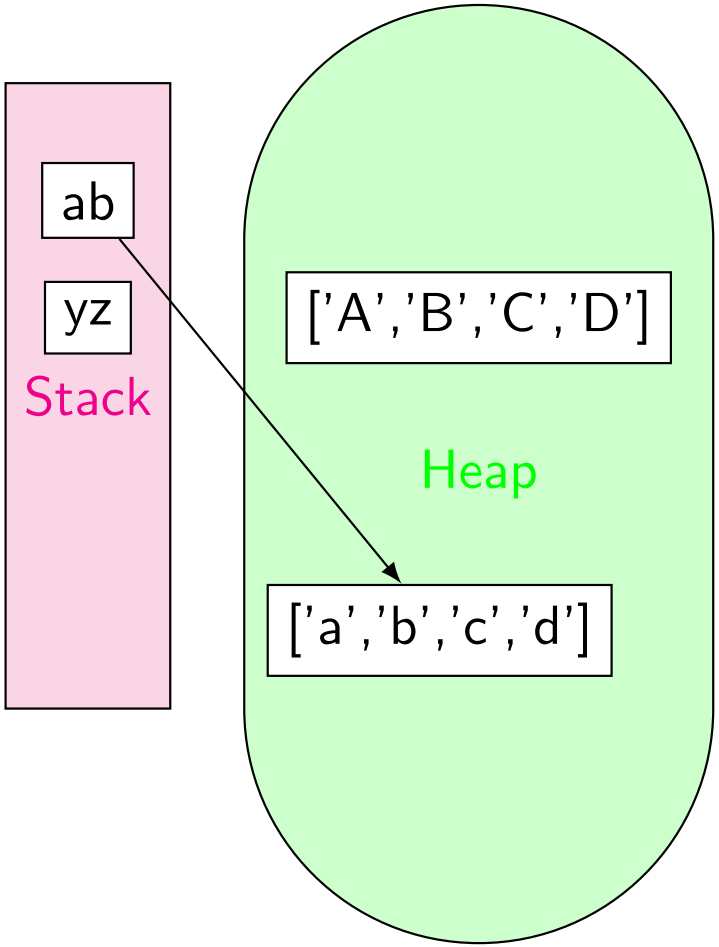



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`ab` "pointe" vers `"abcd"`

IUT Nantes

Les variables sont des références : exemple de `String`

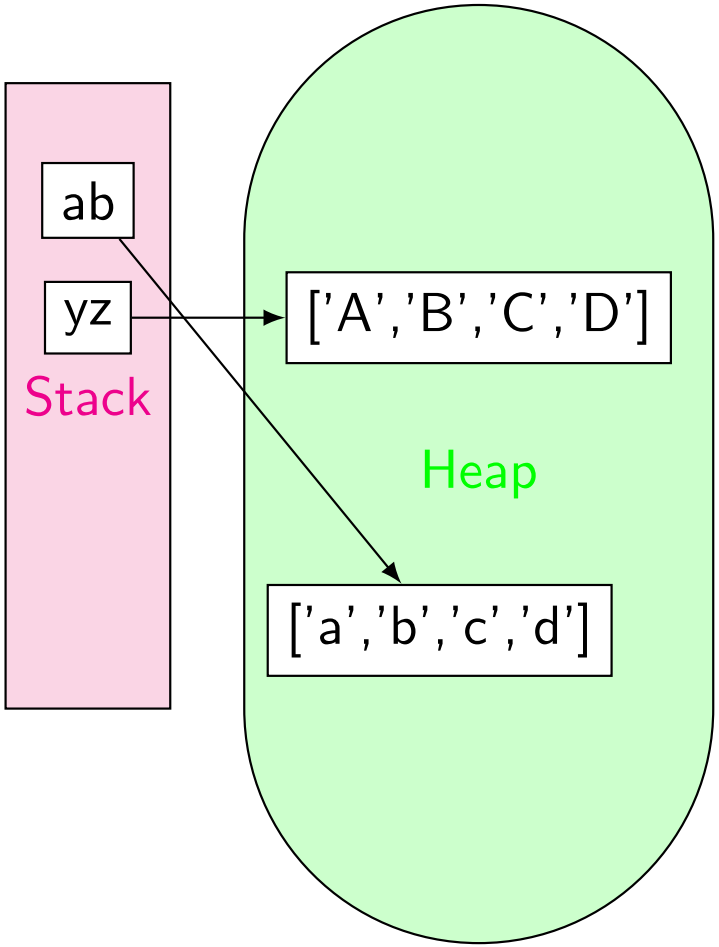



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`"ABCD"` créé dans le tas mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

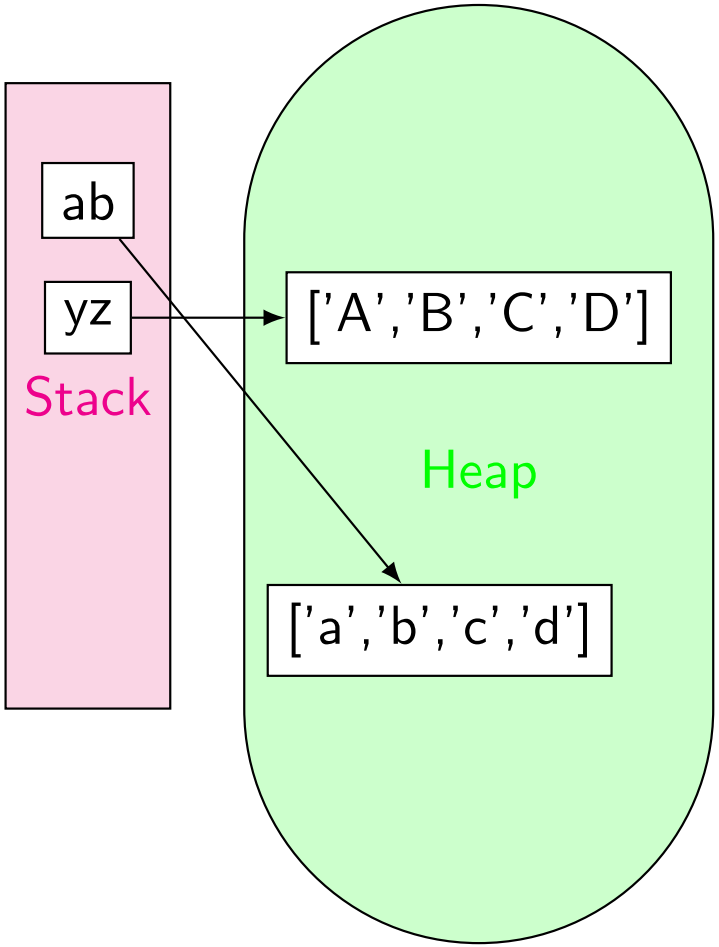



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`yz` "pointe" vers `"ABCD"`

IUT Nantes

Les variables sont des références : exemple de `String`

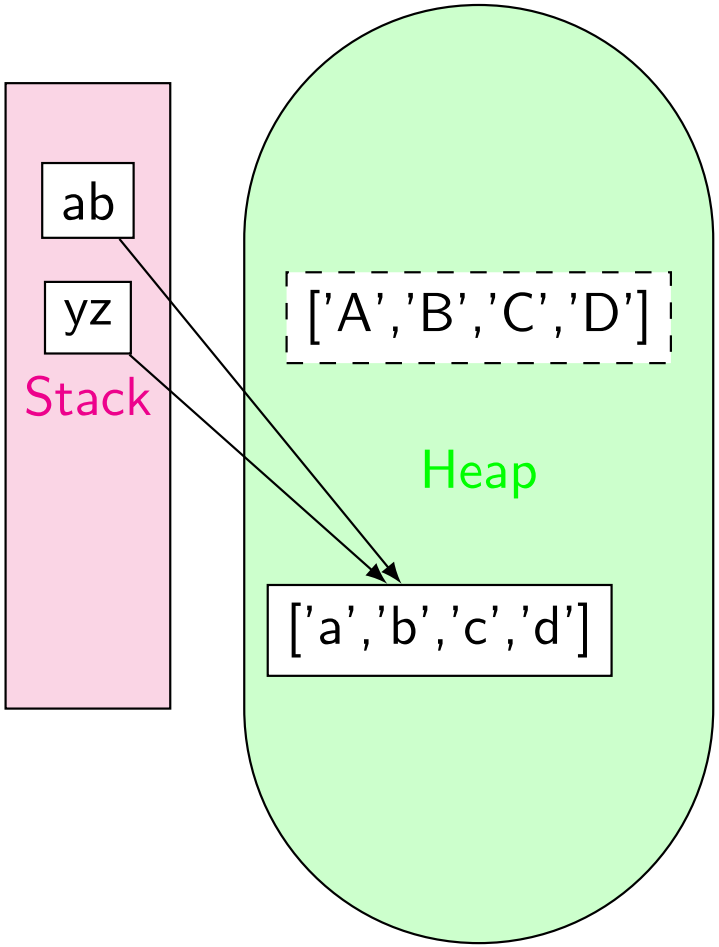



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Les valeurs de `ab` et de `yz` sont $\neq$,
leurs références aussi

IUT Nantes

Les variables sont des références : exemple de `String`

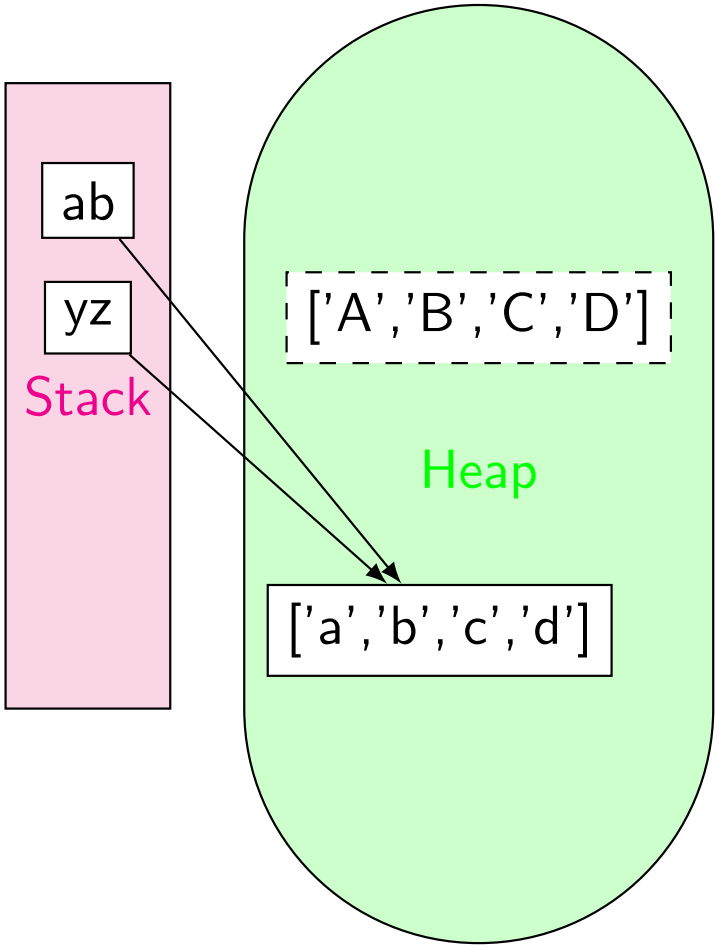



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`yz` et `ab` "pointent" vers la même valeur

IUT Nantes

Les variables sont des références : exemple de `String`

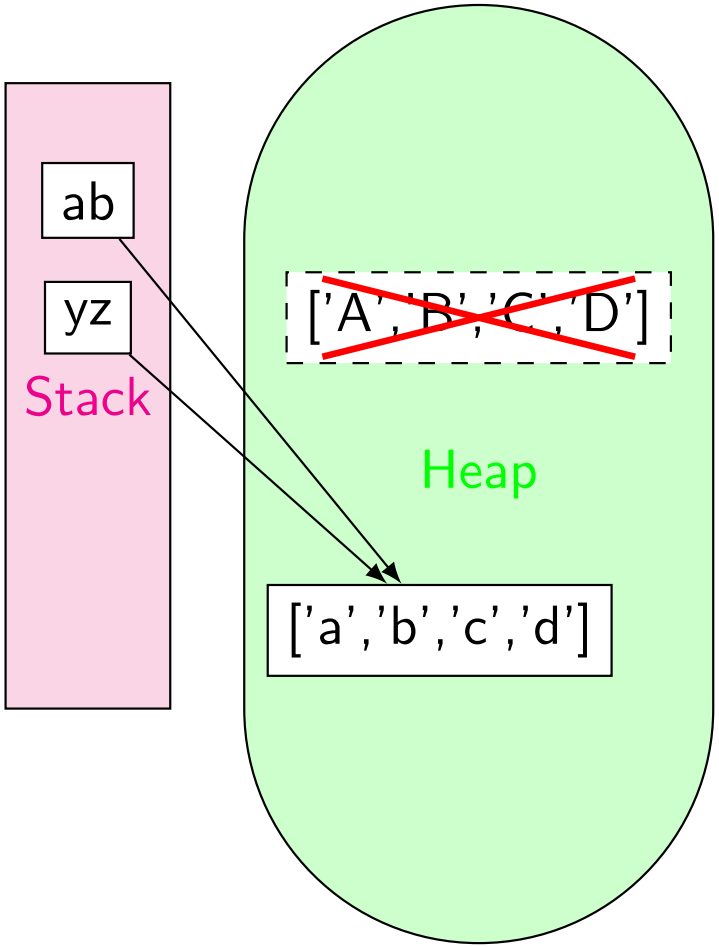



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Les valeurs de `ab` et de `yz` sont $=$
puisque leurs références sont $=$

IUT Nantes

Les variables sont des références : exemple de `String`

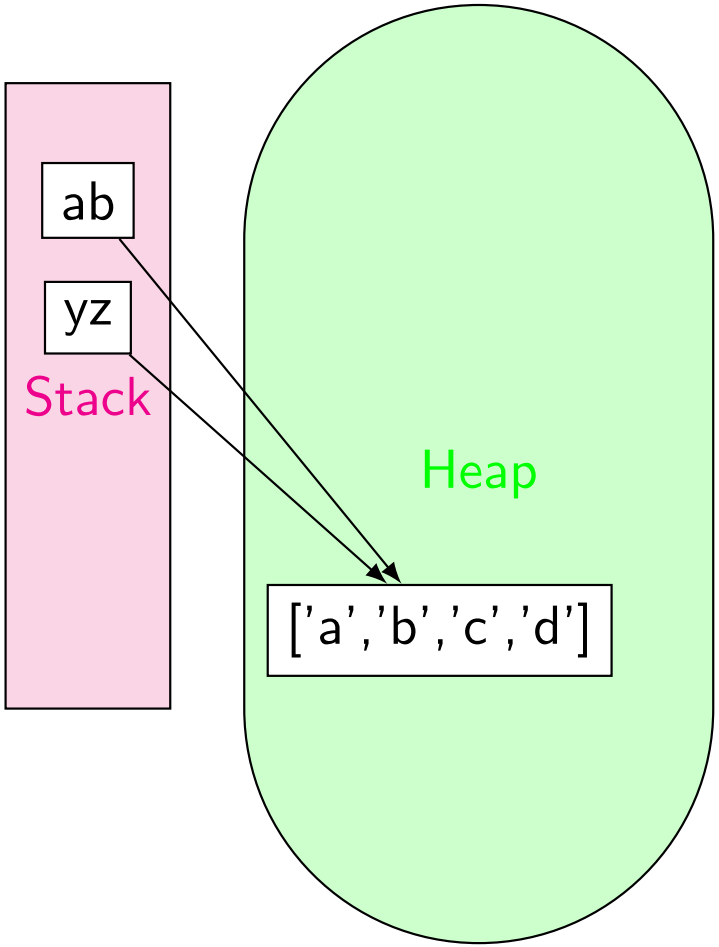



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Le garbage collector efface les objets déréférencés

IUT Nantes

Les variables sont des références : exemple de `String`

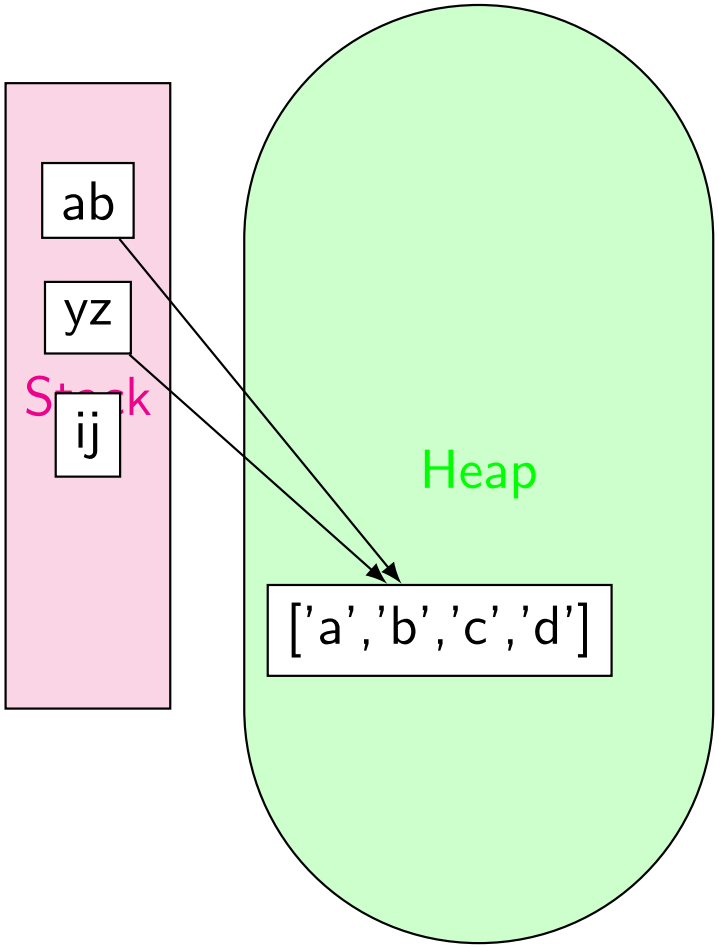



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Le garbage collector efface les objets déréférencés

IUT Nantes

Les variables sont des références : exemple de `String`

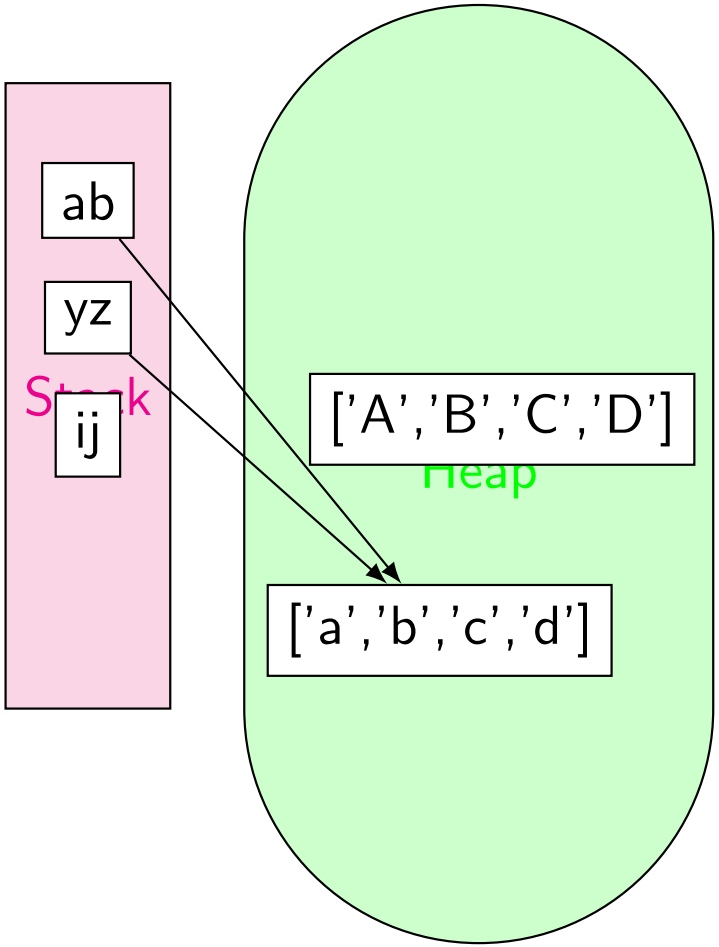



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`ij` créé dans la pile mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

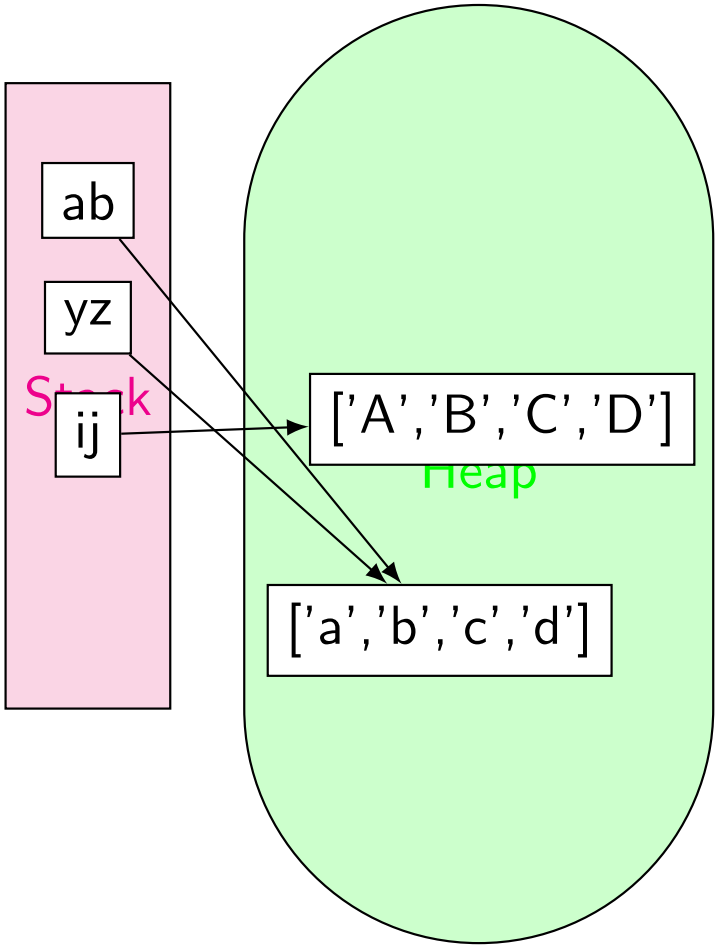



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`"ABCD"` créé dans le tas mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

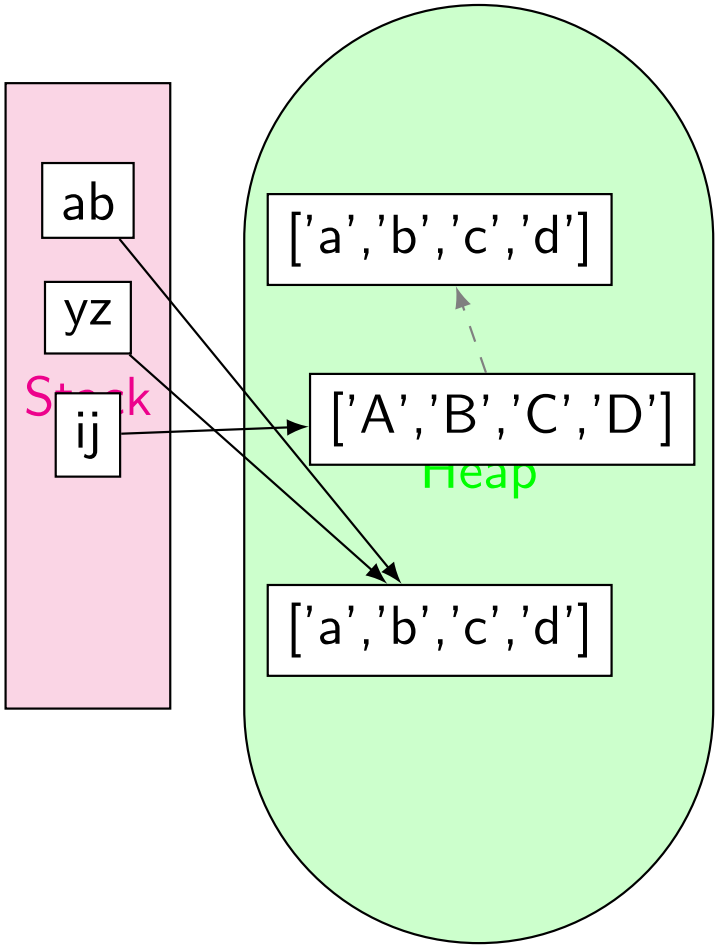



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`ij` "pointe" vers `"ABCD"`

IUT Nantes

Les variables sont des références : exemple de `String`

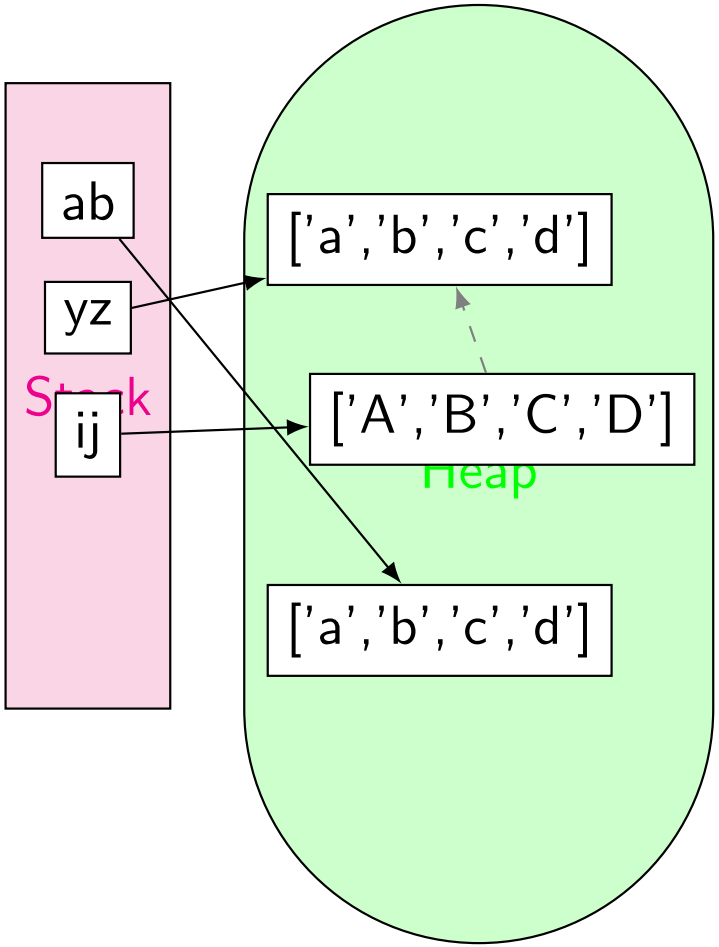



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`ij.lowercase()` créé `"abcd"` dans le tas mémoire

IUT Nantes

Les variables sont des références : exemple de `String`

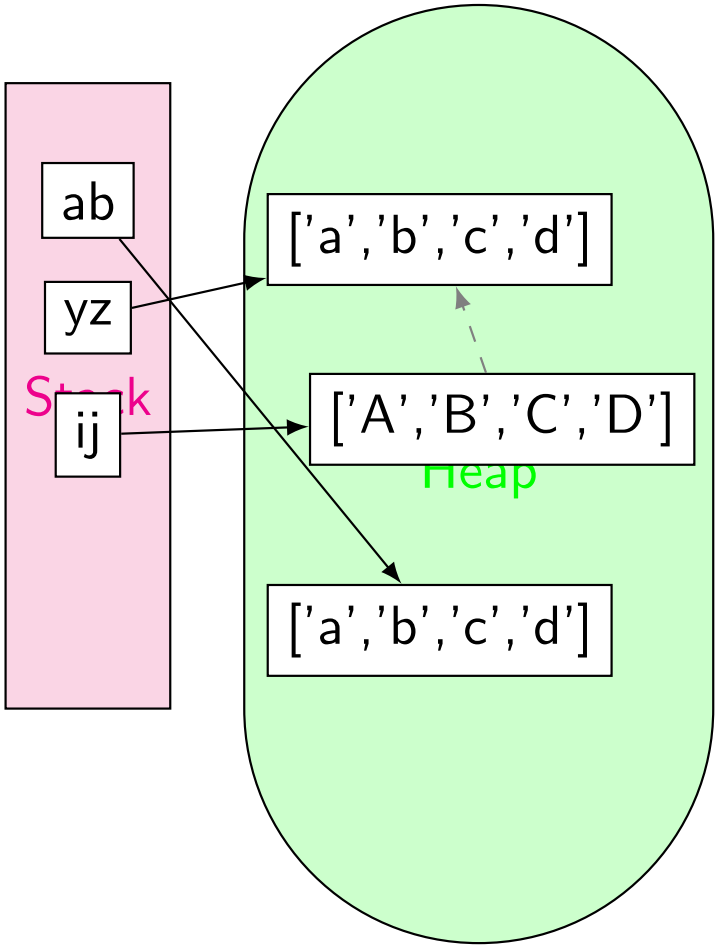



```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

`yz` "pointe" vers `"abcd"`

IUT Nantes

Les variables sont des références : exemple de `String`




```
var ab : String
var yz : String
ab = "abcd"
yz = "ABCD"
println("val: ${ab == yz}")//false
println("ref: ${ab === yz}")//false
yz = ab
println("val: ${ab == yz}")//true
println("ref: ${ab === yz}")//true
var ij = "ABCD"
yz = ij.lowercase()
println("val: ${ab == yz}") //true
println("ref: ${ab === yz}")//false
```

Les valeurs de `ab` et de `yz` sont $=$,
mais leur références sont $\neq$

IUT Nantes

Variables *nullable*

Si toute variable est une référence alors elle peut "pointer" vers rien ?
En Kotlin, **NON**
Sauf si on a précisé explicitement qu'elle pouvait.

- Ajouter `?` après le type indique que la variable est possiblement `null`
- Les paramètres et/ou le résultat d'une fonction peuvent aussi être possiblement `null`

```
var w : Int
val x : Int?
var y : Double? = 10.0
var z : String? = "totoro"
// w = null
// erreur de compilation
y = null
z = null
```

"*The Billion-Dollar Mistake*" (C.A.R. Hoare)

Forcer à indiquer les variables possiblement `null` permet d'éviter une grand nombre d'erreurs "classiques" du genre NullPointerEception, qui arrive dès lors qu'on essaie d'accéder à une variable qui ne référence rien

IUT Nantes

Variables *nullable*

Si toute variable est une référence alors elle peut "pointer" vers rien ?
En Kotlin, **NON**
Sauf si on a précisé explicitement qu'elle pouvait.

- Ajouter `?` après le type indique que la variable est possiblement `null`
- Les paramètres et/ou le résultat d'une fonction peuvent aussi être possiblement `null`

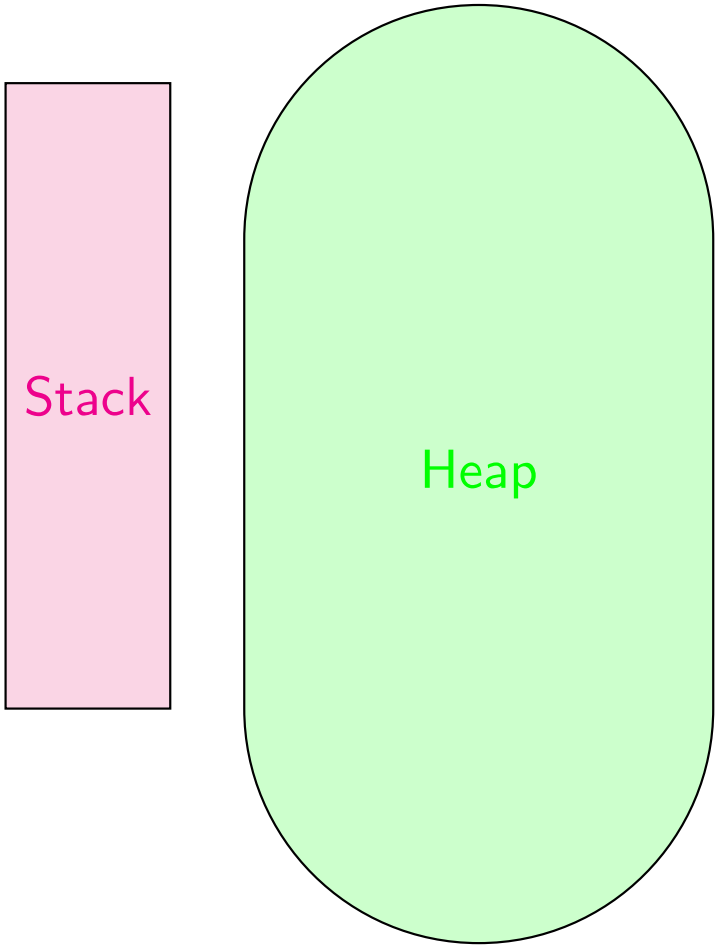
```
var w : Int
val x : Int?
var y : Double? = 10.0
var z : String? = "totoro"
// w = null
// erreur de compilation
y = null
z = null
```

"*The Billion-Dollar Mistake*" (C.A.R. Hoare)

Forcer à indiquer les variables possiblement `null` permet d'éviter une grand nombre d'erreurs "classiques" du genre NullPointerEception, qui arrive dès lors qu'on essaie d'accéder à une variable qui ne référence rien

IUT Nantes

Les variables sont des références : exemple de `String?`

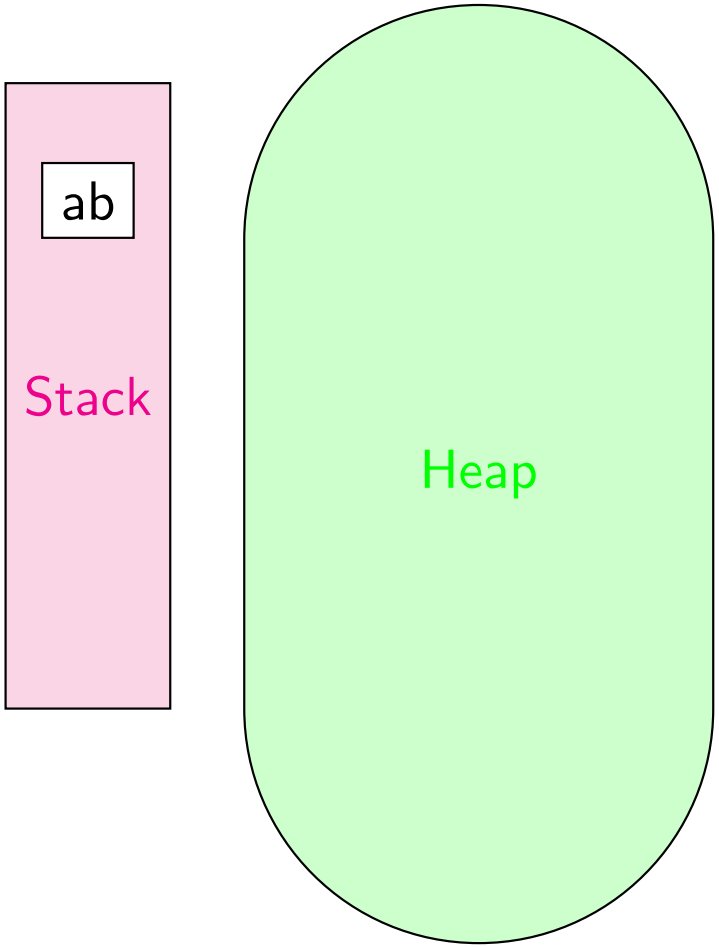



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Schématisation de la mémoire de la JVM

IUT Nantes

Les variables sont des références : exemple de `String?`

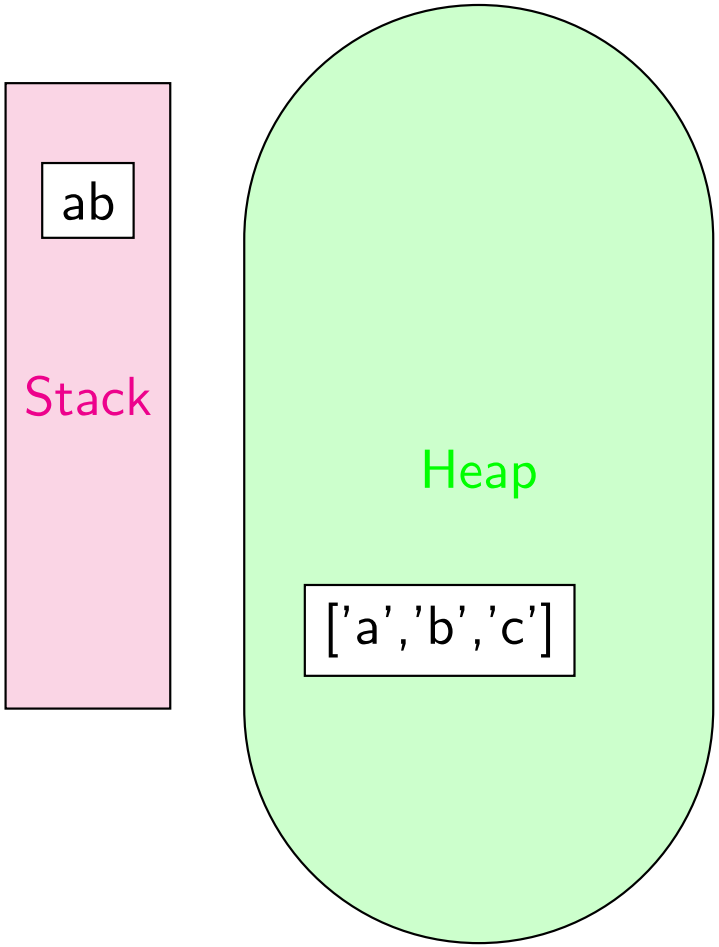



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`ab` créé dans la pile mémoire

IUT Nantes

Les variables sont des références : exemple de `String?`

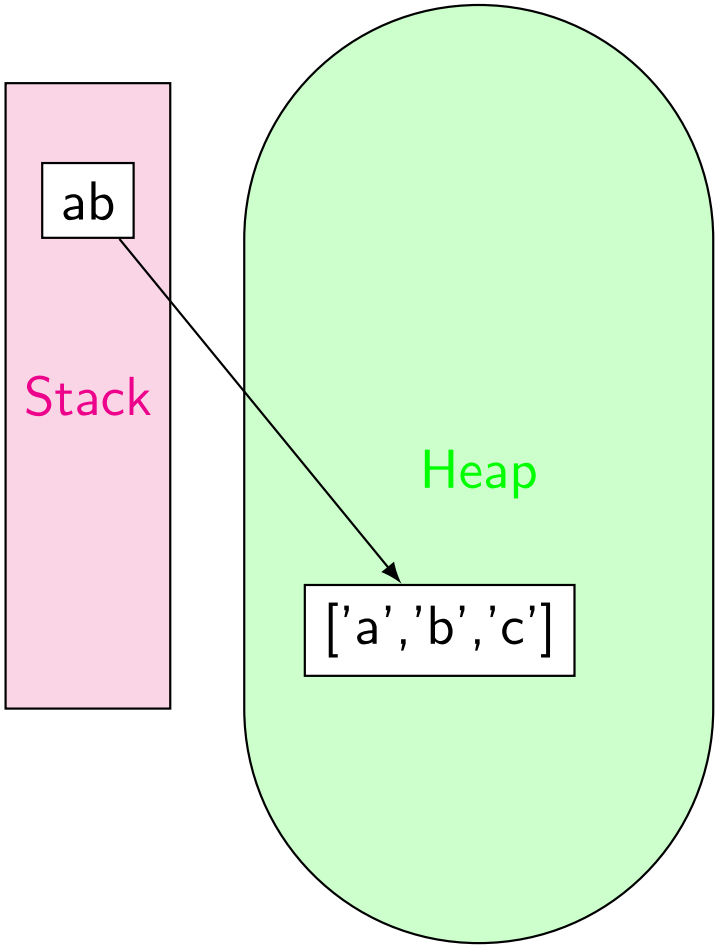



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`"abc"` créé dans le tas mémoire

IUT Nantes

Les variables sont des références : exemple de `String?`

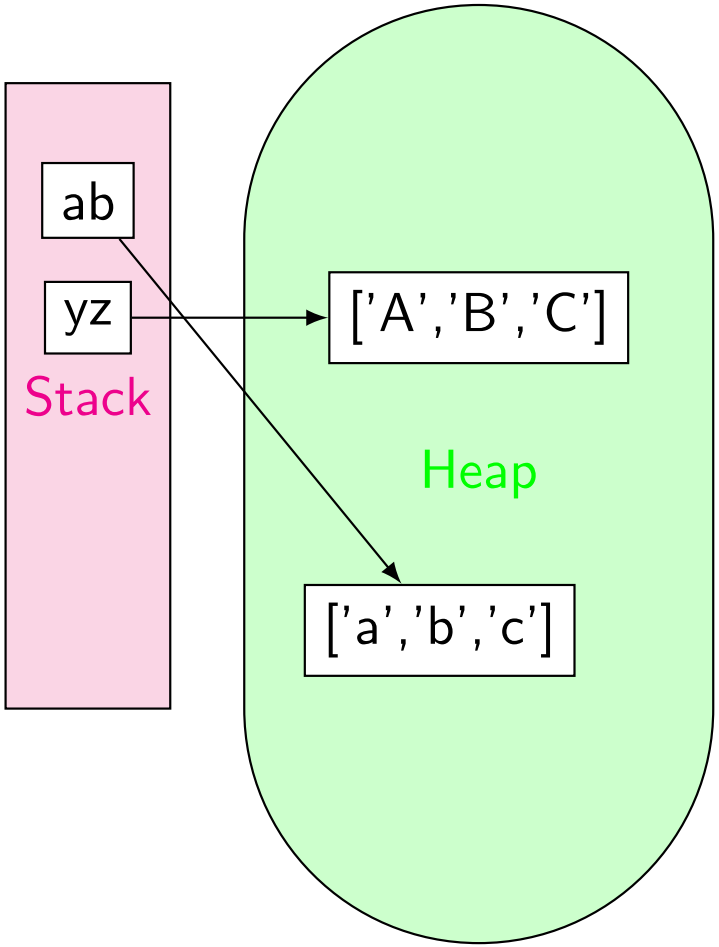



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`ab` "pointe" vers `"abc"`

IUT Nantes

Les variables sont des références : exemple de `String?`



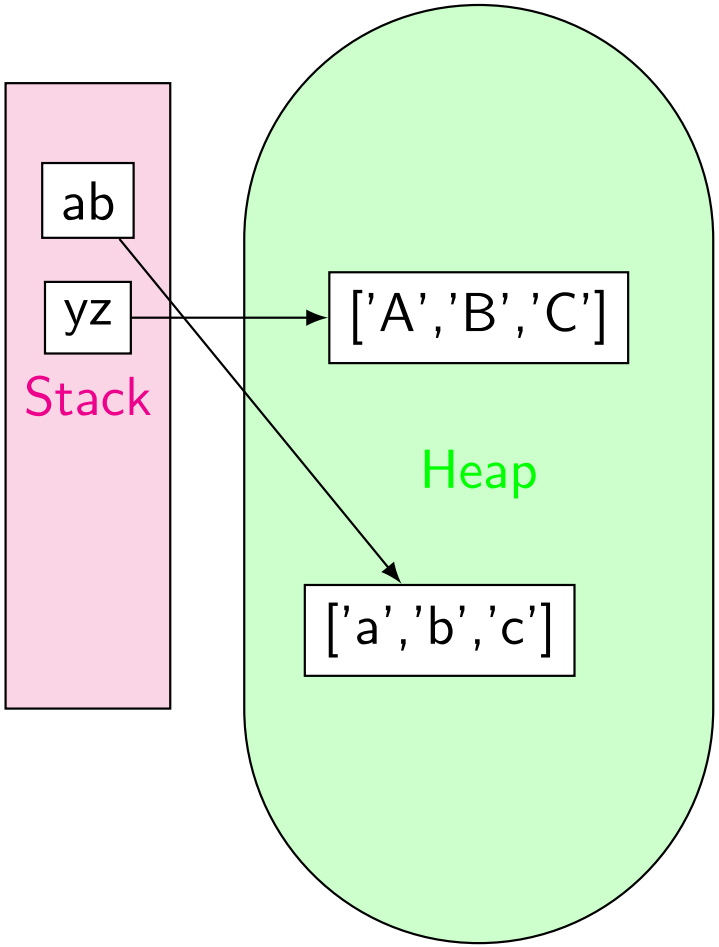

```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`yz` et `"ABC"` créés

`yz` "pointe" vers `"ABC"`

IUT Nantes

Les variables sont des références : exemple de `String?`

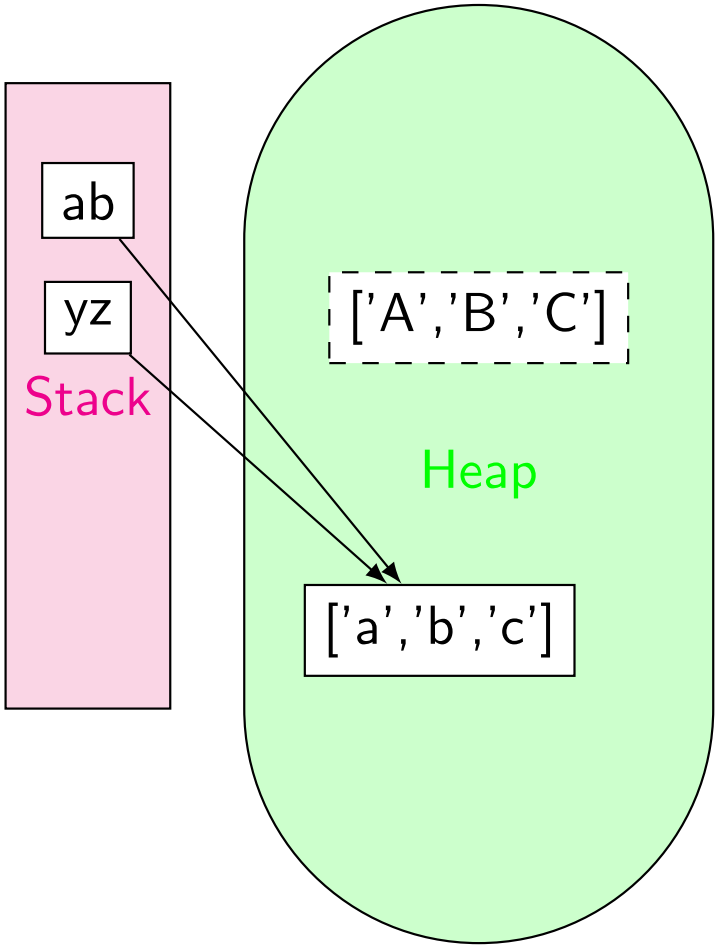



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Les valeurs de `yz` et `ab` sont $\neq$

IUT Nantes

Les variables sont des références : exemple de `String?`

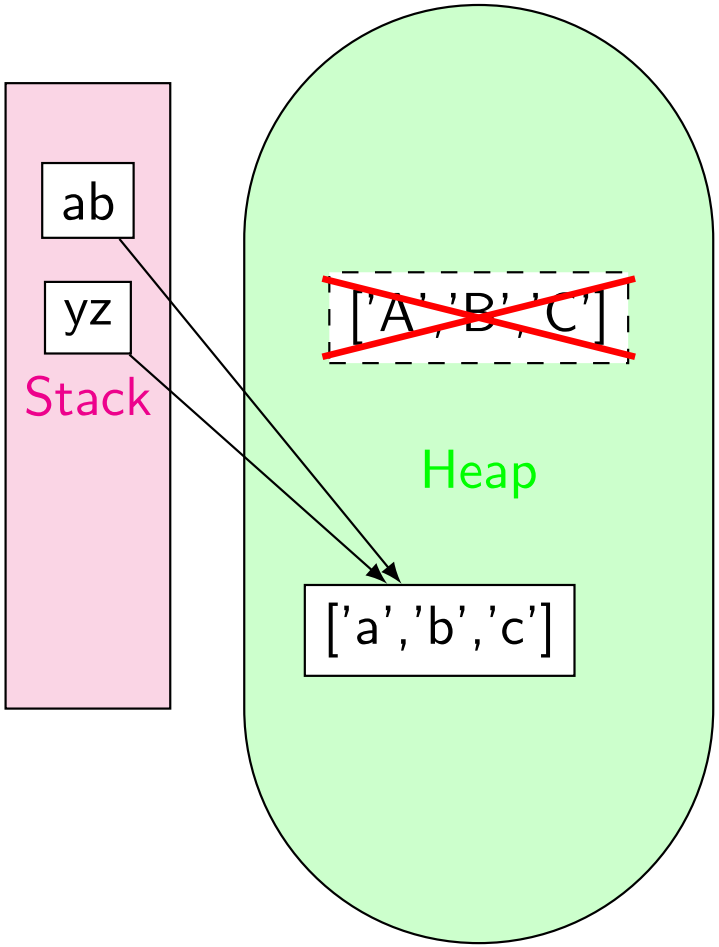



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`yz` et `ab` "pointent" vers la même valeur

IUT Nantes

Les variables sont des références : exemple de `String?`

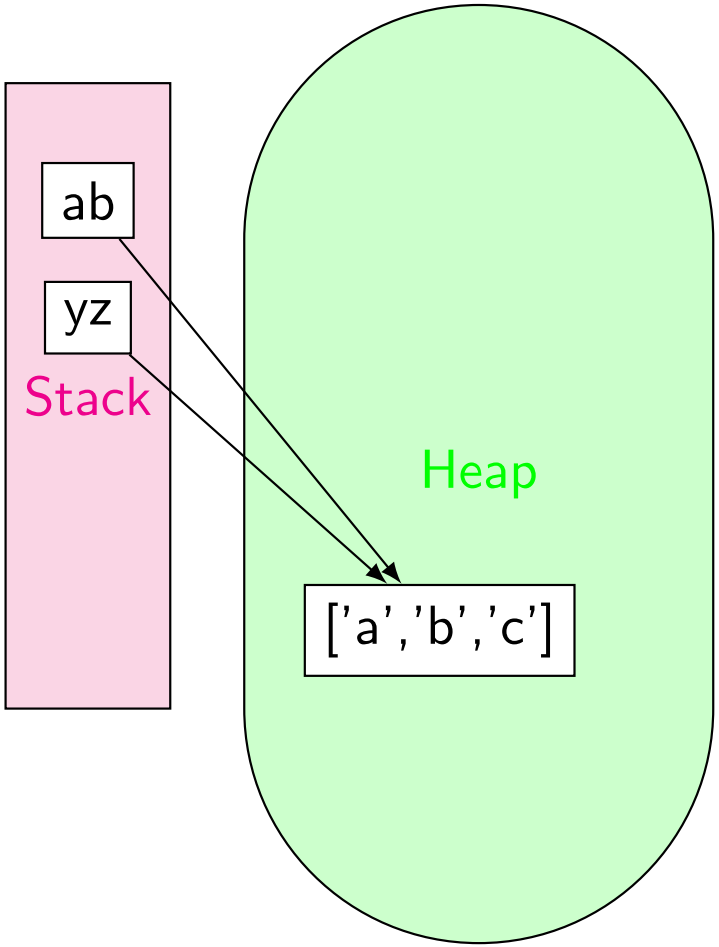



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Le garbage collector efface les objets déréférencés

IUT Nantes

Les variables sont des références : exemple de `String?`

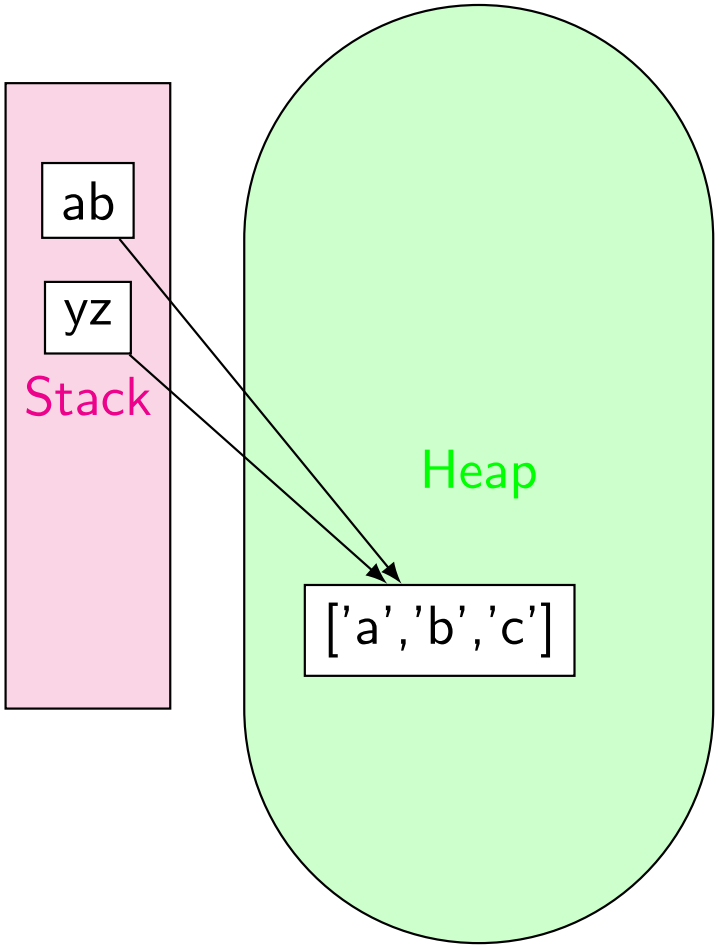



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Le garbage collector efface les objets déréférencés

IUT Nantes

Les variables sont des références : exemple de `String?`

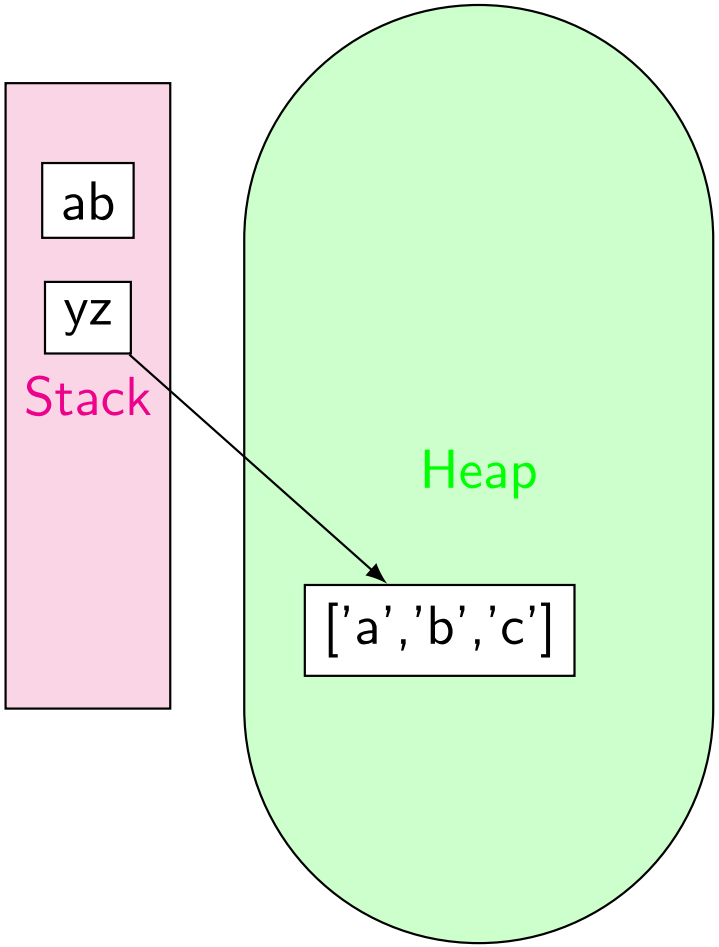



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Les valeurs de `yz` et `ab` sont =

IUT Nantes

Les variables sont des références : exemple de `String?`

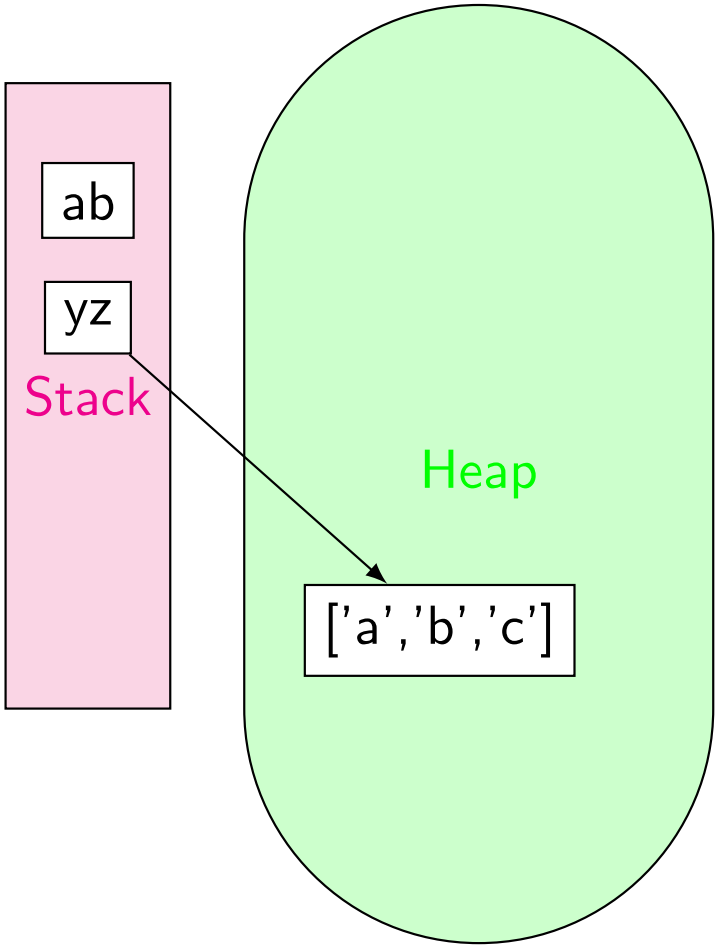



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

ab ne pointe plus vers rien

IUT Nantes

Les variables sont des références : exemple de `String?`

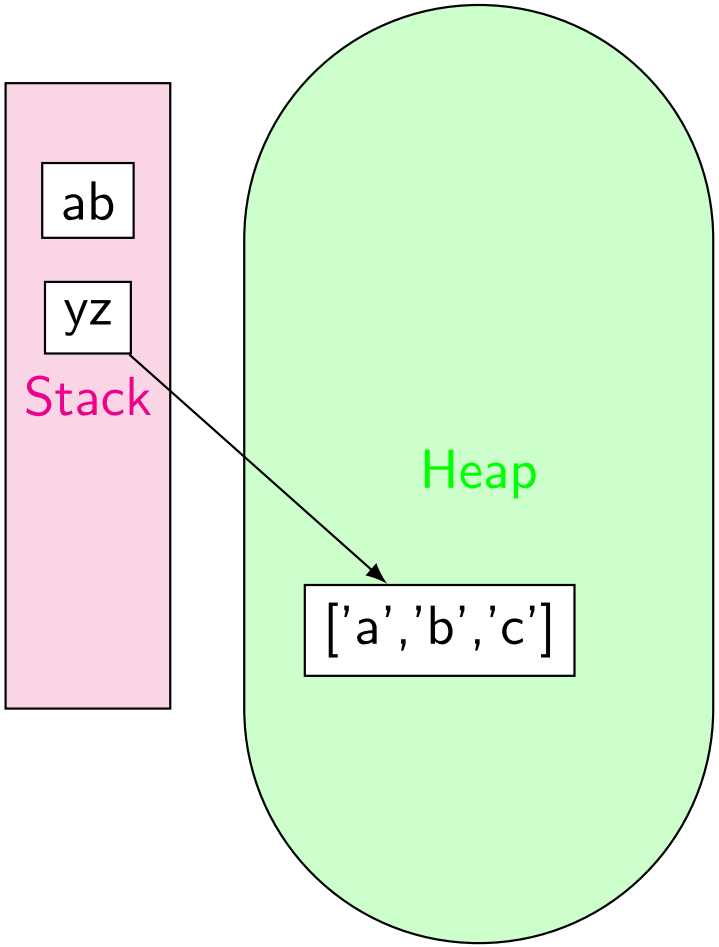



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`yz` n'est pas affecté par la mise à null de `ab`

IUT Nantes

Les variables sont des références : exemple de `String?`




```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

Cet appel provoquerait une erreur, car `ab` ne pointe vers rien

IUT Nantes

Utiliser des variables *nullable*

Kotlin vérrouille l'accès aux variables *nullable*.

1. Réaliser des appels "sûrs" via `?` :
   `z?.length` retourne `z.length` si `z` $\neq$ `null` sinon retourne `null`
2. Utiliser l'opérateur Elvis `?:`
   `z?.length ?: 0` : si la partie gauche, ici `z?.length`, = `null` alors on retourne la partie droite, ici `0`
3. Forcer l'évaluation via `!!` :
   `z!!` retourne une version non-nulle de `z` si `z` $\neq$ `null`
   mais si `z` = `null`
   NullPointerException

```
var z : String? = "totoro"
...
...
//val l = z.length
// erreur de compilation

var l = z?.length
println(l)

l = z!!.length
println(l)

l = z?.length ?: 0
println(l)
```

IUT Nantes

Sommaire

IUT Nantes

Tableaux de taille fixe

1. déclarer un tableau prérempli

```
val notes = arrayOf(12.0, 7.0, 10.5, 8.2, 17.8)
val matieres = arrayOf("Info", "Math", "Anglais", "Eco", "Comm")
```

2. déclarer un tableau vide

```
val notes0 = arrayOfNulls<Double>(4)
val matieres0 = arrayOfNulls<String>(10)
```

- dans le cas 2. il faut déclarer le type des éléments contenus `<...>` et la taille du tableau
- dans le cas 2. toutes les cases du tableau contiennent la valeur `null` [4]
- le type des tableaux est `Array<Double?>` et `Array<String?>`
- la taille du tableau est définitivement fixée

4. On y reviendra

Accéder à un tableau

Classiquement, les tableaux sont indicés de 0 à taille du tableau - 1.
Le tableau `matieres` contient 5 cases indicées de 0 à 4.

matieres =	indice	0	1	2	3	4
	valeur	"Info"	"Math"	"Anglais"	"Eco"	"Comm"

On accède aux valeurs d'un tableau via `[...]` :

```
val mat = matieres[0]
println(matieres[2])
matieres[0] = "Droit"
matieres[2] = "Russe"
```

Parcours indicé :

```
for (indice in 0 until matieres.size) {
println(matieres[indice])
}
```

Foreach :

```
for (mat in matieres) {
println(mat)
}
```

IUT Nantes

Sommaire

IUT Nantes

Héritage en programmation objet

La notion d'héritage est centrale en conception et programmation objet.
Elle permet de

- mieux appréhender le domaine métier modélisé
  - *qu'est-ce qui est commun ? qu'est-ce qui est spécifique ?*
  - généralisation vs. spécialisation
- mutualiser des parties du code pour éviter la duplication
  - covariance
- mieux architecturer le code
- faciliter l'évolution du code, la maintenance
- faciliter la réutilisation et l'adaptation du code
  - polymorphisme

IUT Nantes

Déclarer un héritage en Kotlin

```
open class Animal(nom:String, age:Int) {
protected var nom :String
protected var age : Int
private var maitre : Personne?

init {
    this.nom = nom
    this.age = age
    this.maitre = null
}

fun repondre(unNom : String) =
        (nom == unNom)
}
```

```
class Chien(nom:String, age:Int, race:String)
    : Animal(nom, age) {
private val race : String

init {
    this.race = race
}

fun aboyer() {
  println("$nom dit : ouaf ouaf !!!")
}
```

- La super-classe autorise l'héritage : `open`
  - Attributs `private` ou `protected` ~~`public`~~
- La sous-classe déclare l'héritage via `:` suivi d'un appel au constructeur de la super-classe
  - Les attributs de la super-classe ne sont pas redéclarés
  - La sous-classe accède uniquement aux attributs `protected` de la super-classe

IUT Nantes

Polymorphisme en Kotlin

Polymorphisme

Le polymorphisme consiste à redéfinir dans une sous-classe l'implémentation d'une méthode définie dans la super-classe.
En cas de covariance, c'est bien la méthode redéfinie de la sous-classe qui est appelée.

- La super-classe déclare les méthodes autorisées à être redéfinies : `open`
- La sous-classe déclare les méthodes qu'elle redéfinie : `override`
- Dans l'implémentation d'une méthode redéfinie, il est possible d'appeler la méthode de la super-classe : `super.maMethode()`

IUT Nantes

Polymorphisme en Kotlin : exemple

```
open class Animal(nom:String,age:Int){
...
open fun ageHumain() : Int {
  return 0
}

open fun courir() {
  println("$nom court !!!!")
}
```

```
class Chat(..., pedigree:String)
: Animal(nom, age) {
...
override fun ageHumain():Int{
  return age * 6
}
```

```
class Chien(..., race:String)
  : Animal(nom, age) {
...
override fun ageHumain():Int{
  return age * 7
}

override fun courir(){
  aboyer()
  super.courir()
  aboyer()
  aboyer()
}
```

- `Animal` autorise la redéfinition de `ageHumain()` et de `courir()`
- `Chien` redéfinit `ageHumain()` et `courir()`
- `Chat` ne redéfinit que `ageHumain()`

IUT Nantes

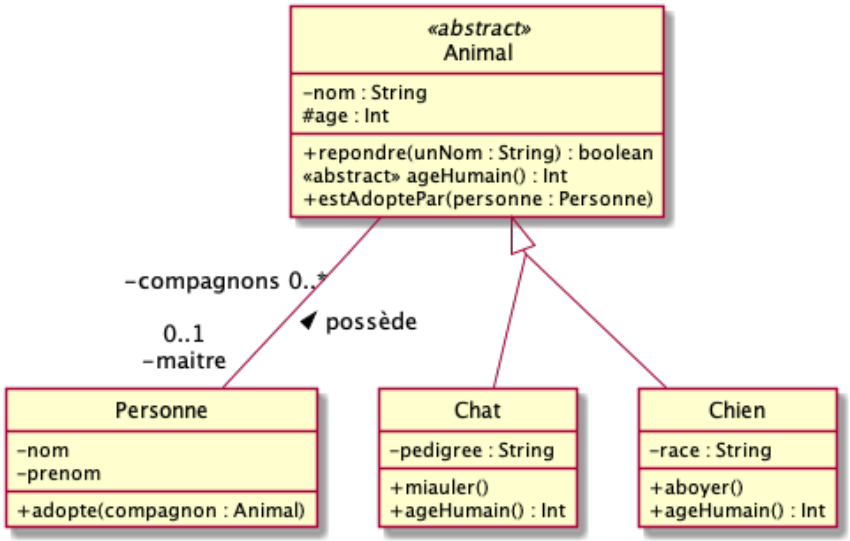
Classes abstraites en Kotlin

```
abstract class Animal(nom:String,age:Int){
protected var nom :String
protected var age : Int
private var maitre : Personne?

init {
    this.nom = nom
    this.age = age
    maitre = null
}

fun repondre(unNom : String) =
        (nom == unNom)

fun estAdopteePar(p : Personne) {
  maitre = p
}

abstract fun ageHumain() : Int

open fun courir() {
  println("$nom court !!!!")
}
```

- La Classe est déclarée abstraite par `abstract`
- La classe déclare des attributs
- La classe a un constructeur
- La classe déclare des méthodes (sans proposer d'implémentation) : `abstract`
- La classe implémente certaines méthodes
- La classe autorise la redéfinition de méthodes : `open`

IUT Nantes

Héritage : d'UML à Kotlin




```kotlin
abstract class Animal(...) {
private var nom :String
protected var age : Int
private var maitre : Personne?
...
fun repondre(unNom : String) =
        (nom == unNom)

abstract fun ageHumain() : Int

fun estAdoptePar(p : Personne) {
  maitre = p
}
```

```kotlin
class Chien(...)
        : Animal(...) {
private val race : String
...
fun aboyer() {
  println("ouaf ouaf !!!")
}
override fun ageHumain() : Int {
  return age * 7
}
```

```kotlin
class Personne (...) {
private val nom : String
private val prenom : String
private val compagnons :Array<Animal?>
private var nbCompagons : Int
...
fun adopte(compagnon : Animal) {
if (nbCompagons < compagnons.size) {
  compagnons[nbCompagons]
                = compagnon
  nbCompagons++
}
```

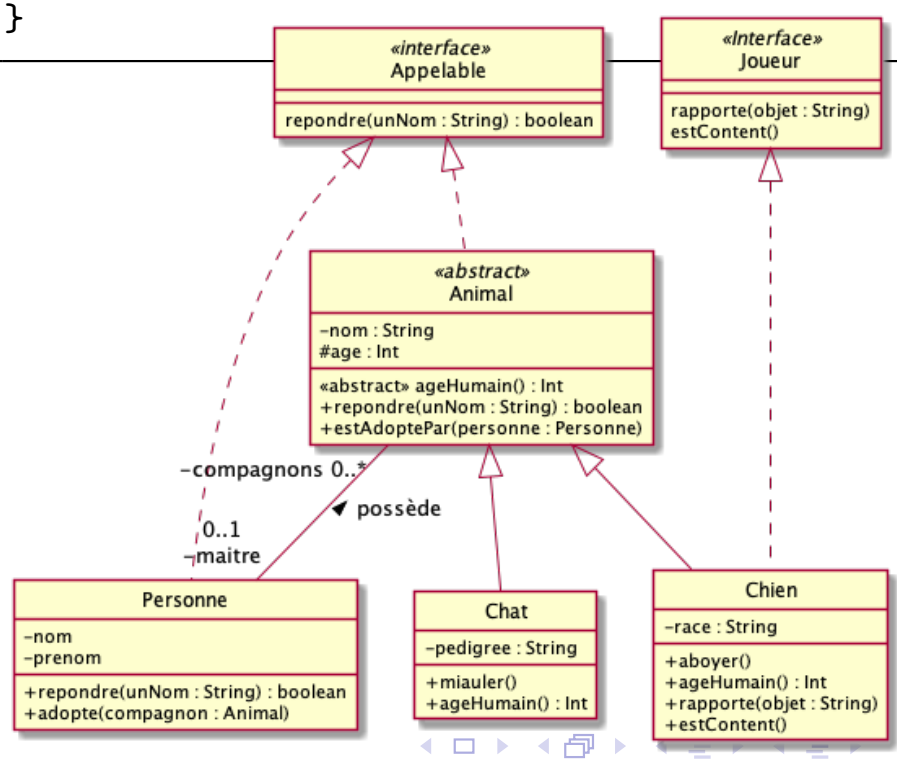
IUT Nantes

Interfaces en Kotlin

- Une interface se déclare via `interface`
- Elle déclare des méthodes
  - Elle peut proposer une implémentation par défaut
- La classe réalisant l'interface l'indique via `:`

```
interface Joueur {
fun rapporte(objet : String)

fun estContent() {
    println(" :-) ")
}
```

```
class Chien(nom:String,age:Int,race:String)
      : Animal(nom,age), Joueur {
...
override fun rapporte(objet : String) {
  courir()
  print("$nom rapporte $objet")
  if (maitre != null)
    print(" a ${maitre!!.donneNom()}")
  println("")
}
```



IUT Nantes