Développement Orienté Objets

Des objets ?

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

IUT de Nantes
Départemement informatique

Sommaire

La Programmation Orientée Objet (POO) ?

POO

Paradigme de programmation qui consiste à définir des *briques logicielles* appelées Objets et à faire interagir entre eux ces objets

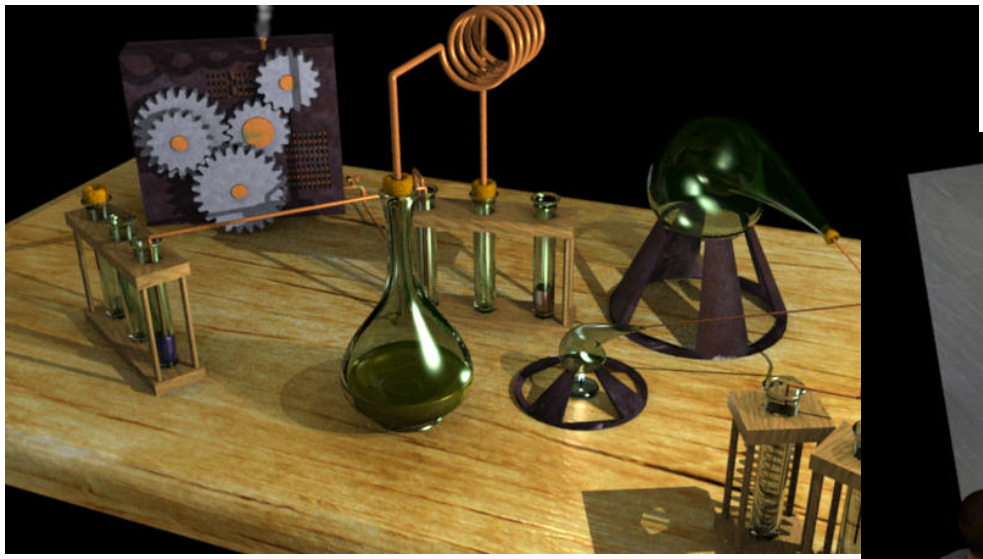
Objet

Un objet est une variable complexe représentant un concept générique. Il possède une structure interne (= attributs) et sait interagir avec d'autres objets (= méthodes)

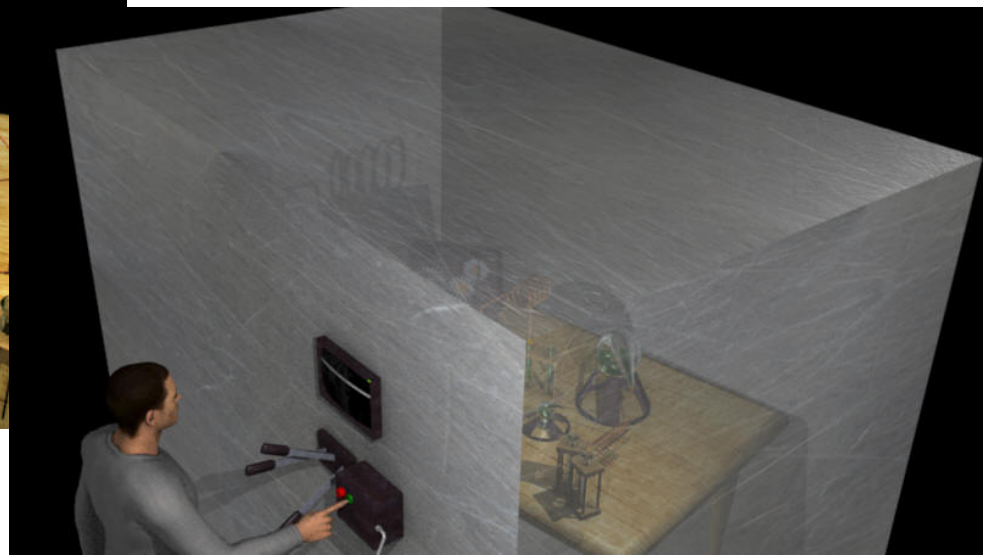
Les grands principes de la POO

- **Encapsulation :** un objet offre des méthodes permettant de le manipuler, mais cache sa structure interne
- **Abstraction :** l'utilisateur d'un objet n'a pas à connaître son fonctionnement interne pour pouvoir l'utiliser
- **Héritage :** un objet peut être défini à partir d'un autre, en ne redéfinissant que ce qui est nécessaire
- **Composition :** on construit le logiciel par assemblage d'objets simples pour obtenir des objets de plus en plus complexes
- **Polymorphisme :** une même méthode peut produire des effets différents en fonction du contexte

Encapsulation et abstraction [1]



Le code est quelque chose de très complexe...



... sauf s'il est caché dans une "boite" offrant uniquement quelques leviers ! ! !

1. Exemple repris de [Openclassrooms]

Les avantages de la POO

- (écrire moins de code)
- organiser et hiérarchiser du code complexe(modularité, composition)
- faciliter le code collaboratif
- faciliter la maintenance et l'évolution
- réutiliser du code / créer des bibliothèques
- ...

Définir des objets : la classe

= sorte de "moule" [2] 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
- ...

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

Définir des objets : la classe

= sorte de "moule" [2] 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,
- ...

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

Manipuler des objets

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

Manipuler des objets

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

POO en pratique

La POO s'inscrit dans une démarche de développement plus vaste

1. **Conception orientée Objet**
   - une méthode / langage fait consencus : UML, en particulier les diagrammes de classe

2. **Programmation par Objet**
   De très nombreux langages à objets existent :
   - C++, Java, ObjectiveC, C#, Swift, **Kotlin**,
   - Eiffel, Ada, Python, Javascript, PHP...

Sommaire

La classe `String`

- Sans le savoir, vous avez déjà manipulé des objets en Kotlin :
  La classe `String` par exemple
- Représentation interne "complexe" = un tableau de `Char` + ... :
- On n'a pas directement accès à la structure interne
- `var chaine = "ToToRo"` ≡

0	1	2	3	4	5
'T'	'o'	'R'	'o'	'R'	'o'

- `chaine` est un objet, c'est une instance de `String`
- La documentation complète :
  `https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/`

La classe `String` (2)

`var chaine = "ToToRo"` ≡

0	1	2	3	4	5
'T'	'o'	'R'	'o'	'R'	'o'

- propriétés
  - `length` – longueur de la chaîne
  - `lastIndex` – indice du dernier caractère
  - ...

```
val longueur = chaine.length
var indice = chaine.lastIndex

var verif : Boolean
verif = chaine.isEmpty()
verif = chaine.isNotEmpty()
verif = chaine.isNotBlank()
```

- méthodes
  - `isEmpty():Boolean` – la chaîne est vide
  - `isNotEmpty():Boolean` – la chaîne n'est pas vide

Les appels aux propriétés et aux méthodes d'un objet se font avec un `.` comme dans `chaine.length` ou `chaine.isEmpty()`

La classe `String` (3)

`var chaine = "ToToRo"` ≡

0	1	2	3	4	5
'T'	'o'	'R'	'o'	'R'	'o'

- Accès indicé
  - `get(i:Int):Int` ≡
  - notation `[i]` (comme pour un tableau)
- Concaténation
  - `plus(other:String) : String` ≡
  - opérateur `+`
- Appartenance
  - `contains(other:String, ignoreCase=false) : Boolean` ≡
  - opérateurs `in` ou `!in`

```
var carac = chaine.get(0)
carac = chaine.get(0)
carac = chaine[0]
carac = chaine[3]

var chaine2 : String
chaine2 = "Tonari no ".plus(chaine)
chaine2 = "Tonari no " + chaine

verif = chaine.contains("o")
verif = "o" in chaine
verif = "O" in chaine
verif = chaine.contains("O",
                ignoreCase=true)
```

La classe `String` (4)

`var chaine = "ToToRo"` ≡

0	1	2	3	4	5
'T'	'o'	'R'	'o'	'R'	'o'

- Egalité
  - `equals(other:String, ignoreCase=false) : Boolean`

    ≡ opérateurs `==` et `!=`
- Comparaison
  - `compareTo(other:String) : Int` ≡ opérateurs `<`, `<=`, `>` ou `>=`

```
verif = chaine.equals("ToToRo")
verif = chaine == "ToToRo"
verif = chaine == "totoro"
verif = chaine != "totoro"
verif = chaine.equals("totoro", ignoreCase=true)
var comp = chaine.compareTo("TOTORO")
comp = chaine.compareTo("totoro")
verif = chaine > "totoro"
verif = chaine <= "totoro"
```

La classe `String` (5)

`var chaine = "ToToRo"` ≡

0	1	2	3	4	5
'T'	'o'	'R'	'o'	'R'	'o'

- en majuscule `uppercase() : String`
- en minuscule `lowercase() : String`
- renversé `reversed() : String`
- carac. aléatoire `random() : Char`
- sous-chaines
  `substring(start:Int) : String`
  ou
  `substring(start:Int, end:Int) : String`
- Position
  `indexOf(c:Char, ignoreCase=false) : Int`

```
chaine2 = chaine.uppercase()
chaine2 = chaine.lowercase()
chaine2 = chaine.reversed()
carac = chaine.random()
carac = chaine.random()
chaine2 = chaine.substring(2)
chaine2 = chaine.substring(2, 4)
indice = chaine.indexOf('o')
indice = chaine.indexOf('O')
indice = chaine.indexOf('O',
                        ignoreCase=true)
```

Et les autres types de base ?

Tous les types de base `Int`, `Double`, `Boolean`, `Char`, etc. sont des classes.
Ils offrent certaines méthodes et propriétés.

Le type `Int`

- `toDouble() : Double`
- `toString() : String`
- `plus(other : Int) : Int` ≡ `+`, `times(other : Int) : Int` ≡ `*`,
  ...
- https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int/

Sommaire

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

Les variables sont des références

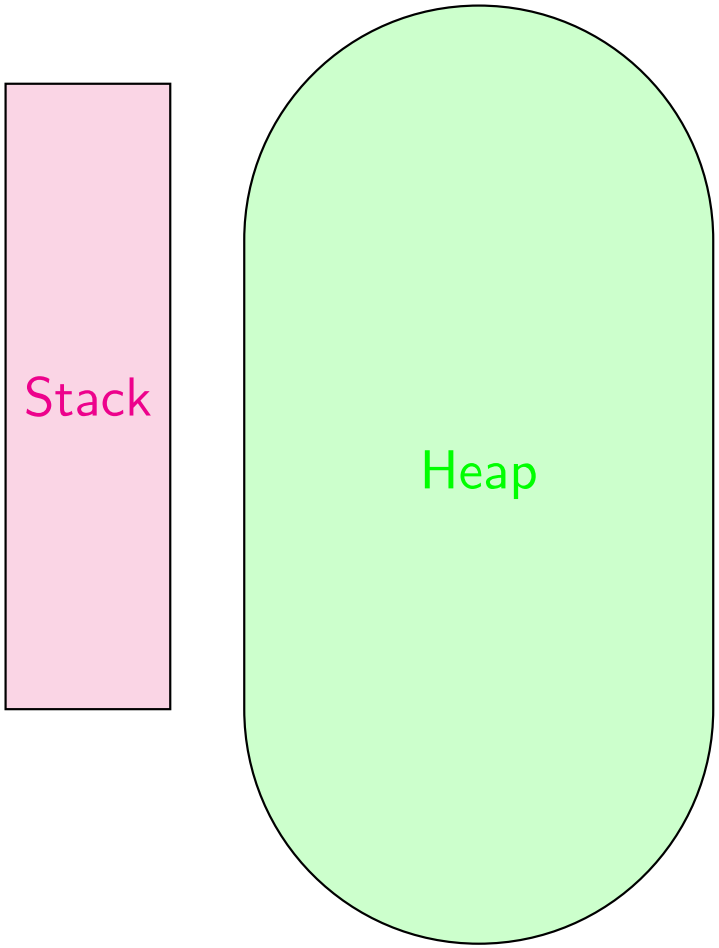
- En Kotlin, toutes les variables sont des références (dans la pile mémoire) qui "pointent" vers leur valeur (dans le tas mémoire)
- Une référence en Kotlin correspond à un pointeur en C/C++, avec une gestion simplifiée de l'allocation mémoire :
  - On ne s'occupe pas de réserver de l'espace mémoire
  - 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

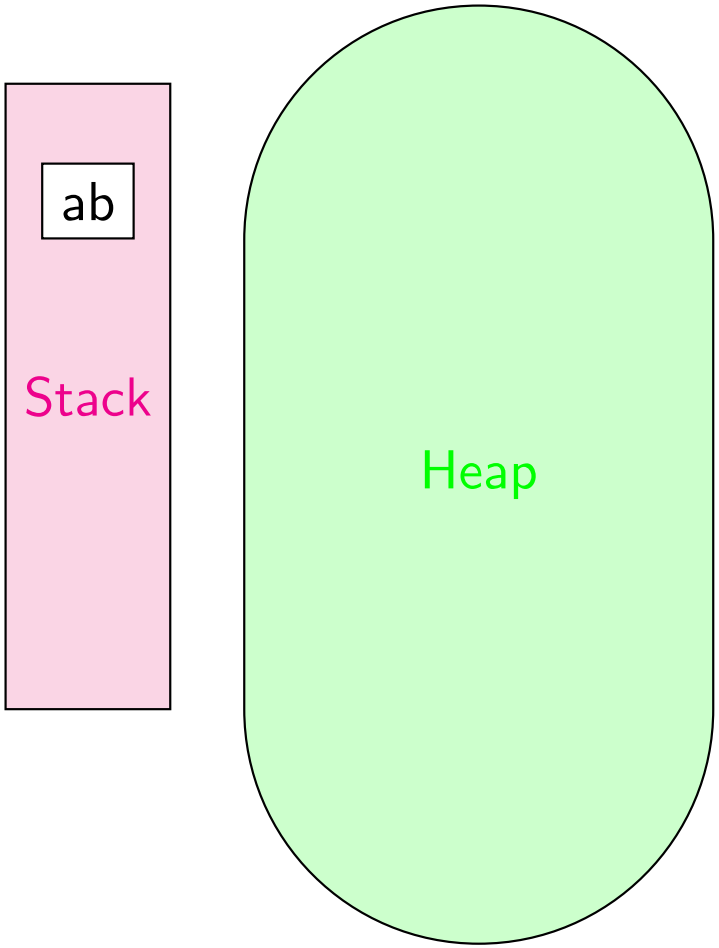
Les variables sont des références : exemple de `String`




```
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

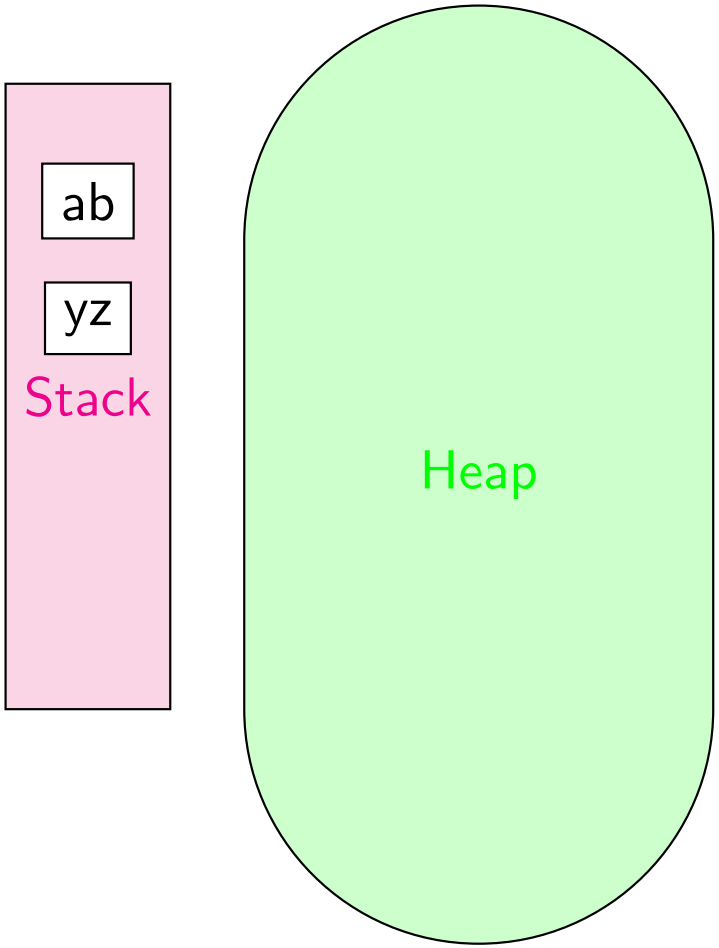
Les variables sont des références : exemple de `String`




```
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

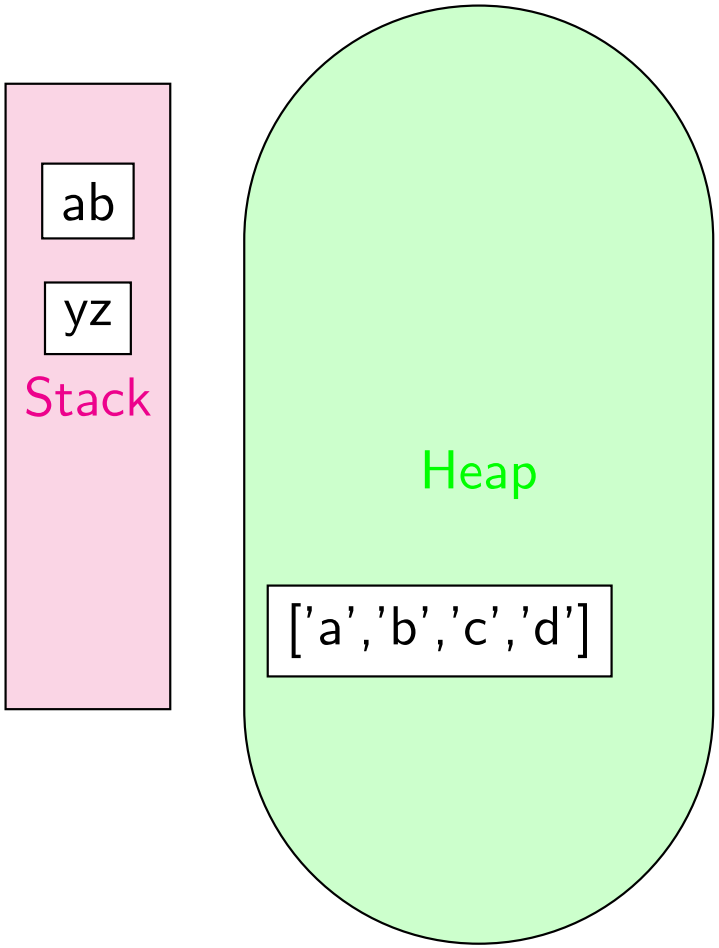
Les variables sont des références : exemple de `String`




```
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

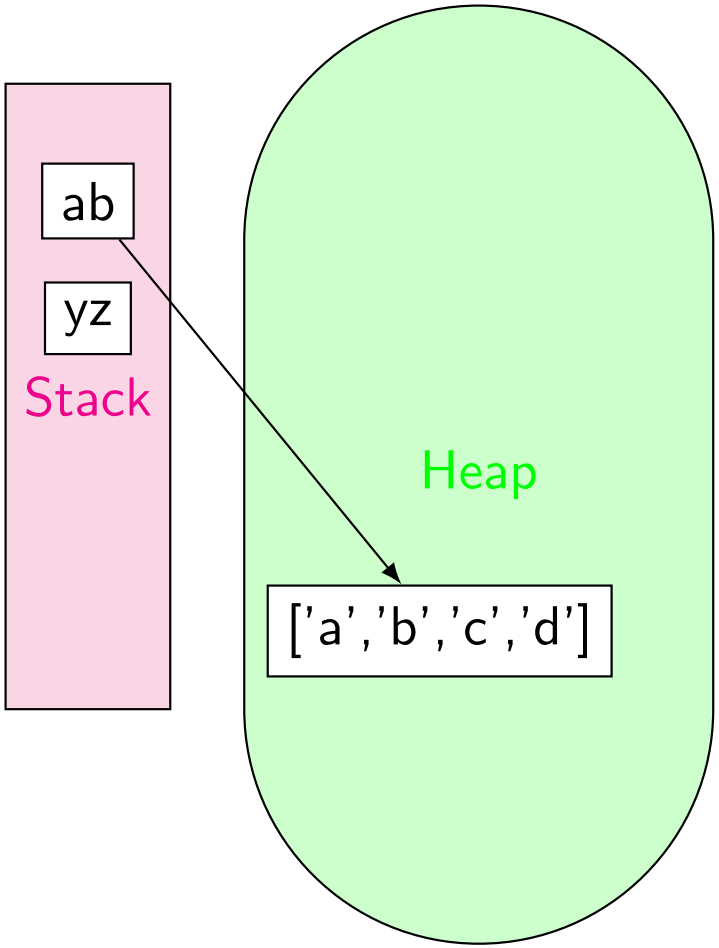
Les variables sont des références : exemple de `String`




```
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

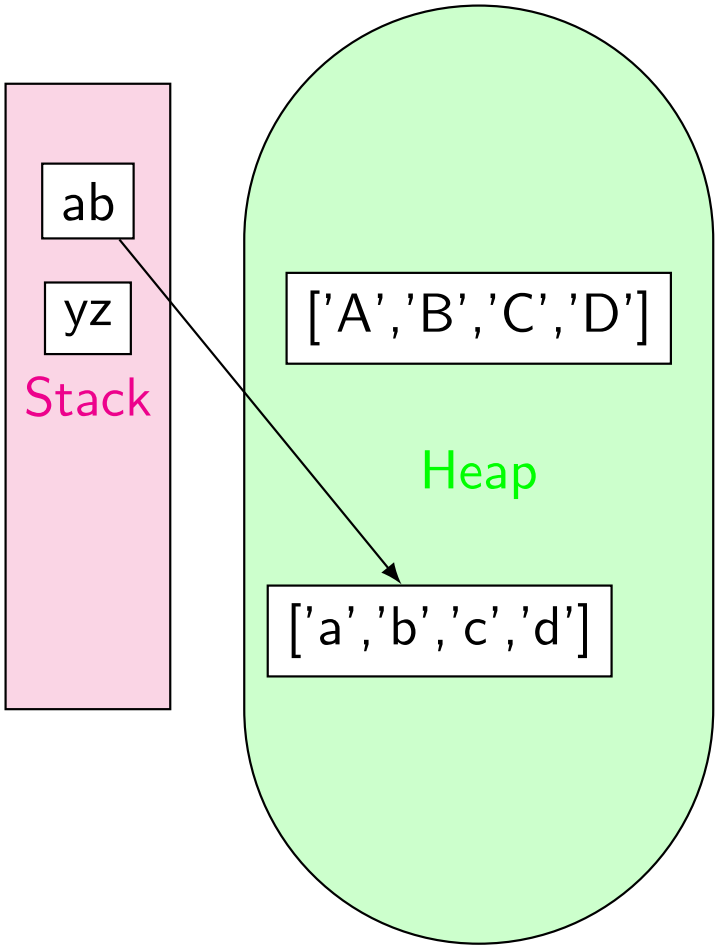
Les variables sont des références : exemple de `String`




```
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"`

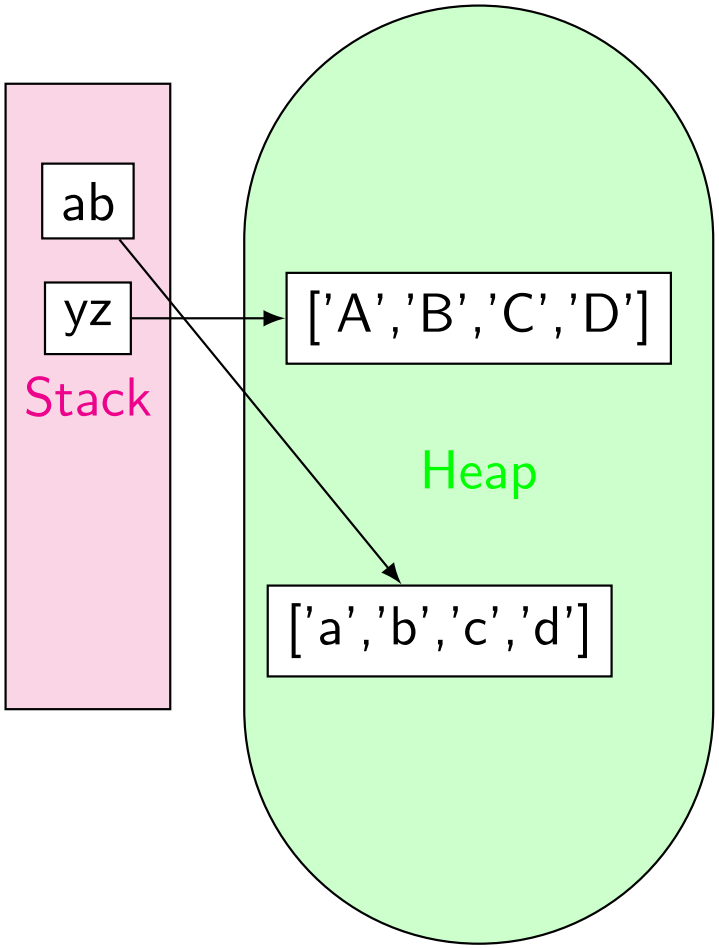
Les variables sont des références : exemple de `String`




```
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

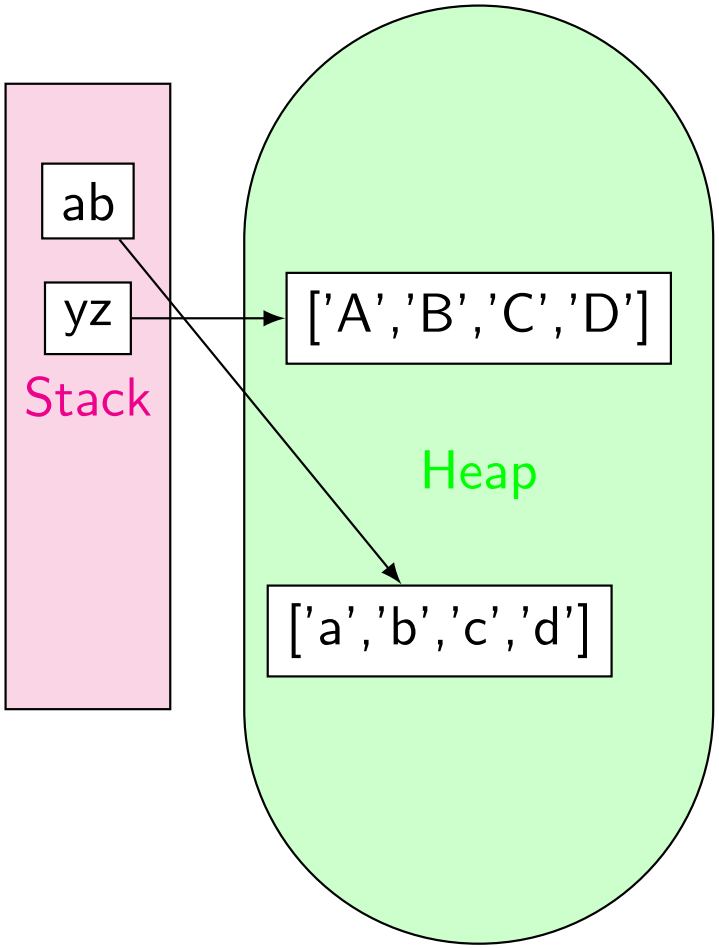
Les variables sont des références : exemple de `String`




```
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"`

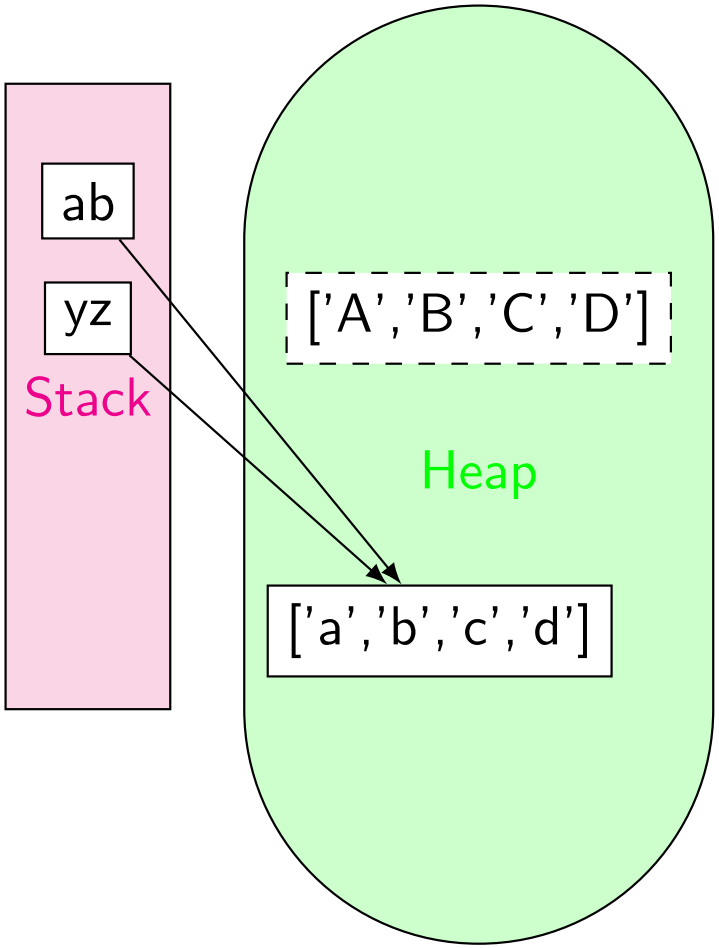
Les variables sont des références : exemple de `String`




```
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

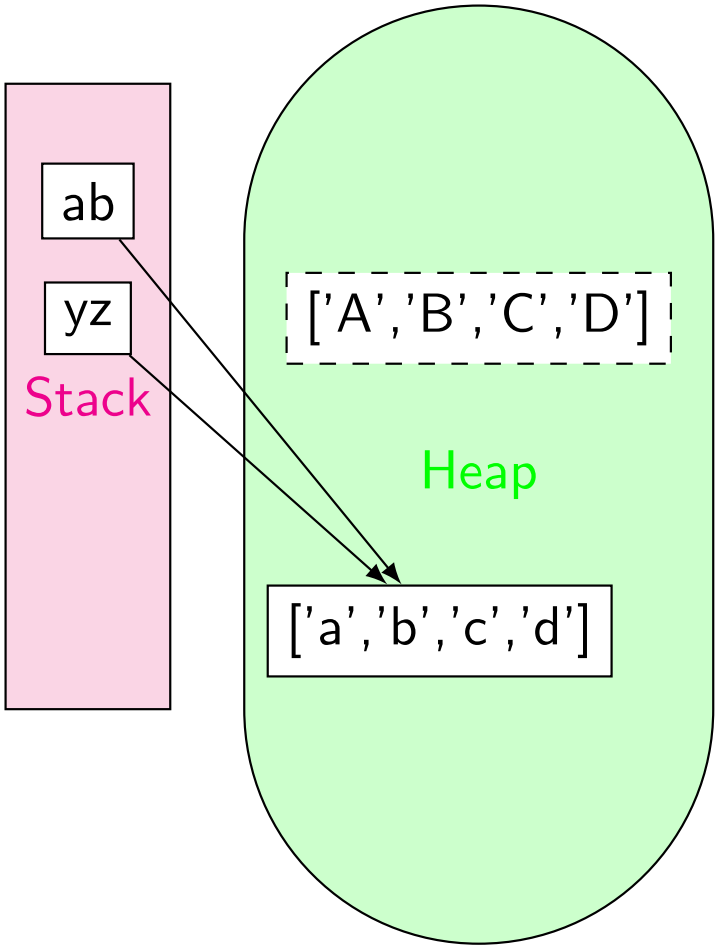
Les variables sont des références : exemple de `String`




```
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

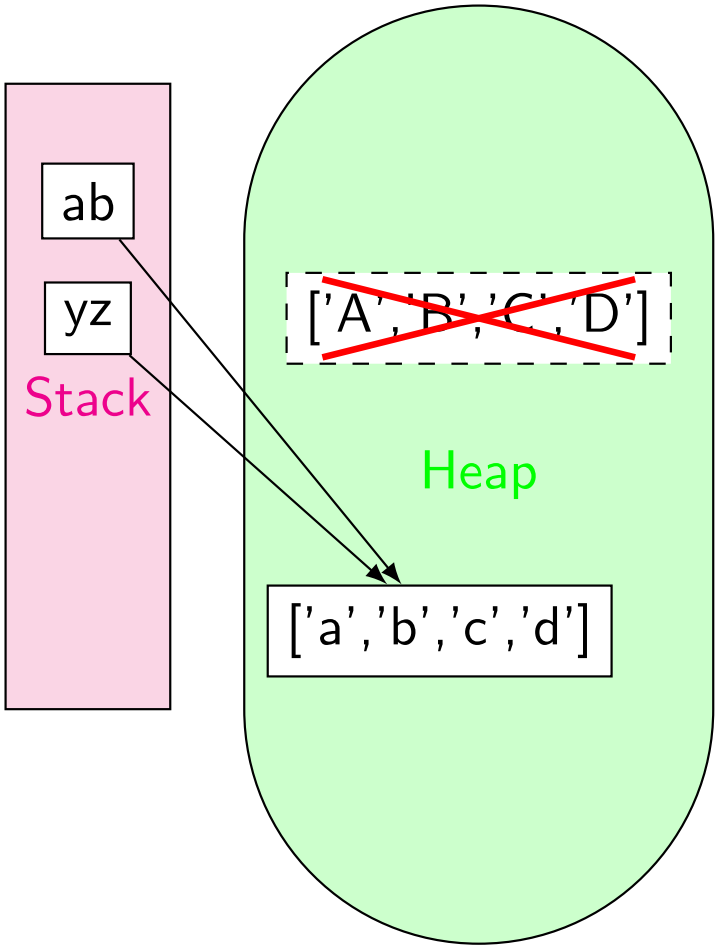
Les variables sont des références : exemple de `String`




```
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 $=$

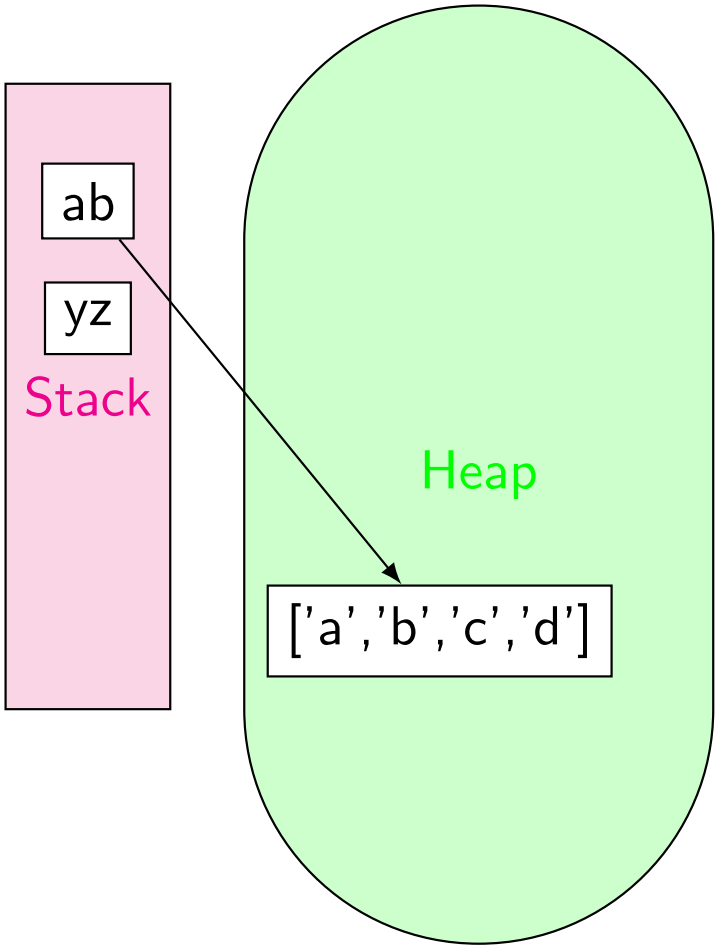
Les variables sont des références : exemple de `String`




```
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

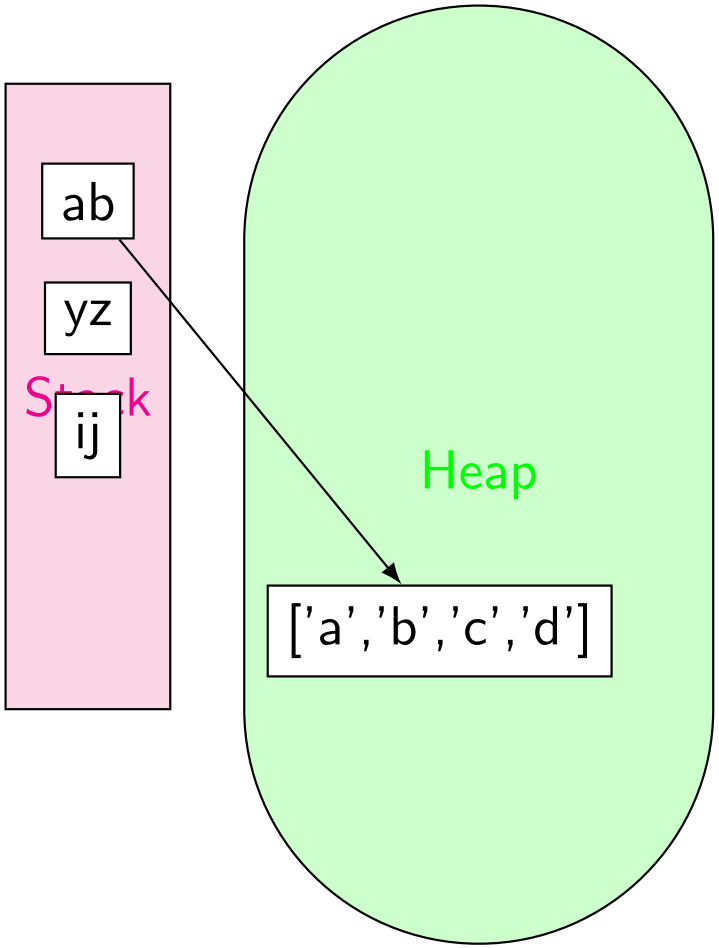
Les variables sont des références : exemple de `String`




```
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

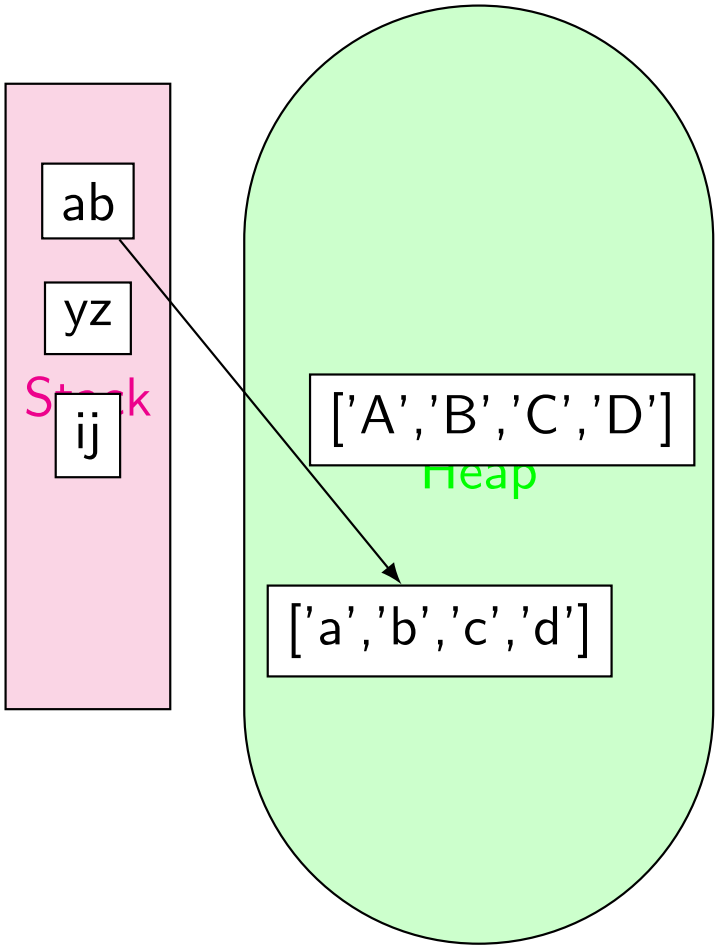
Les variables sont des références : exemple de `String`




```
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

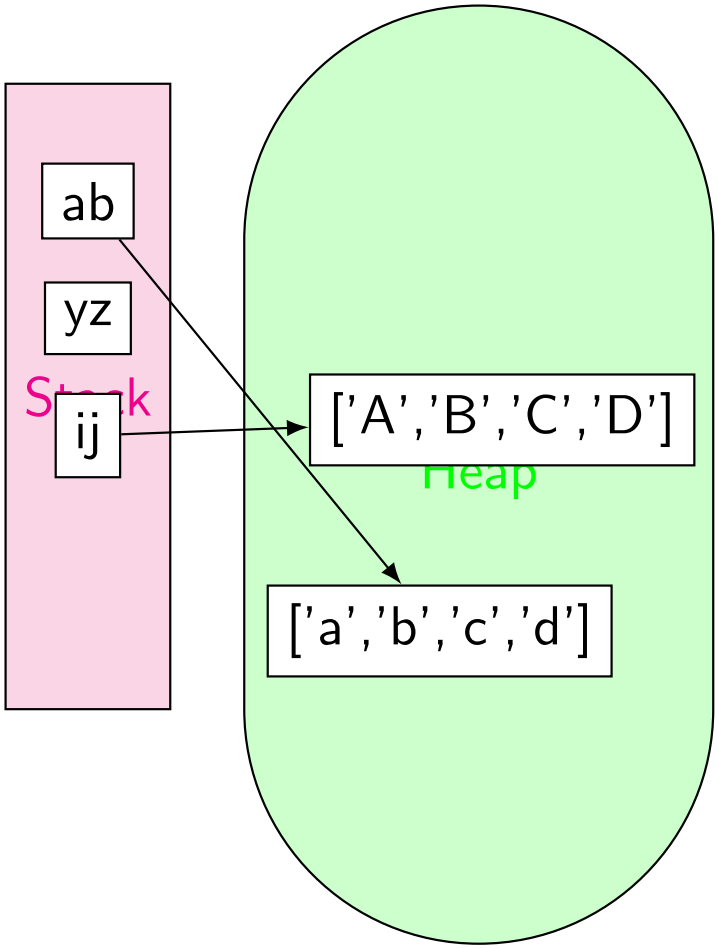
Les variables sont des références : exemple de `String`




```
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

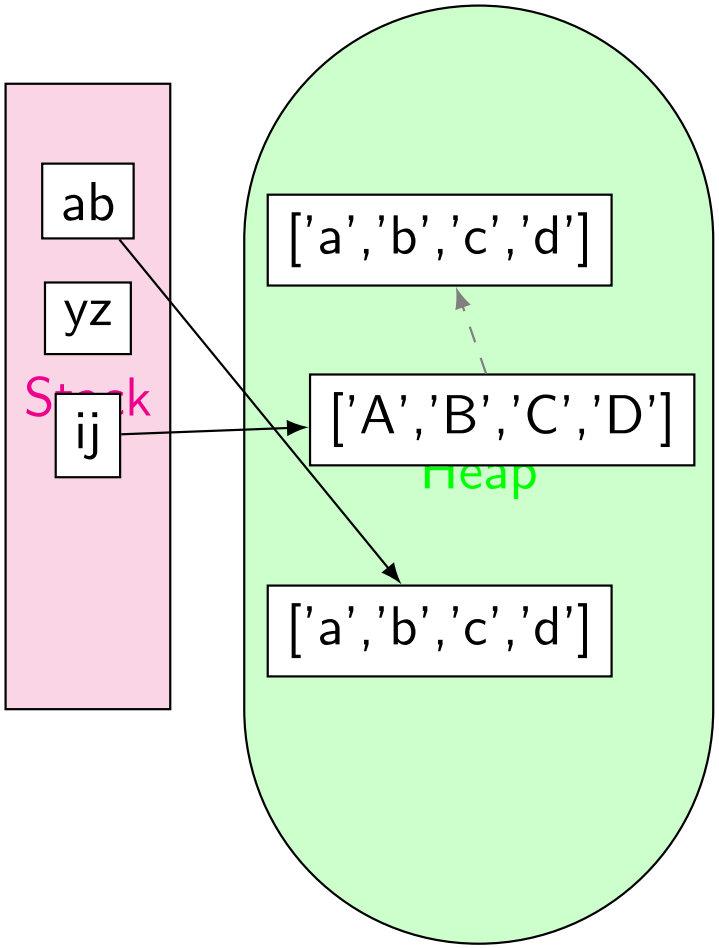
Les variables sont des références : exemple de `String`




```
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"`

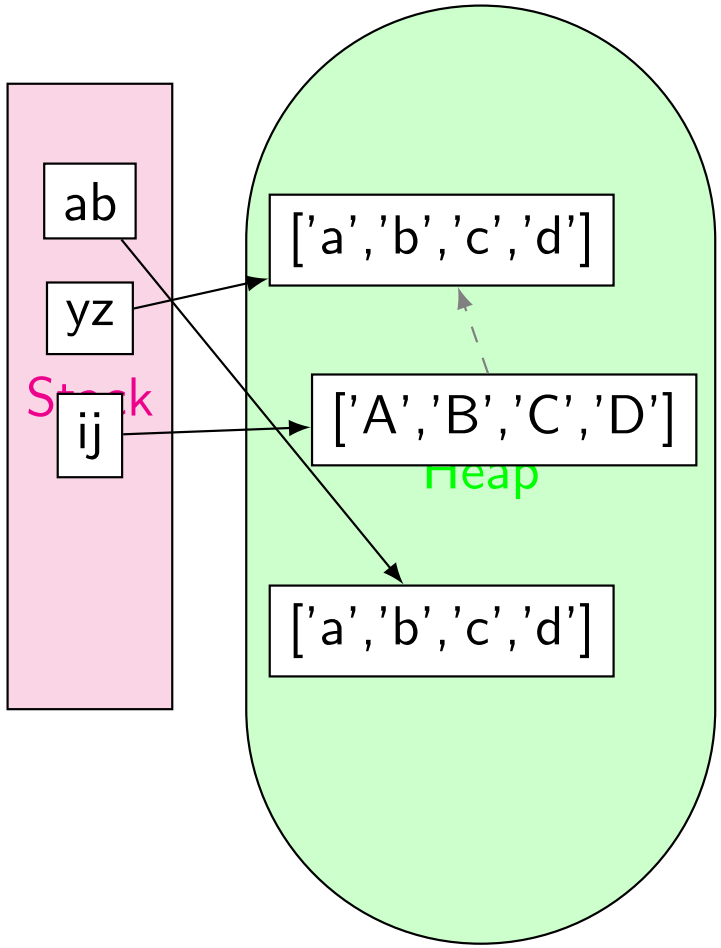
Les variables sont des références : exemple de `String`




```
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

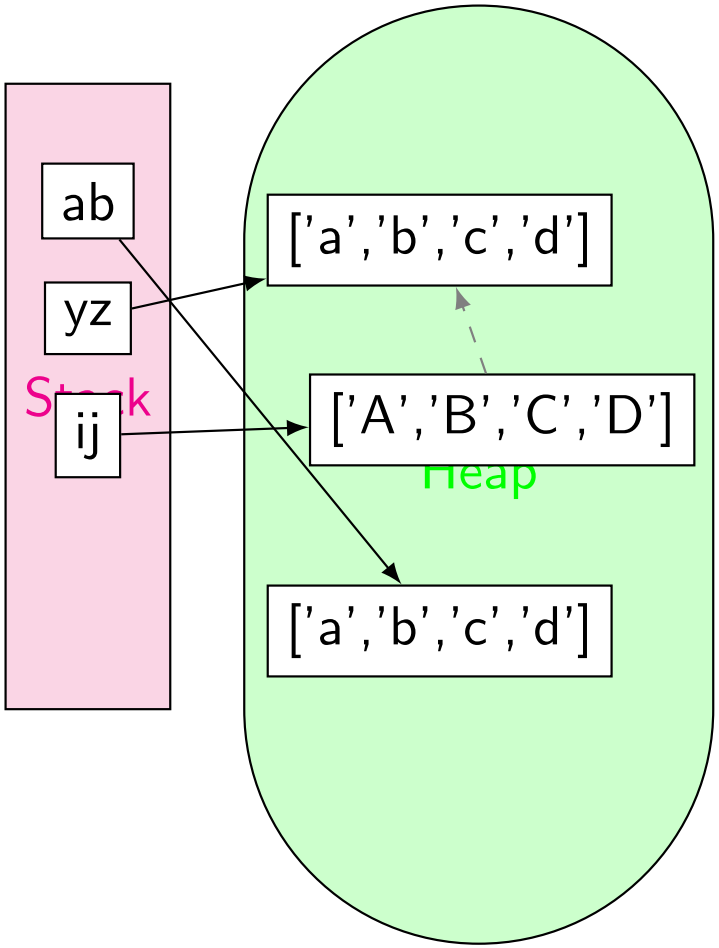
Les variables sont des références : exemple de `String`




```
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"`

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 $\neq$

Variables *nullable*

Si une variable est une référence alors elle peut "pointer" vers rien ? **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" en Java : NullPointerEception, qui arrive dès lors qu'on essaie d'accéder à une variable qui ne référence rien

Variables *nullable*

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

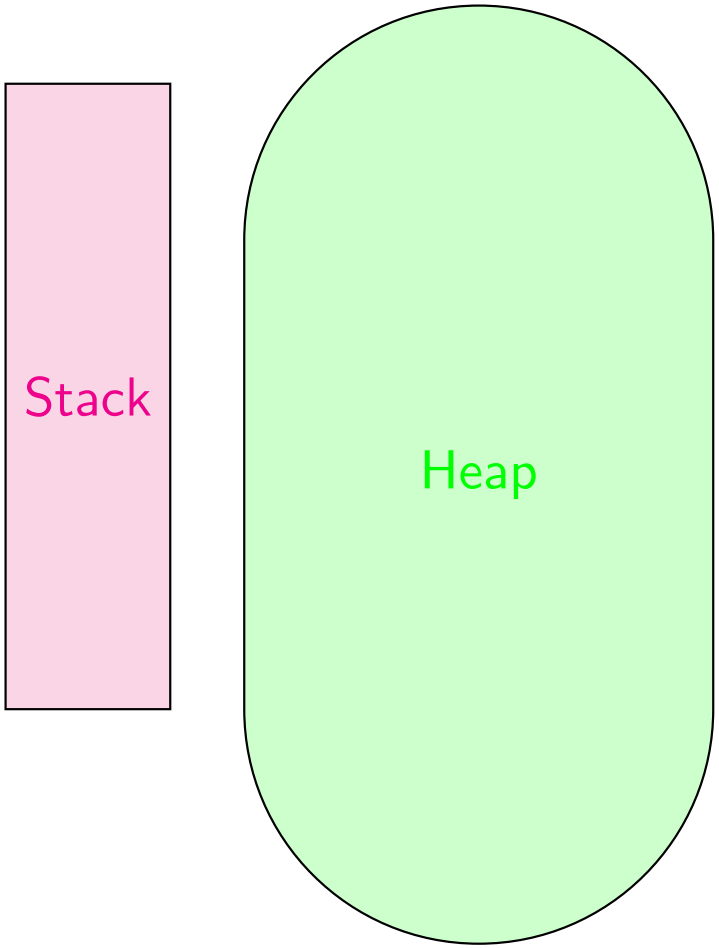
- Ajouter `?` après le type indique que la variable est possiblement `null`
- Les paramètres et/ou le résultat d'une fonction peuvent aussi être possiblement `null`

```
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" en Java : NullPointerEception, qui arrive dès lors qu'on essaie d'accéder à une variable qui ne référence rien

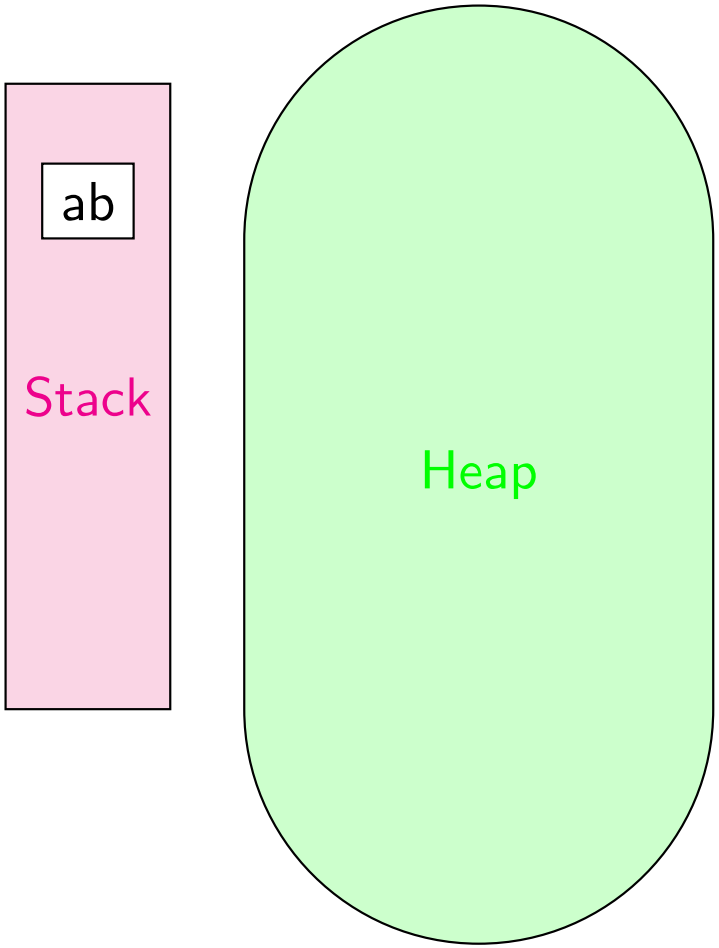
Les variables sont des références : exemple de `String?`




```
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

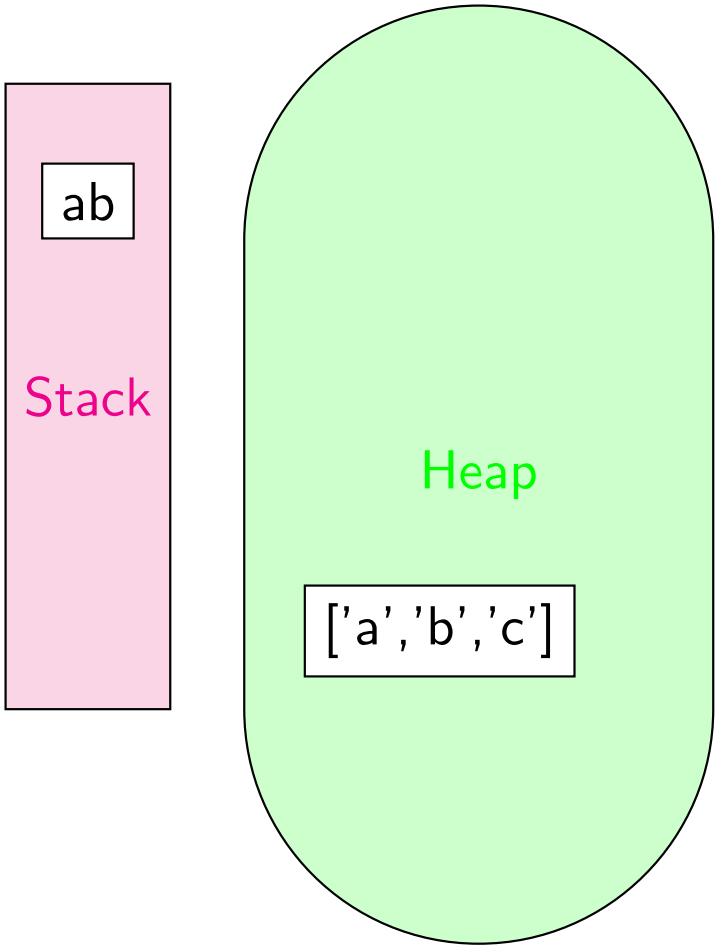
Les variables sont des références : exemple de `String?`




```
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

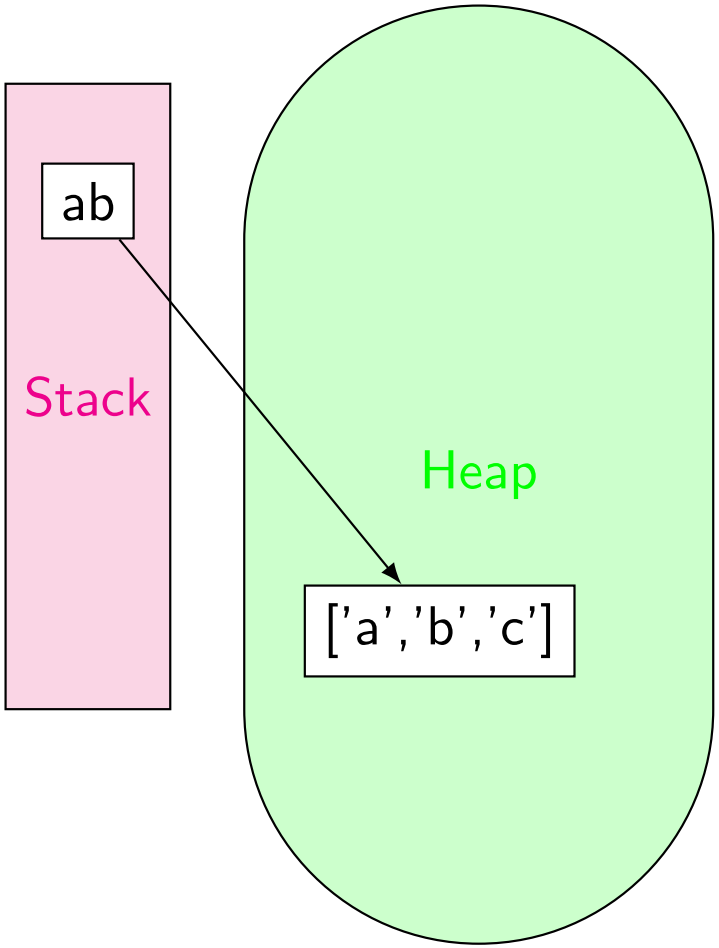
Les variables sont des références : exemple de `String?`




```
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

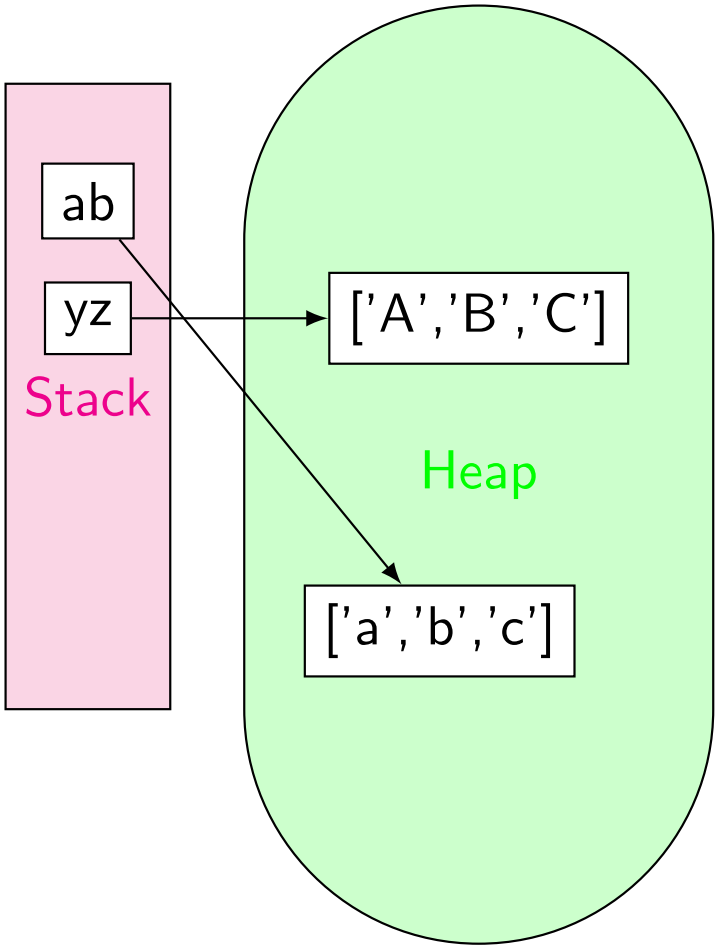
Les variables sont des références : exemple de `String?`




```
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"`

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

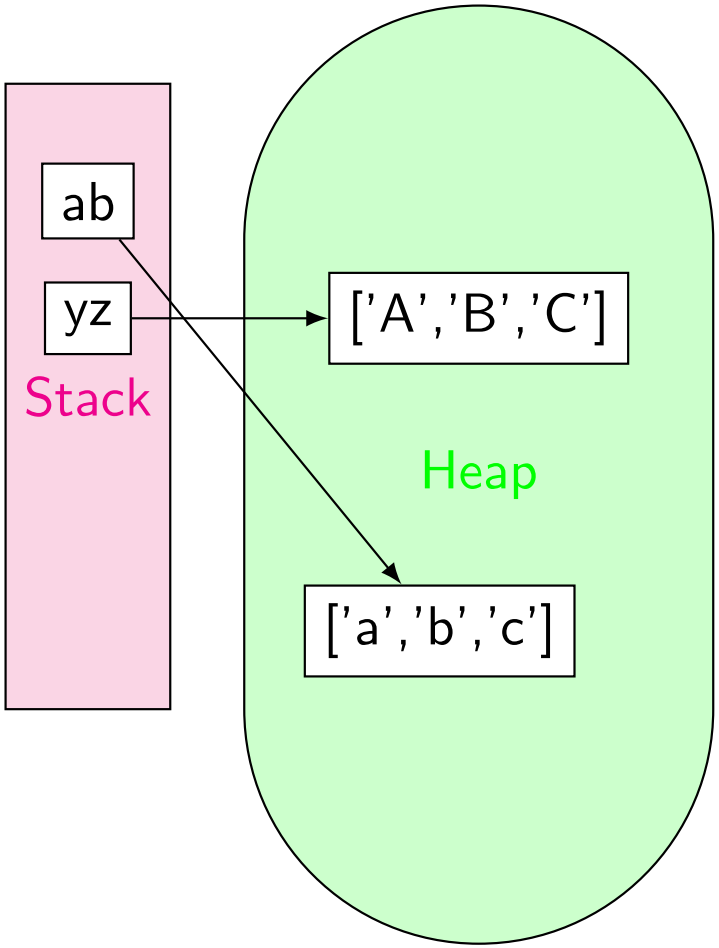



```
var ab : String? = "abc"
var yz : String? = "ABC"
println(ab) //"abc"
println(yz) //"ABC"
yz = ab
println(ab) //"abc"
println(yz) //"abc"
ab = null
println(ab) //null
println(yz) //"abc"
ab.uppercase() //erreur
```

`yz` et `"ABC"` créés

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

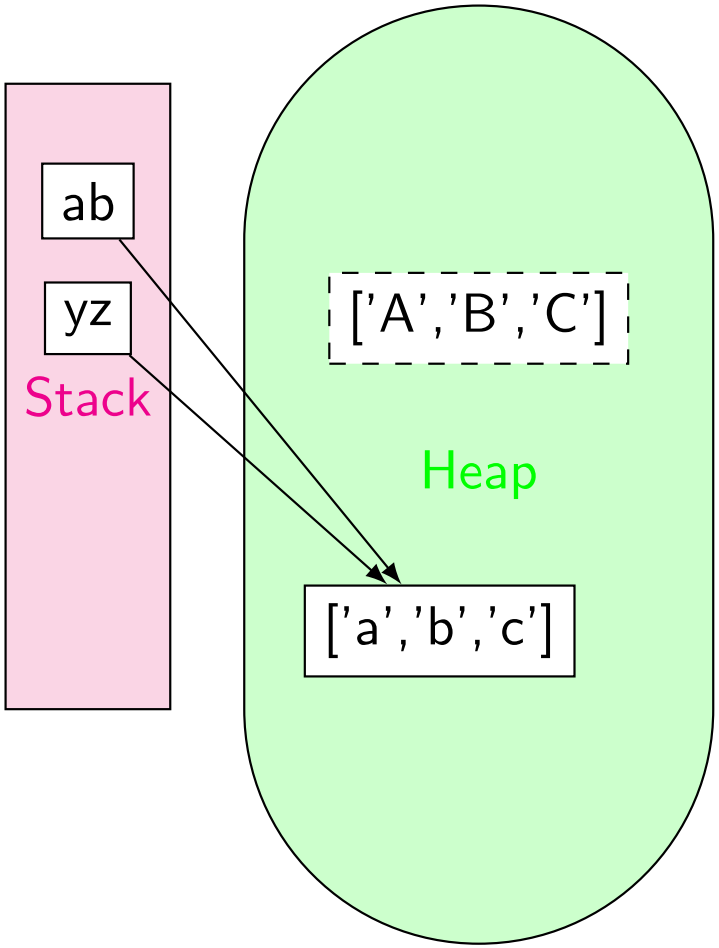
Les variables sont des références : exemple de `String?`




```
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 $\neq$

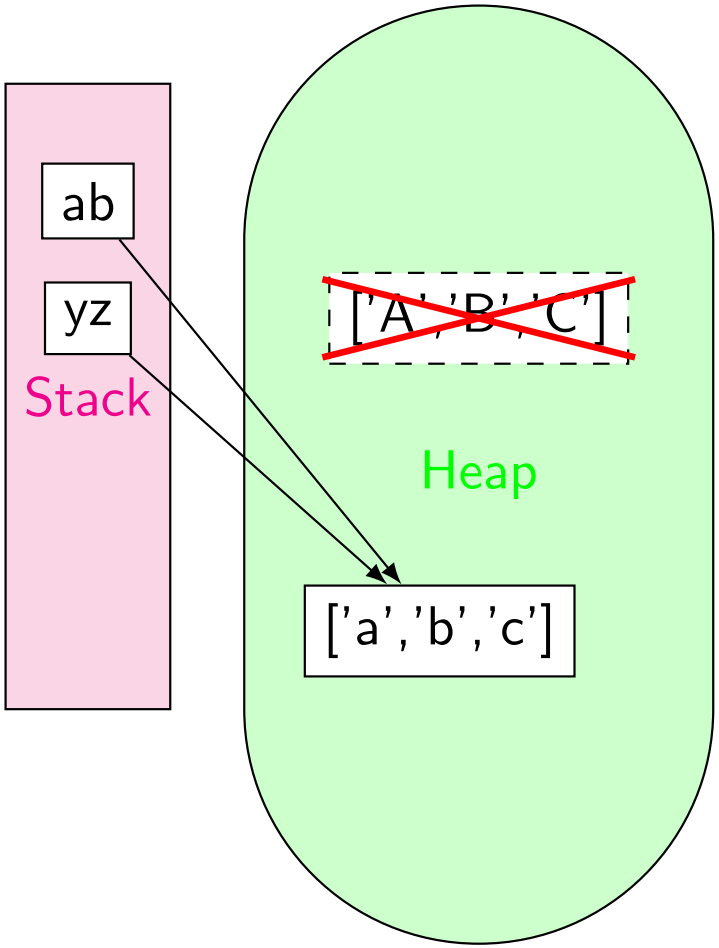
Les variables sont des références : exemple de `String?`




```
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

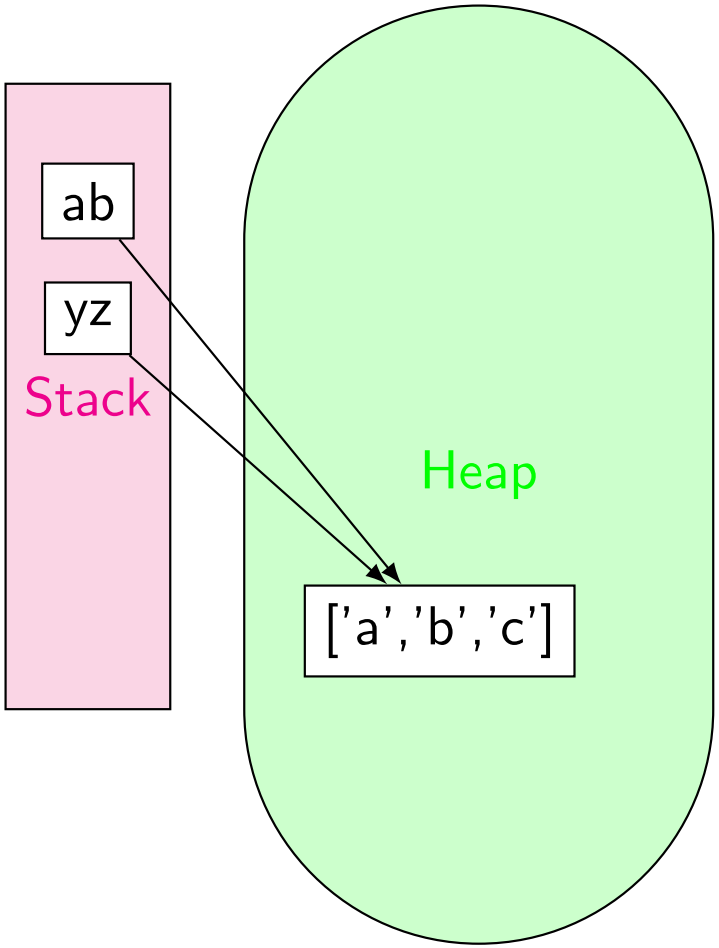
Les variables sont des références : exemple de `String?`




```
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

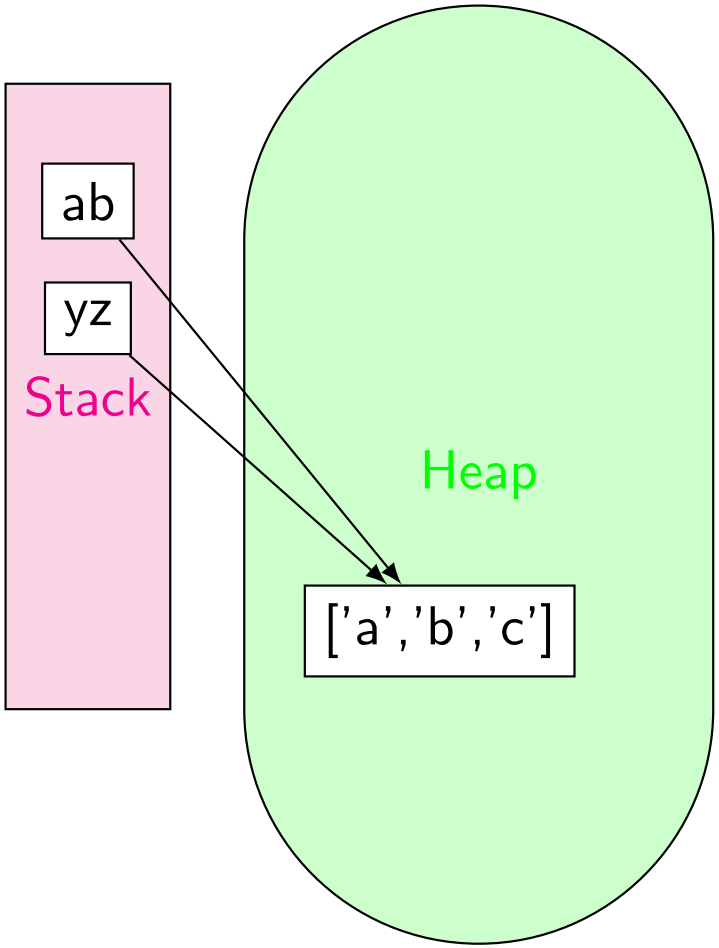
Les variables sont des références : exemple de `String?`




```
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

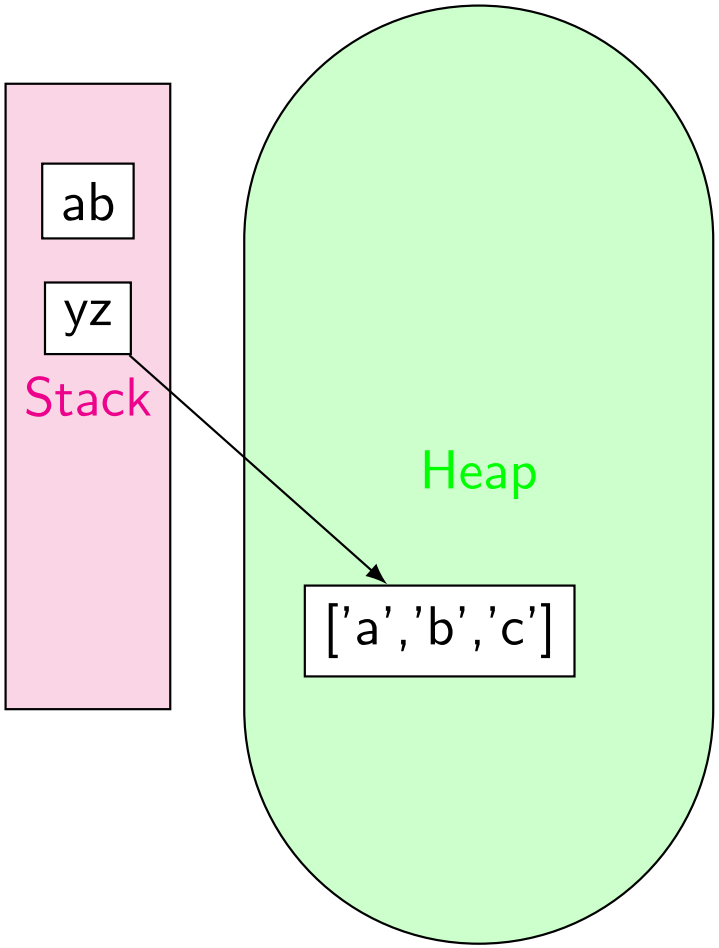
Les variables sont des références : exemple de `String?`




```
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 =

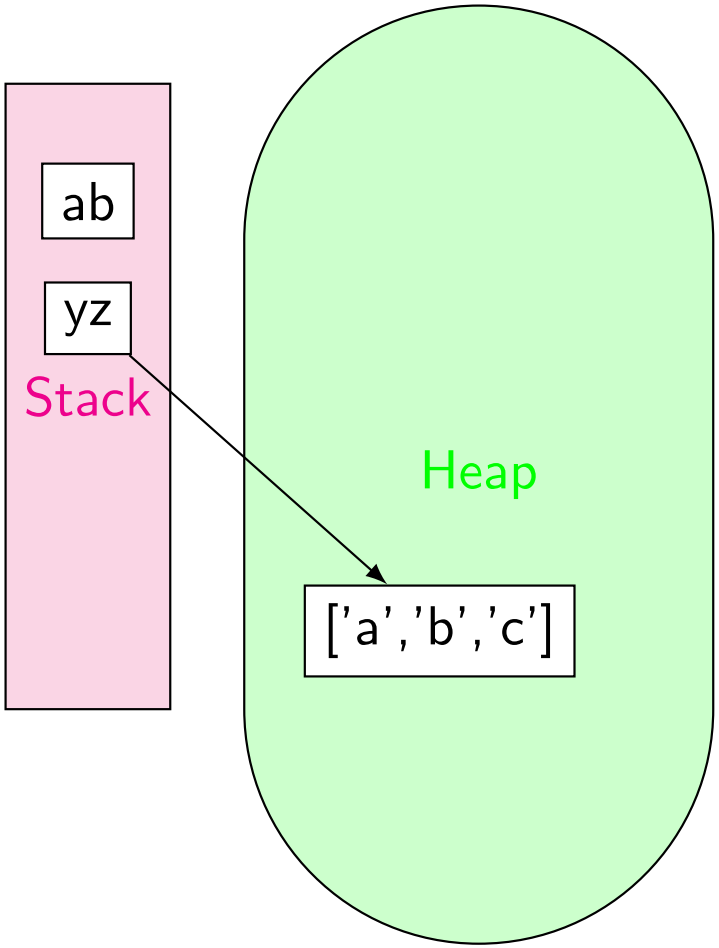
Les variables sont des références : exemple de `String?`




```
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

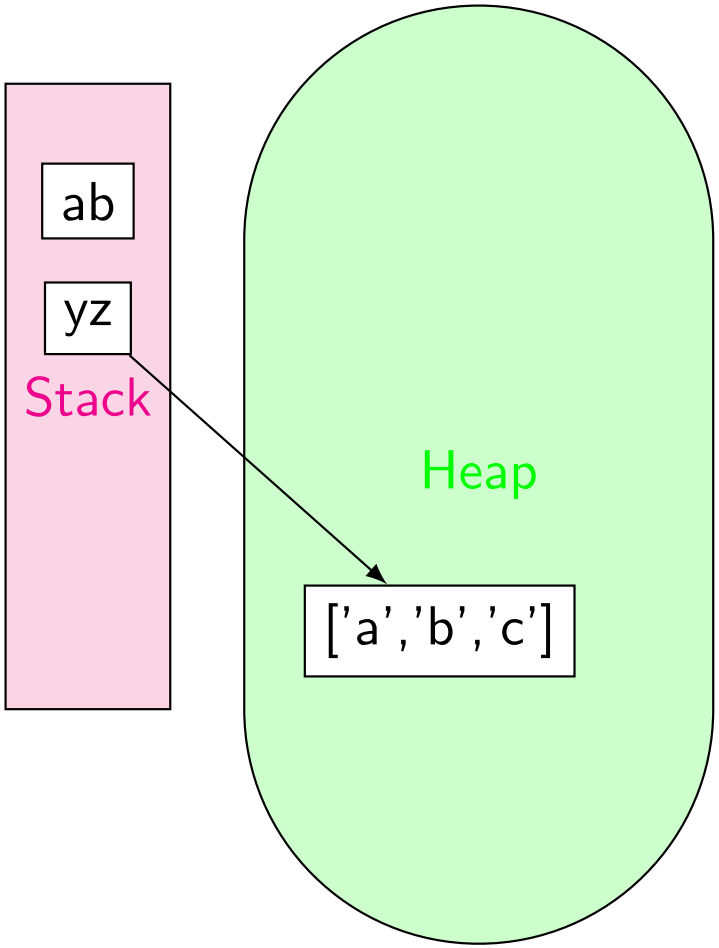
Les variables sont des références : exemple de `String?`




```
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`

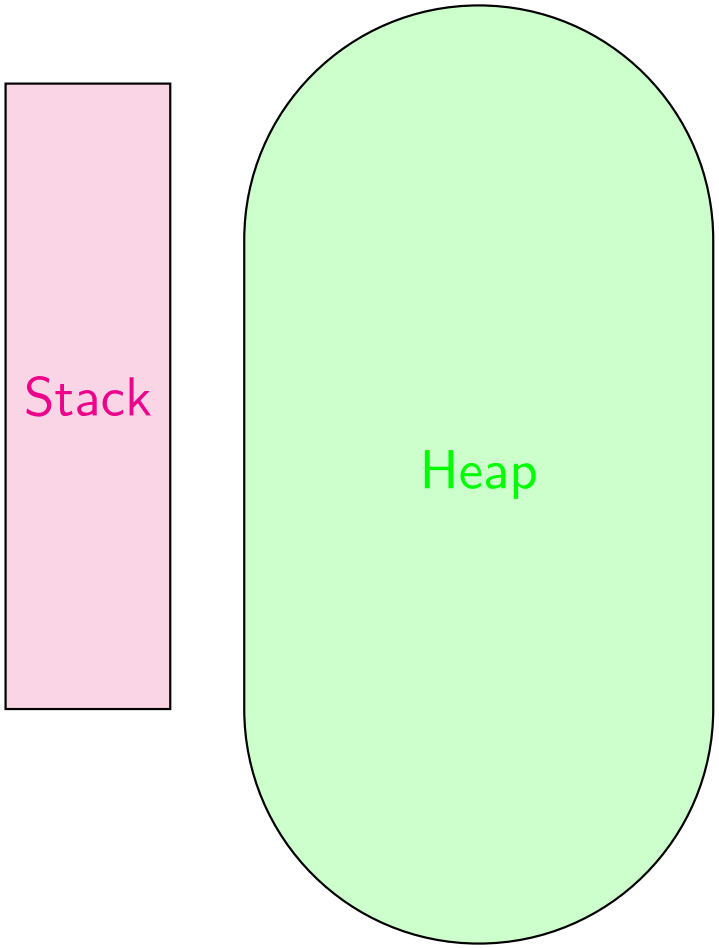
Les variables sont des références : exemple de `String?`




```
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

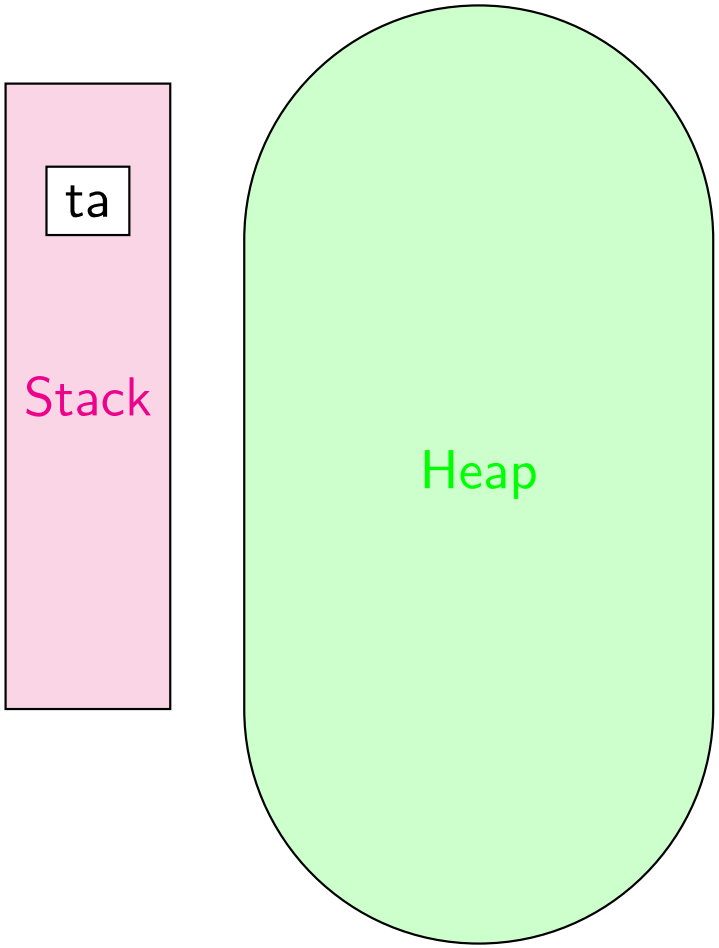
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

Schématisation de la mémoire de la JVM

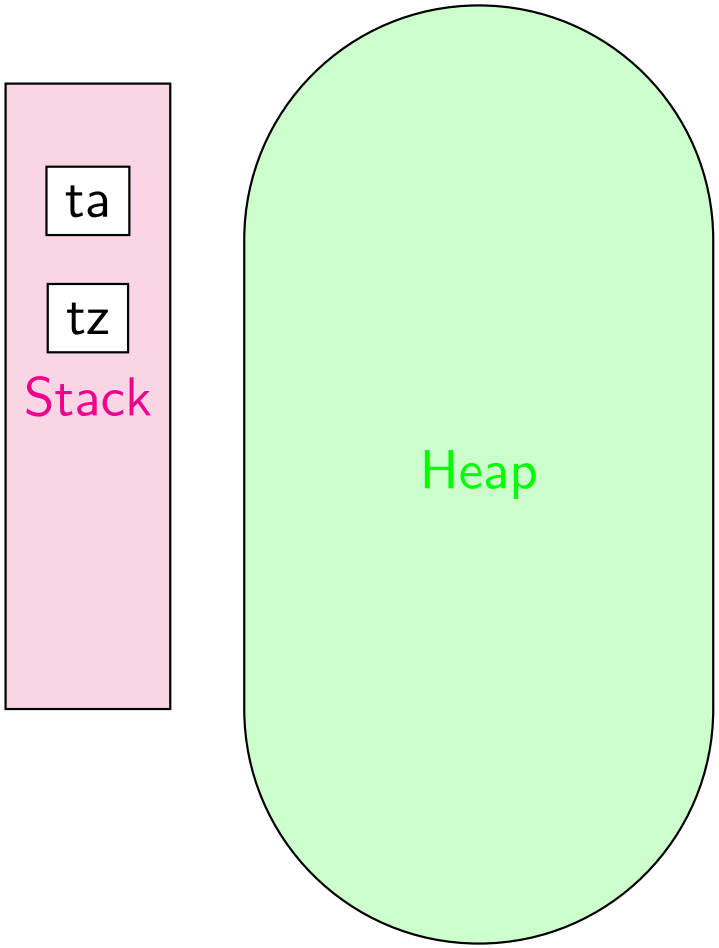
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

`ta` créé dans la pile mémoire

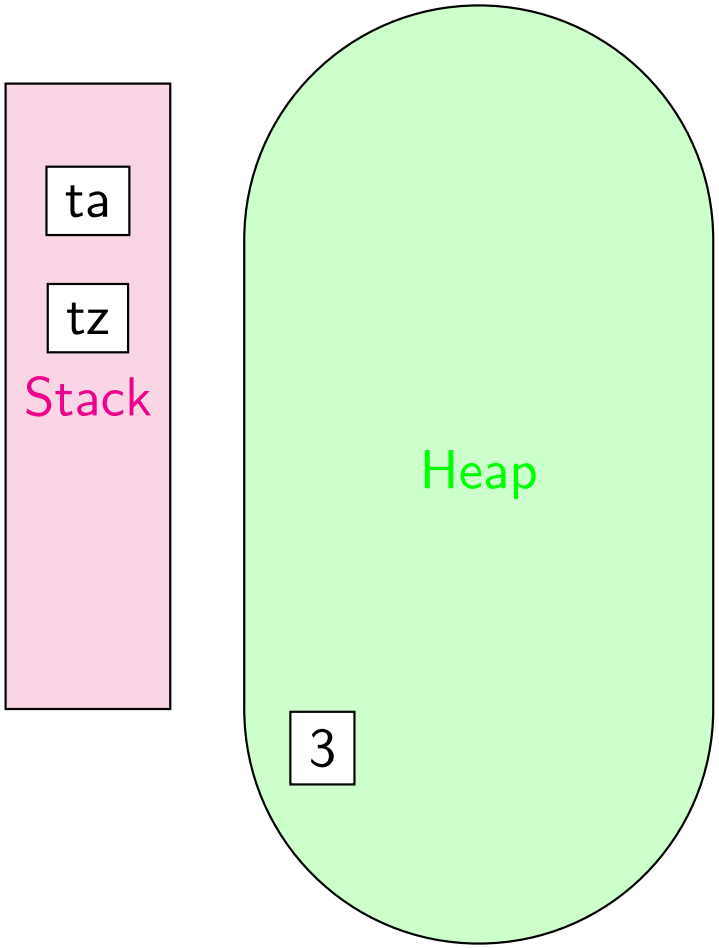
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

`tz` créé dans la pile mémoire

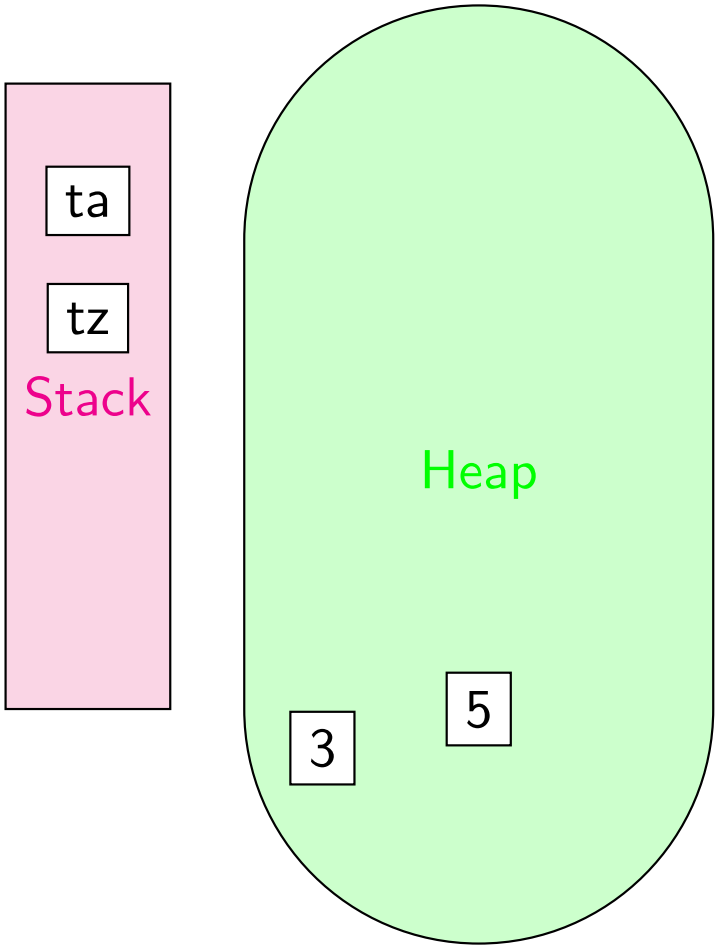
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

3 créé dans le tas mémoire

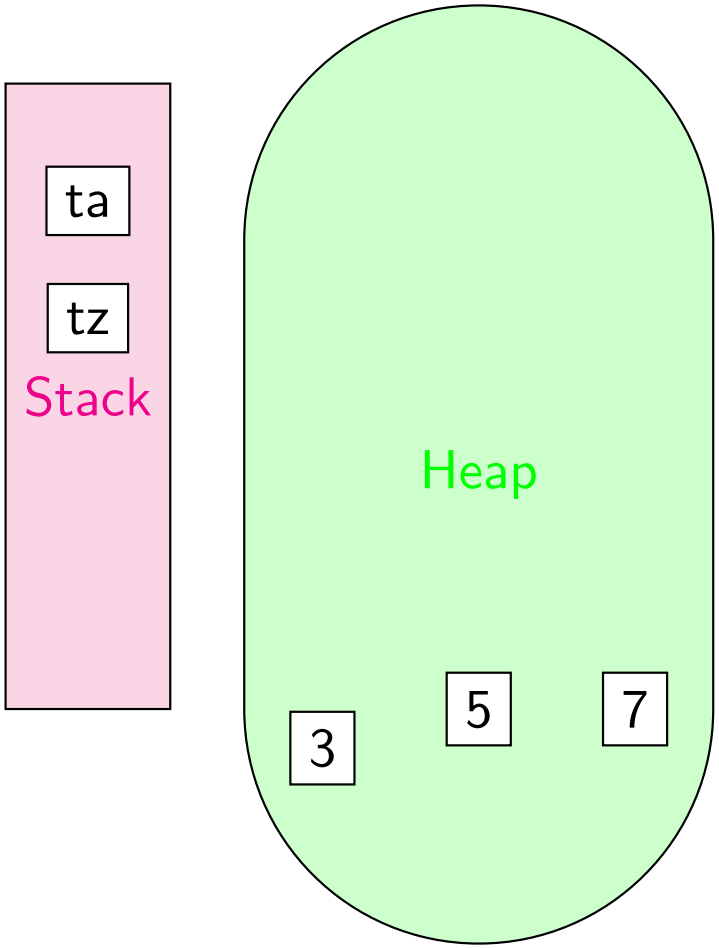
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

5 créé dans le tas mémoire

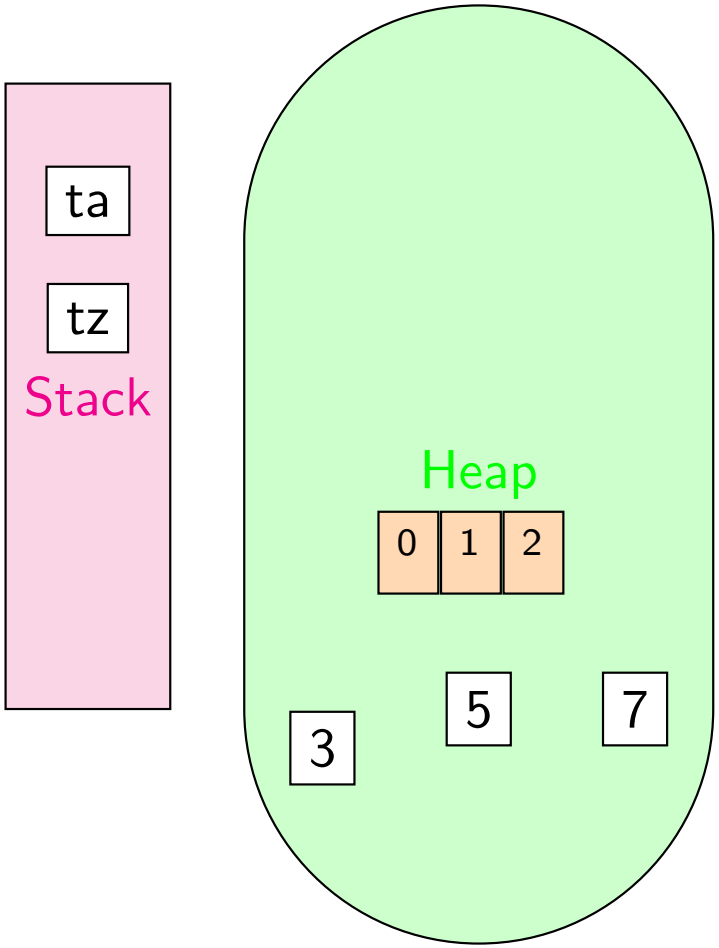
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

7 créé dans le tas mémoire

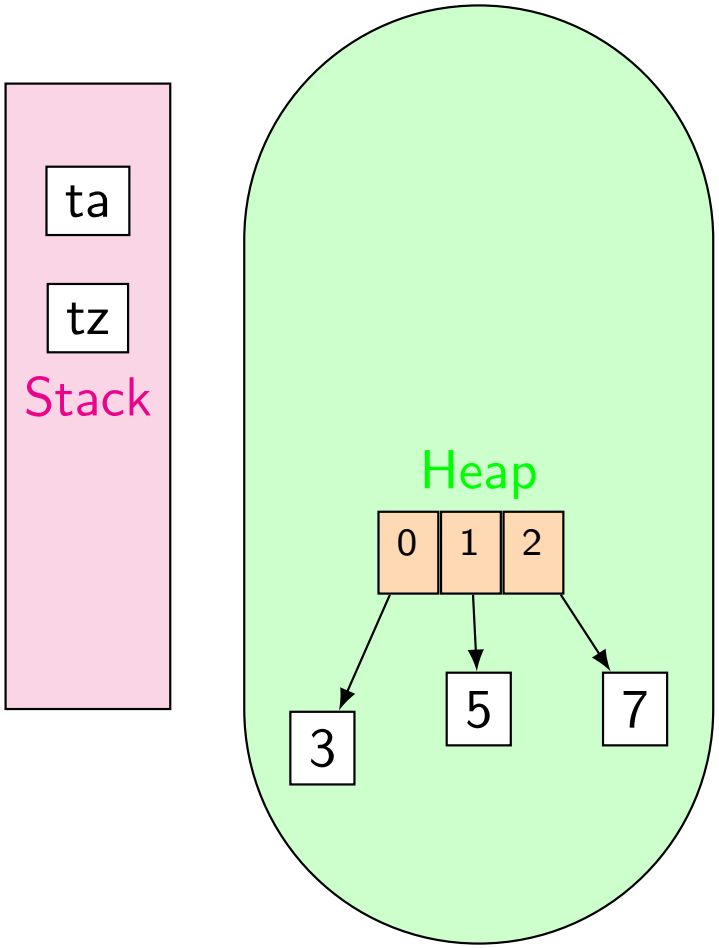
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

un tableau `Array<Int?>(3)` créé dans le tas mémoire

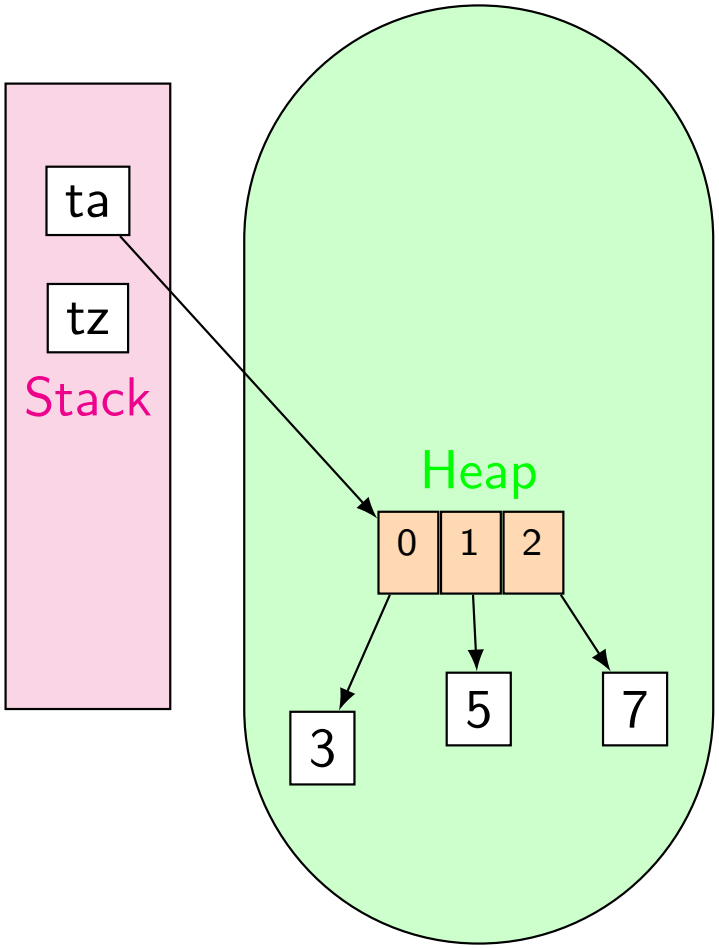
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

Le tableau "pointe" vers les valeurs précédemment créées

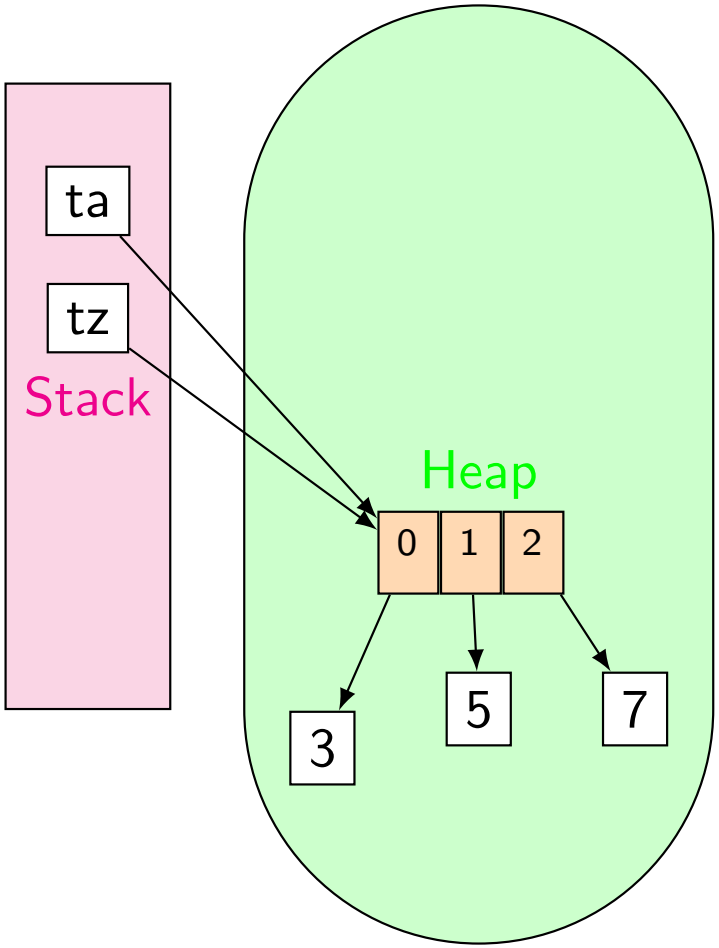
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

`ta` "pointe" vers le tableau

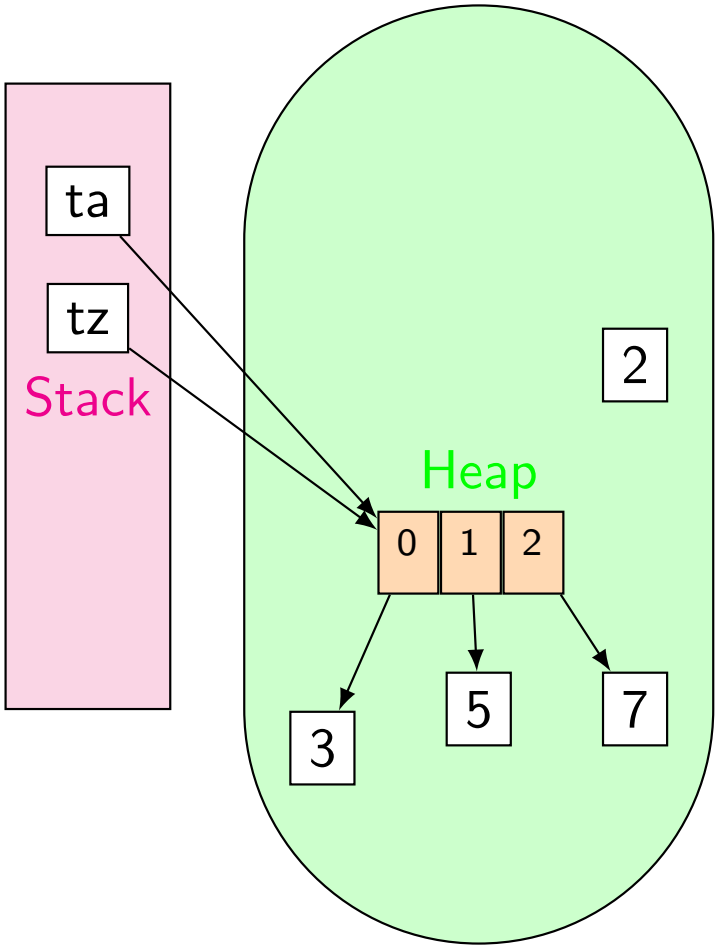
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

`tz` "pointe" aussi vers le tableau

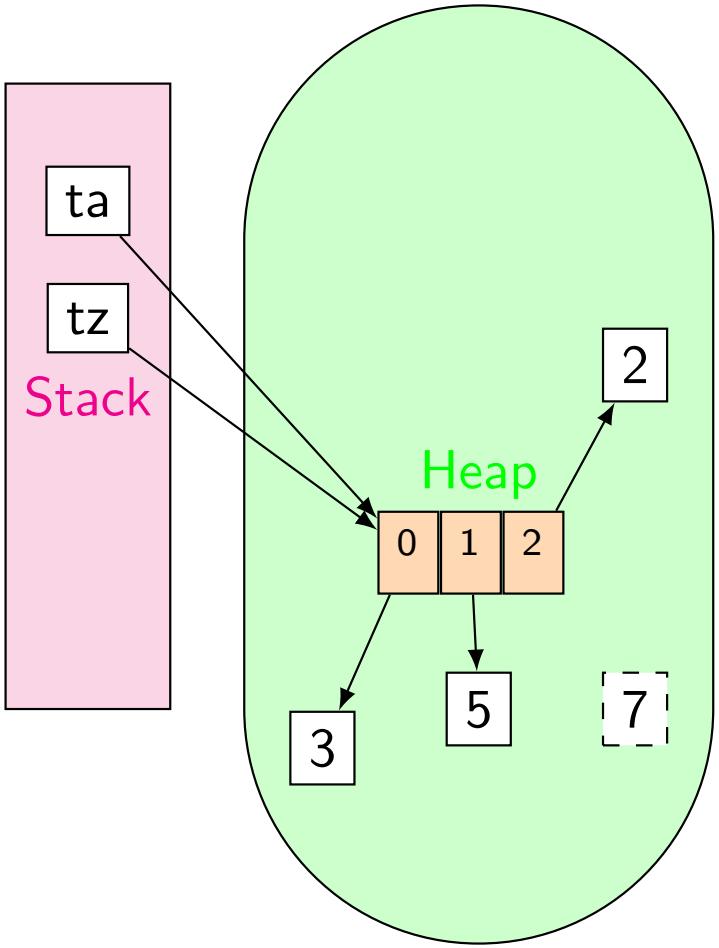
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

2 créé dans le tas mémoire

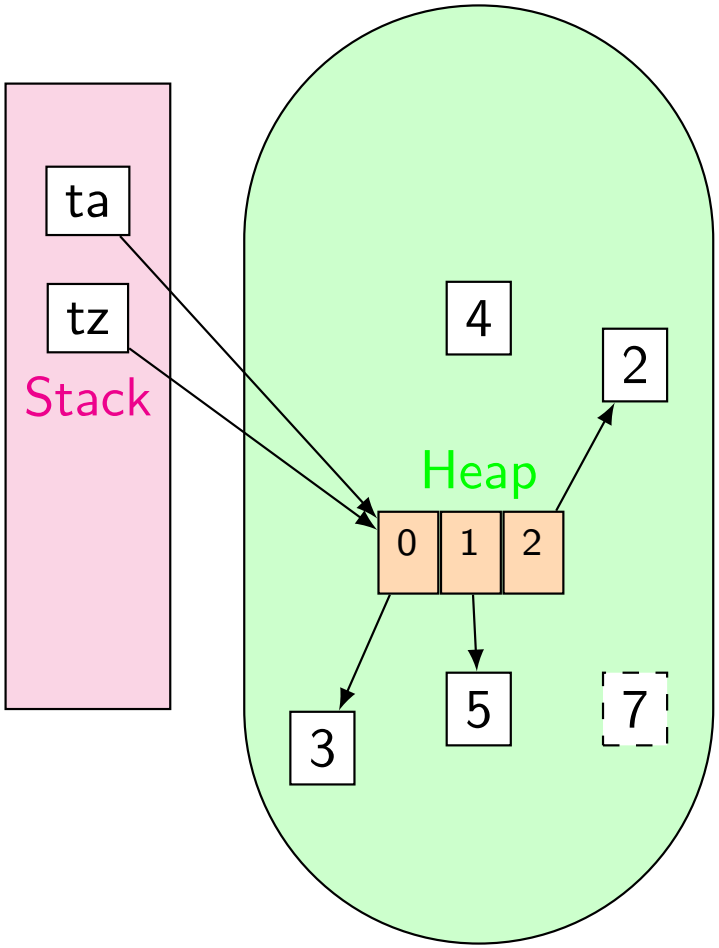
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

le tableau "pointe" maintenant vers 2

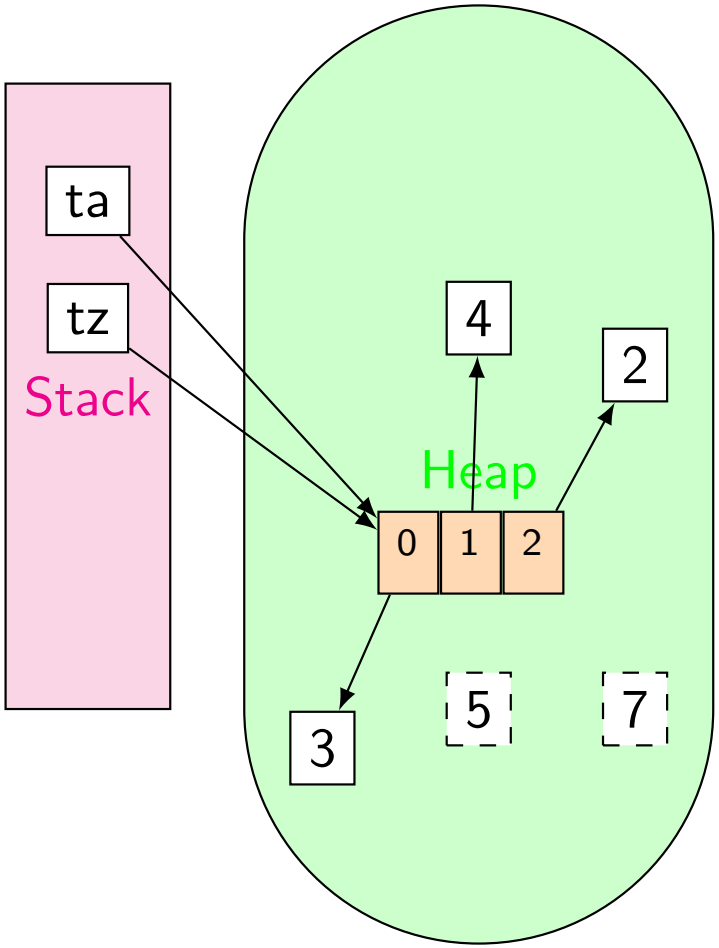
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

4 créé dans le tas mémoire

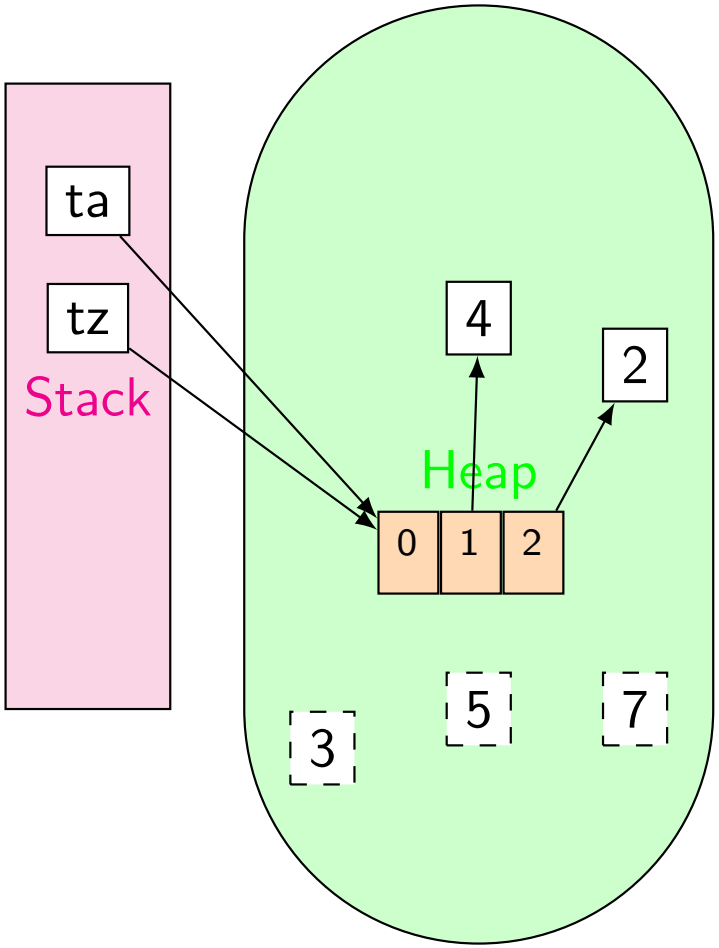
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

le tableau "pointe" maintenant vers 4

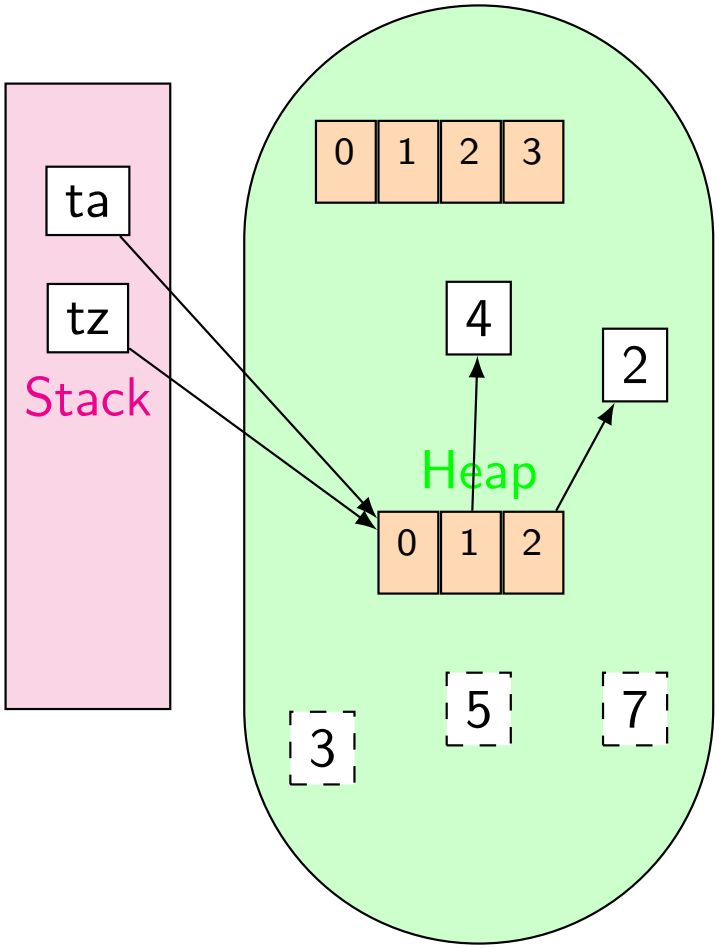
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

3 n'est plus référencé

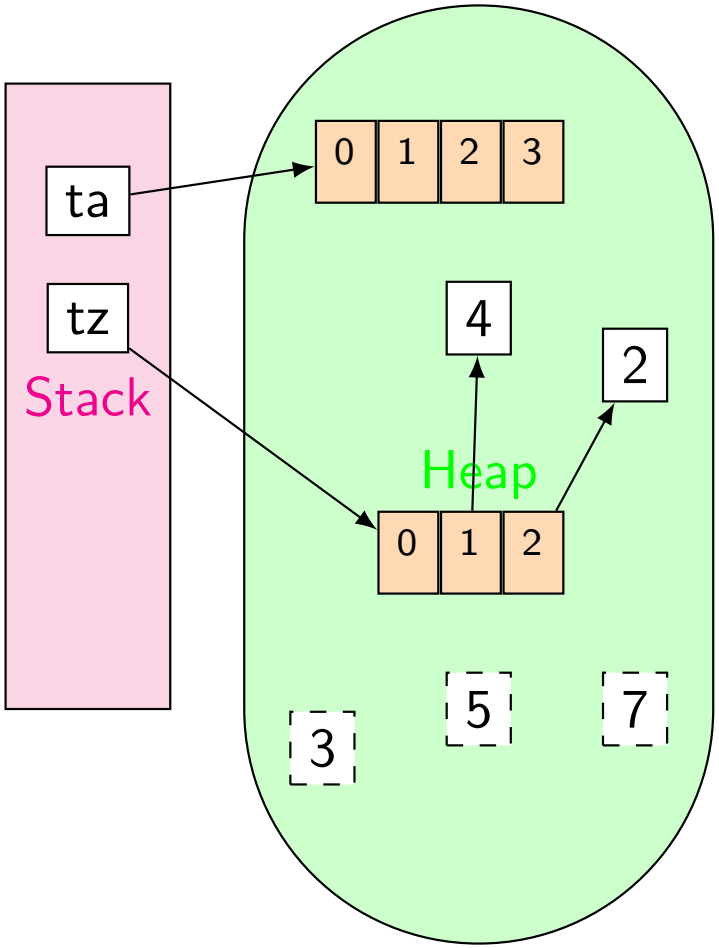
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

Un nouveau tableau `Array<Int?>(4)` est créé

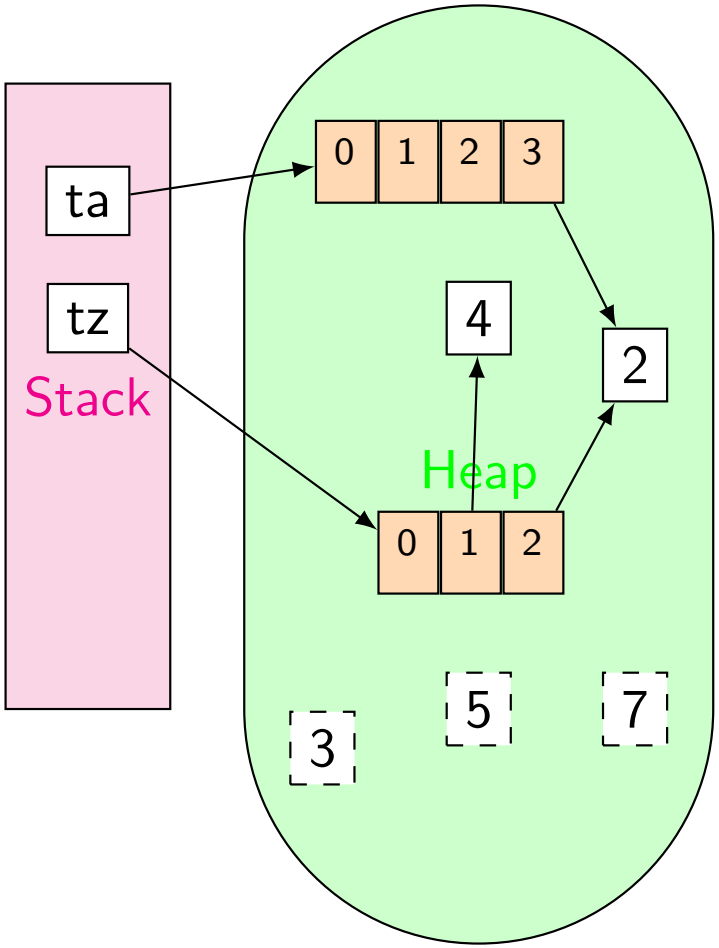
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

`ta` "pointe" vers le nouveau tableau

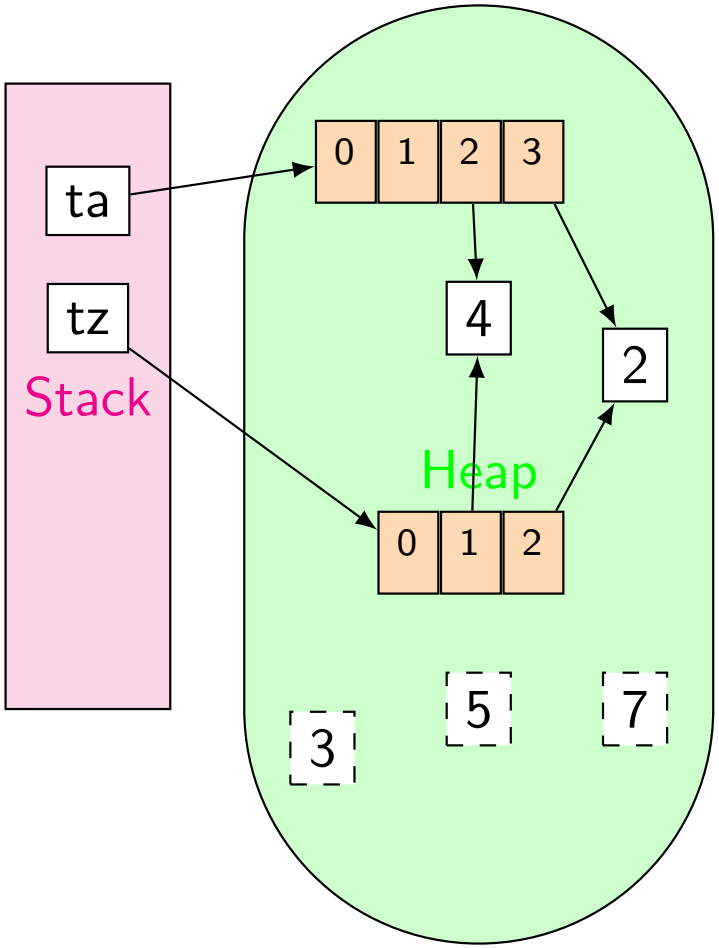
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

le nouveau tableau "pointe" maintenant vers 2

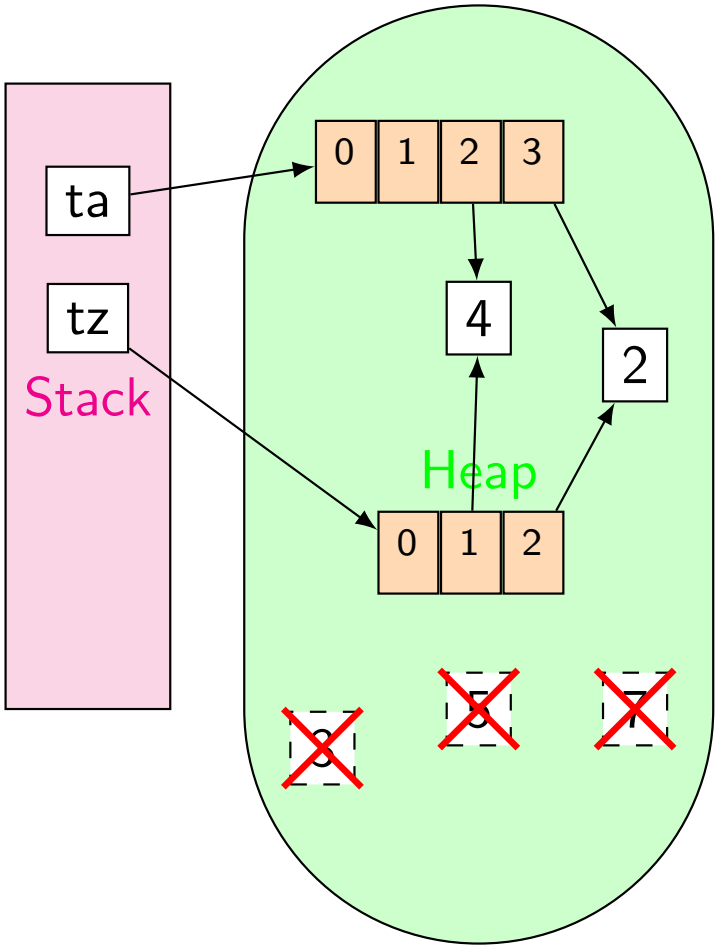
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

le tableau "pointe" maintenant vers 4

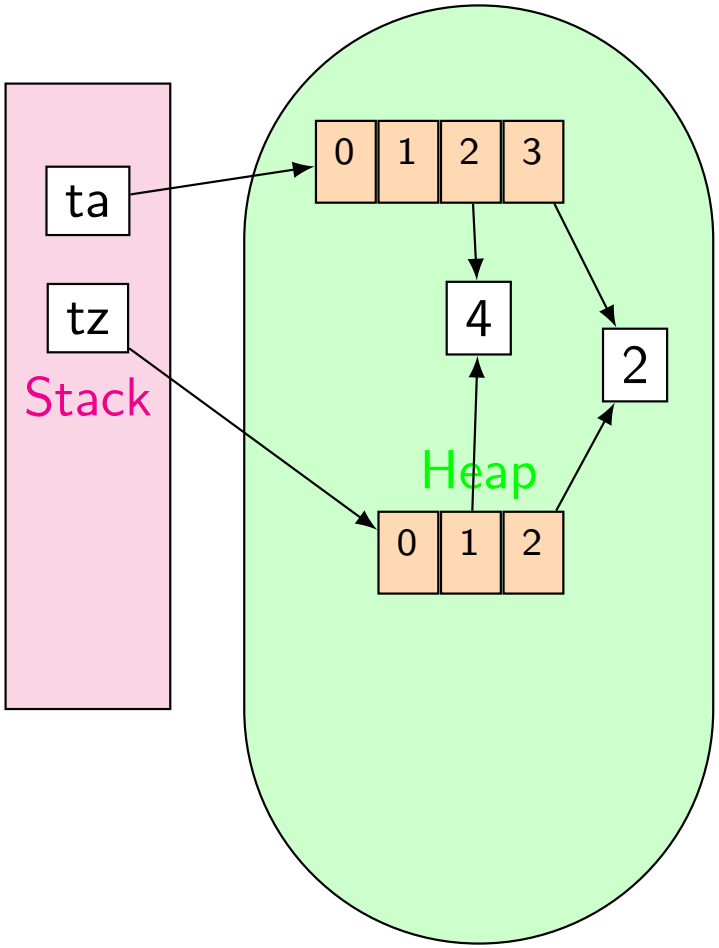
Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

Le garbage collector efface les objets déréférencés

Les tableaux contiennent (aussi) des références




```
var ta : Array<Int?>
var tz : Array<Int?>
ta = arrayOf(3, 5, 7)
tz = ta
ta[2] = 2
tz[1] = 4
tz[0] = null
ta = arrayOfNulls<Int>(4)
ta[3] = tz[2]
ta[2] = tz[1]
```

Le garbage collector efface les objets déréférencés

Utiliser des variables *nullable*

Kotlin interdit d'accéder simplement 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)
```