Chapitre 2
Objets et Java

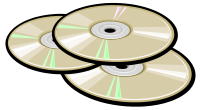
Voir Liskov&Guttag,
ch. 2 « Understanding Objects in Java », pp 15-37

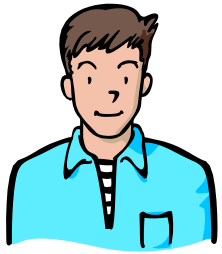
Les objets

- Objet (sens restreint)
  - « morceau » de programme **en cours d'exécution** qui possède
    - un **état** contenant des **données** manipulées par le programme
    - un **comportement** fait d'**opérations** (appelées **méthodes**)
  - *On* interagit avec un objet en invoquant ses méthodes
    - « On » = le code appelant (un objet, une procédure ou le programme principal)
  - Les méthodes d'un objet permettent (entre autres) d'accéder à son état : **lire/consulter** (*Read*, le R de CRUD) ou **écrire/modifier** (*Update,* le U de CRUD)

Exemple: amazon.com

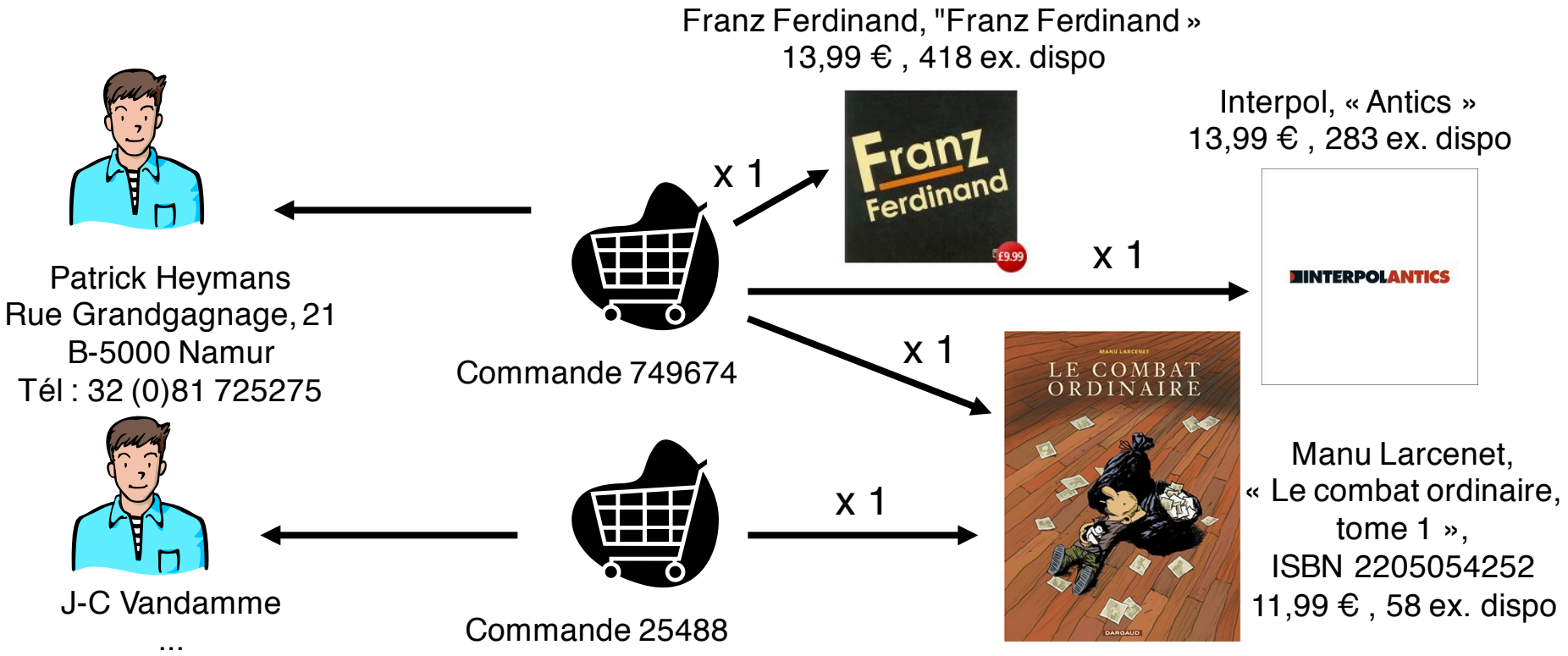
- Ce programme doit maintenir des données sur
  - les articles en vente
    - leur titre, leur(s) auteur(s)/artiste(s), leur descriptif,...
    - le nombre d'exemplaires disponibles en stock
    - leur prix
    - ...
  - les clients
    - leur nom
    - leurs coordonnées
    - ...
  - les commandes
    - les livres commandés et en quelle quantité
    - le client qui a passé la commande
    - ...
  - etc...





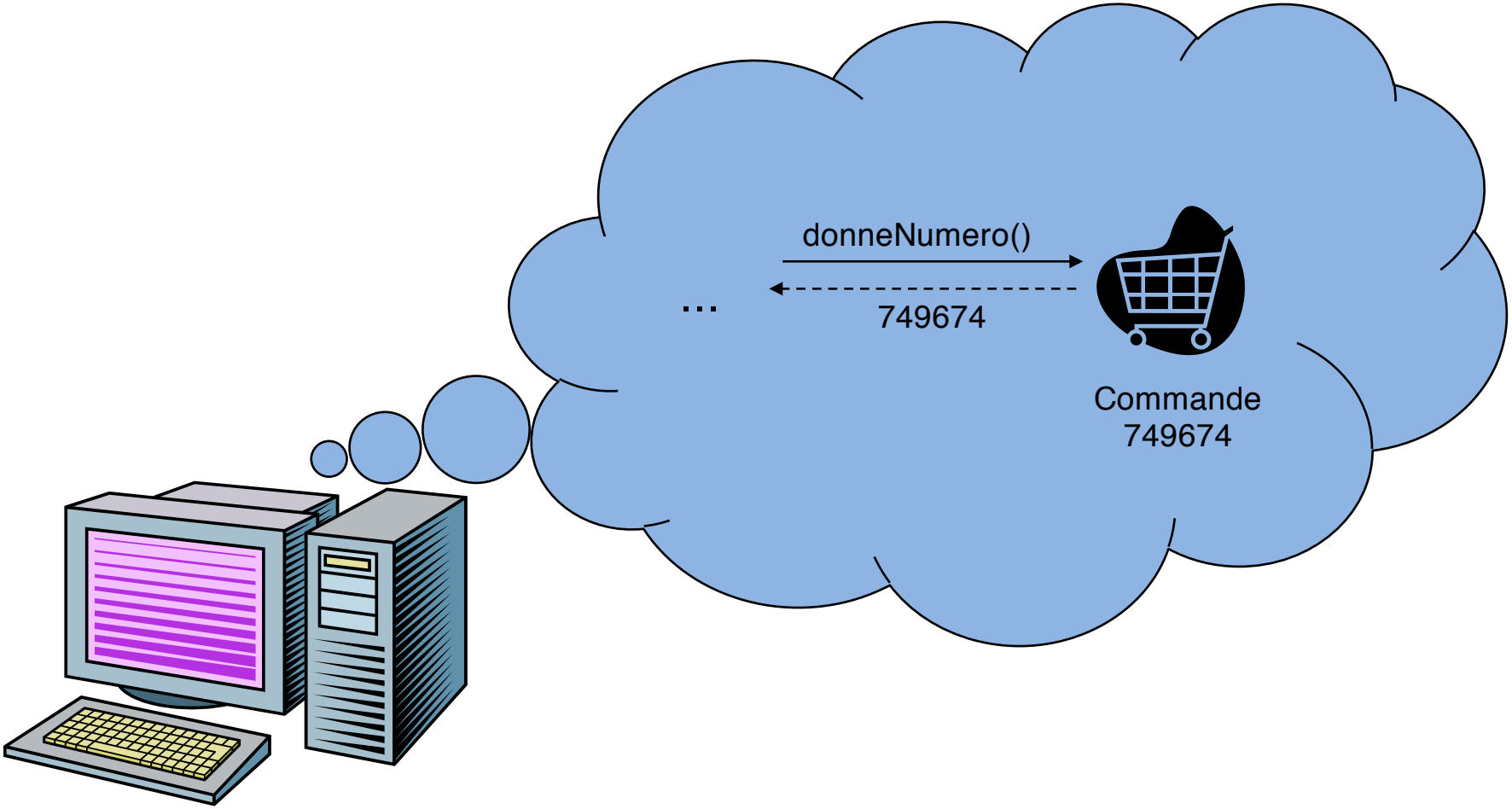
Exemple: amazon.com

- Si ce programme est « orienté-objet », lors de son exécution, on trouvera dans celui-ci une multitude d'objets de chaque **type** (Article, Client, Commande, ...)



Exemple: amazon.com

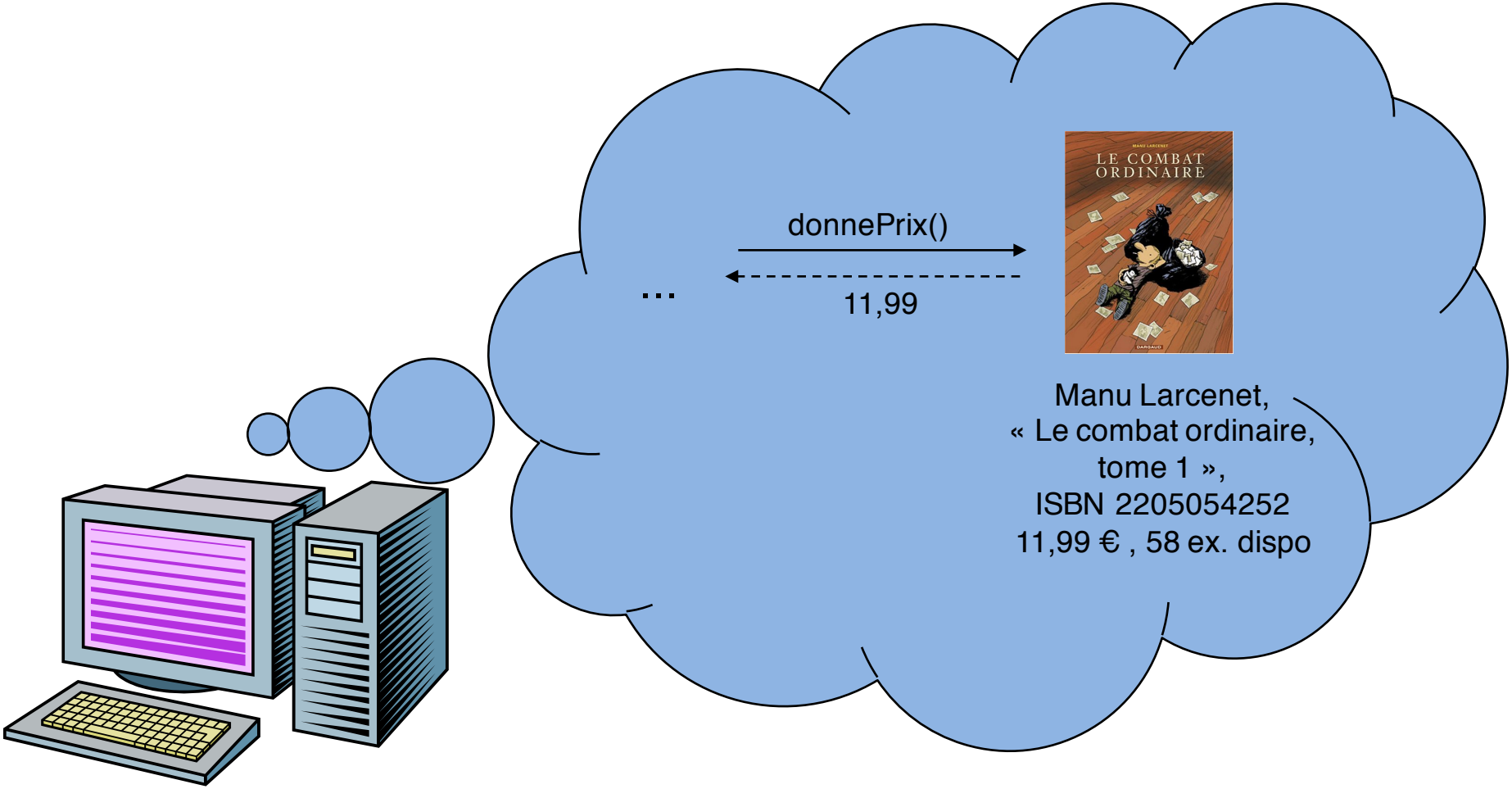
Invocation de méthode



Exemple: amazon.com

Invocation de méthode

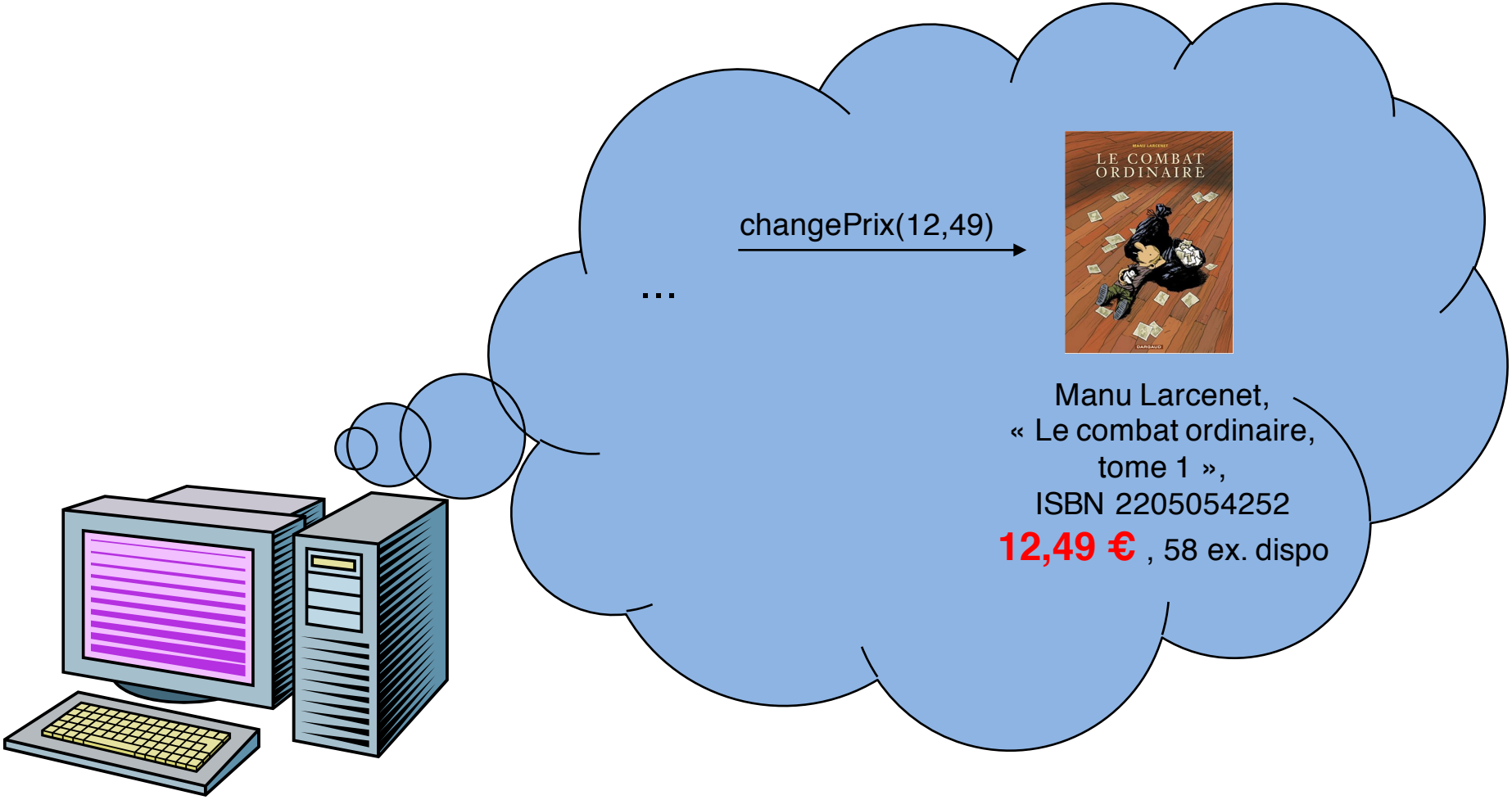
- Lecture



Exemple: amazon.com.

Invocation de méthode

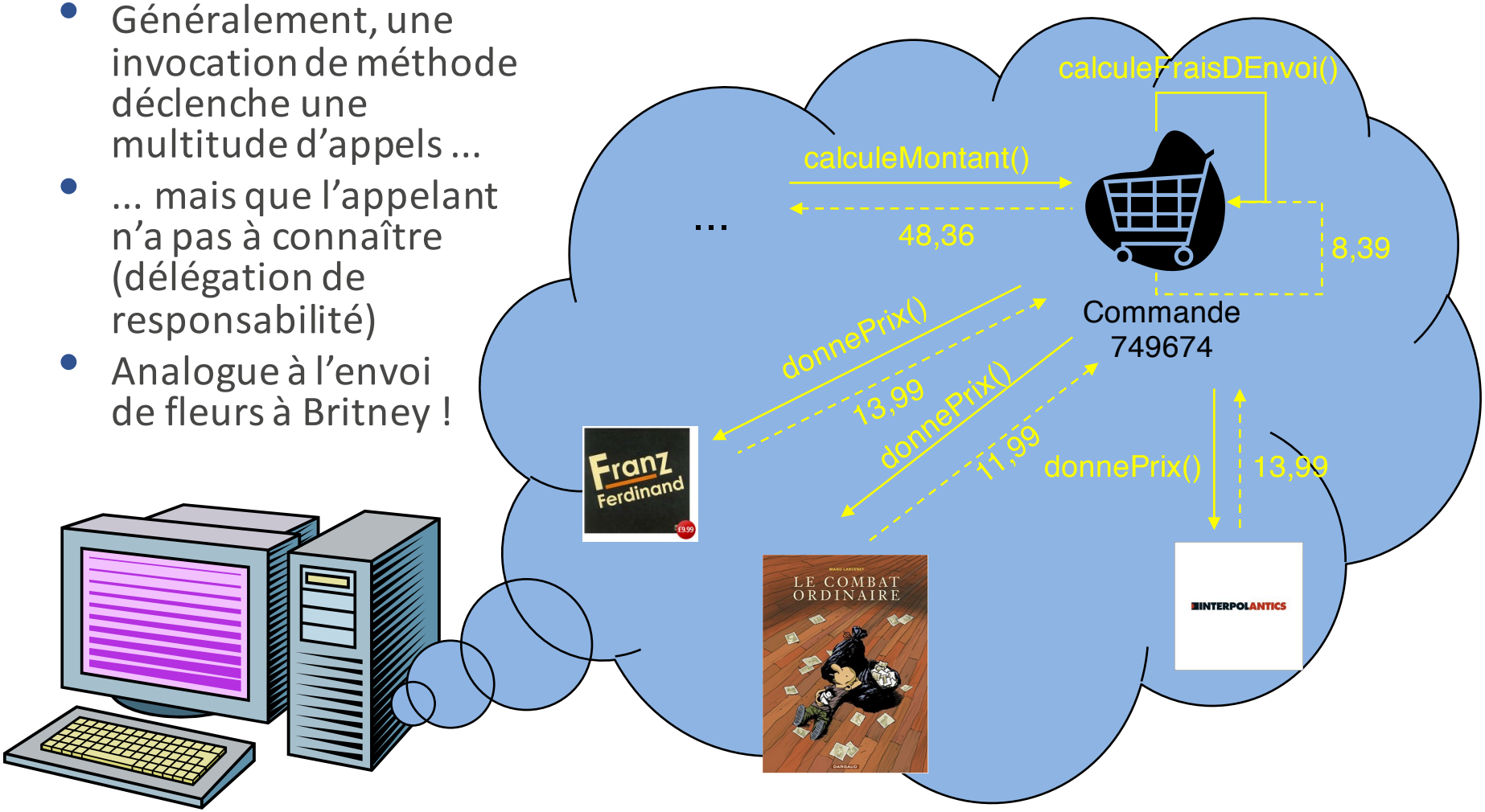
- Ecriture



Exemple: amazon.com

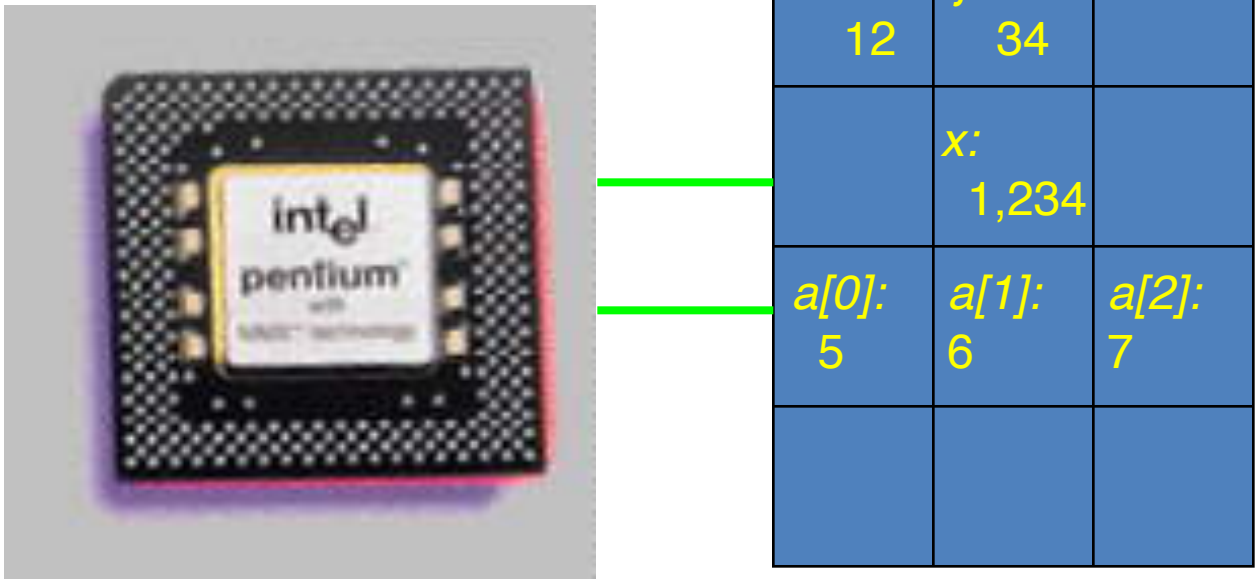
Invocation de méthodes en chaîne

- Généralement, une invocation de méthode déclenche une multitude d'appels ...
- ... mais que l'appelant n'a pas à connaître (délégation de responsabilité)
- Analogue à l'envoi de fleurs à Britney !



Vue de *bas niveau*

- Traditionnellement, l'exécution d'un programme est vue comme ceci : un *processeur* exécute une suite d'instructions machine qui consultent et modifient les éléments d'une *mémoire*



Vue orientée-objet

- Vue de *haut niveau* selon laquelle l'exécution d'un programme est **essentiellement** composée d'**interactions entre objets**
- Les objets **représentent** (= maintiennent des informations sur) **souvent** des **entités du monde réel** (personnes, choses, concepts) issues du domaine d'application du programme

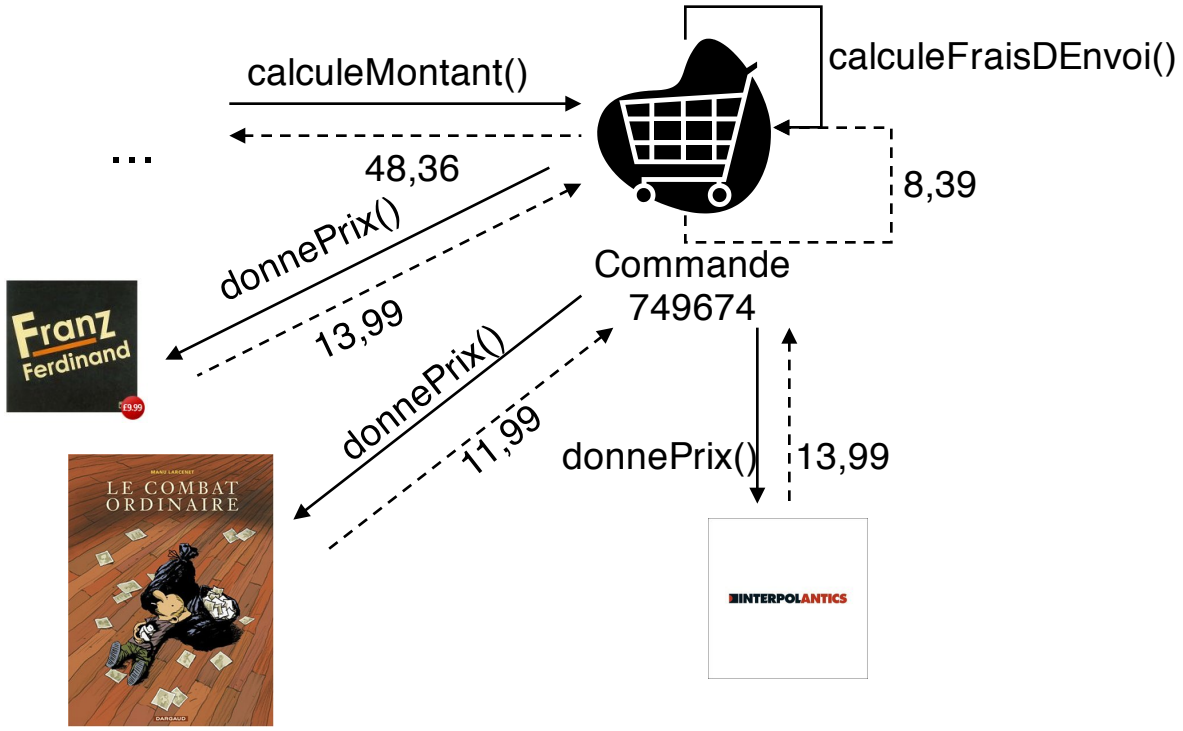
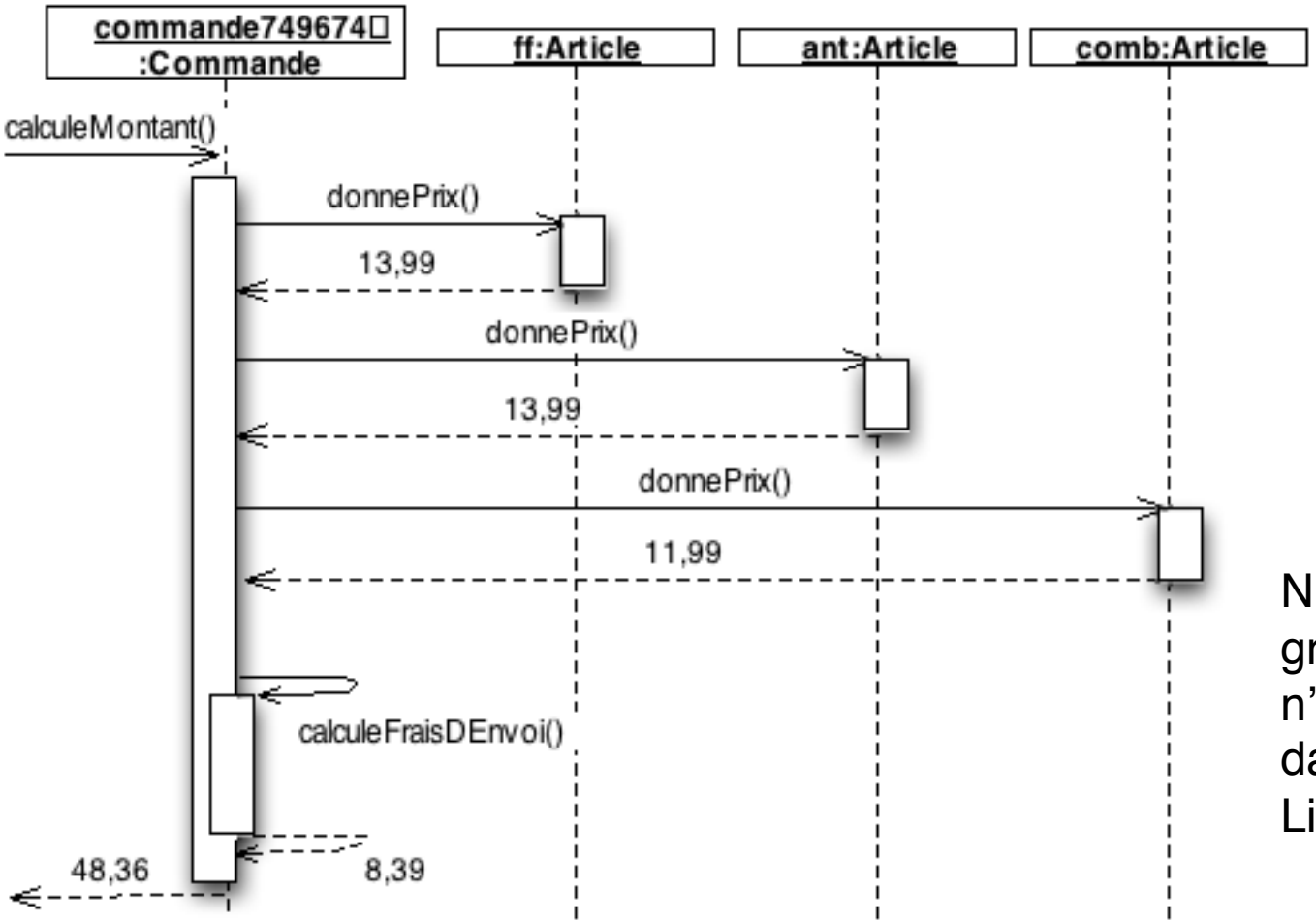


Diagramme de séquence

Un « bout d'exécution » de programme est souvent illustré par un diagramme de séquence



NB: ces notations graphiques n'apparaissent pas dans le livre de Liskov et Guttag

L'orienté-objet (OO)

- Programme OO

  = programme **pensé selon la vue OO**

- Conception et Programmation OO

  = résoudre un problème par le biais d'un programme OO (i.e. pensé OO)

  - concevoir → structurer (via décomposition et abstraction)
  - programmer → écrire le code

- Langage (de programmation) OO

  = langage possédant des mécanismes qui **facilitent** la programmation OO

  - on peut développer non-OO dans un langage OO
  - on peut développer OO dans un langage non-OO (mais c'est plus facile dans un langage OO!)

Java

- **Java** est un langage de programmation OO
  - non OO : C, Pascal, Lisp, Prolog, Ada,...
  - OO : C++, C#, Eiffel, Smalltalk, Simula, Python, PHP,... + extensions OO de langages non OO
- Influencé par les langages OO et des langages non OO
  - Esprit : Lisp, Simula67, CLU, SmallTalk,...
  - Forme (syntaxe) : C, C++
- A influencé d'autres langages
  - Python, C#

Et vous dans tout ça

- Au fil de ce cours, nous allons
  - comprendre et mettre en pratique les **principes de base** de la conception et de la programmation OO
  - comprendre et utiliser les mécanismes que Java met au service de ces principes
- Objectif : vous faire devenir de bon concepteurs/programmeurs OO
  - qui **maîtrisent les principes de l'OO** et peuvent les **appliquer dans de nombreux langages**
- Pas un objectif : faire de vous des hackers Java
  - *en effet, les langages passent, mais les principes restent...*

Classes

- Les programmes Java sont composés de définitions de **classes** (et d'interfaces)

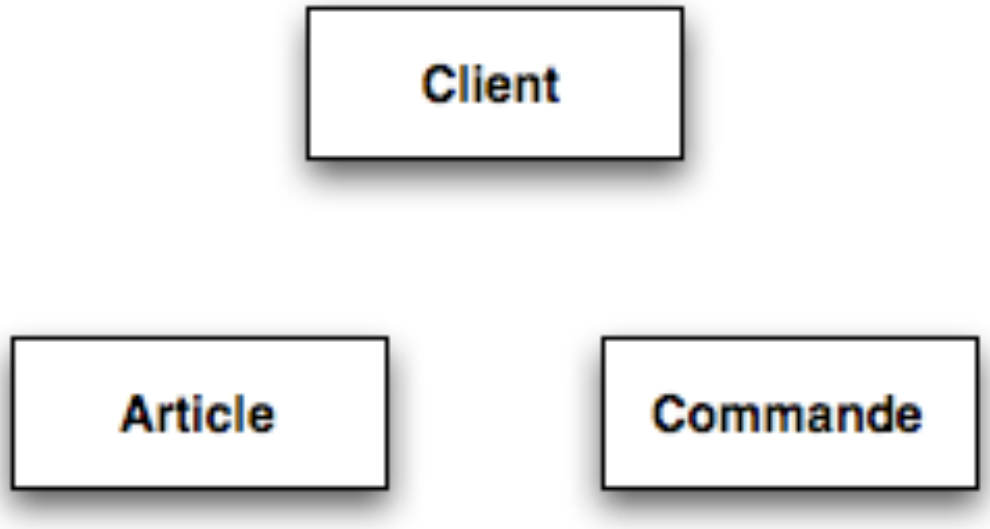
```
class Client {
   ...
   ...
}

class Article {
   ...
   ...
}

class Commande {
   ...
   ...
}
```

Classe = élément d'un programme
Objet = élément d'une exécution

A tout moment de l'exécution, chaque classe du programme peut avoir plusieurs objets (instances)

Diagrammes de classes

- On représentera graphiquement les classes d'un programme dans des Diagrammes de Classes UML (Class Diagrams)



NB: ces notations graphiques n'apparaissent pas dans le livre de Liskov et Guttag

Classes
Utilités

Une classe a deux utilités possibles

- définir des **collections de procédures/méthodes/opérations**

  (abstraction procédurale)

- définir des **types de données**

  (abstraction de données)

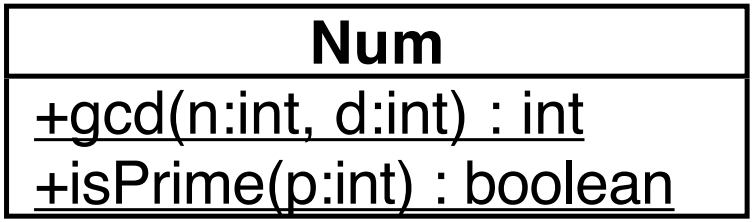
Utilisation des classes pour l'abstraction procédurale

Exemple 1 : la classe `Num` qui rassemble diverses méthodes de calcul numérique

```
class Num {
  public static int gcd(int n, int d) {
  // pré : n>0, d>0
  // post : retourne le pgcd de n et d
    while (n != d)
       if (n > d) n = n - d; else d = d - n;
    return n;
  }

  public static boolean isPrime(int p) {
  // pré : p>0
  // post : retourne true si p est premier,
  // retourne false sinon
    ...
  }
...
}
```

Diagramme de classe

- Les méthodes **`static`** sont soulignées
- Les méthodes **`public`** (visibles des autres classes) sont précédées d'un **+**



Utilisation des classes pour l'abstraction procédurale

Des exemples d'appels à ces méthodes pourraient être :

```
int x = Num.gcd(15,6);
if (Num.isPrime(y)) ...
```

Le nom de la méthode appelée est précédé du nom de la classe dans laquelle elle est définie suivi d'un . (point)

Utilisation des classes pour l'abstraction procédurale

Exemple 2 : la classe `Arrays` qui rassemble diverses méthodes de manipulation de tableaux d'entiers

```
class Arrays {
  public static void sort(int[] a) {
  // post : a est trié en ordre croissant
     ...
  }

  public static int search(int[] a, int el) {
  // post : retourne -1 si el n'apparaît pas dans a;
  // sinon retourne i tq a[i]=el et qu'il n'y a pas de j<i tq a[j]=el
     ...
  }
...
}
```

- `int[]` designe un tableau d'entiers de longueur indéterminée
- L'absence de valeur de retour s'indique par le « type » `void`

Diagramme de classe

Utilisation des classes pour l'abstraction procédurale

- Exemples d'appels:

```
int[] monTableau;
// initialisation de monTableau
...
Arrays.sort(monTableau);
int i = Arrays.search(monTableau,3);
```

Utilisation des classes pour l'abstraction de données

Exemple 1 : la classe `MultiSet`

```
class MultiSet {
  public void insert(int e) {
     ...
  }
  public void delete(int e) {
     ...
  }
  public int size() {
     ...
  }

  public void union(Multiset m) {
     ...
  }
...
}
```

`MultiSet` devient un nouveau type de données pouvant être utilisé dans le programme

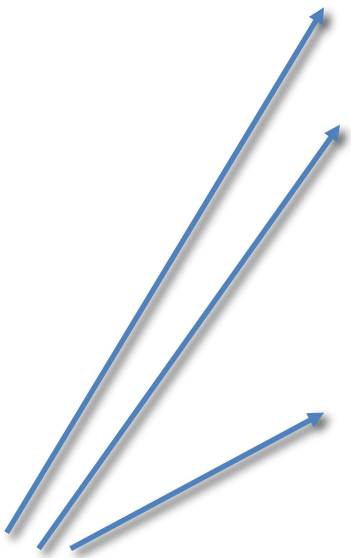
Utilisation des classes pour l'abstraction de données

- Attention
  - Pas de `static` dans les déclarations de méthodes de types de données
  - Pas de `MultiSet` en paramètres ni en type de retour alors que les opérations de l'ADT (voir p. 40) en avaient.
    Pourquoi ?

Utilisation des classes pour l'abstraction de données

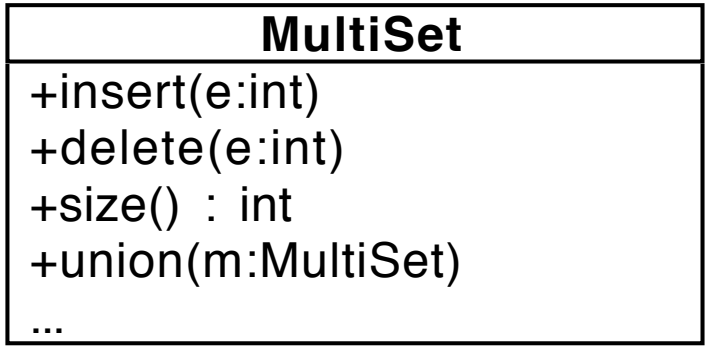
Exemples d'appels:

```
MultiSet m = ...; // initialisation de
m...
m.insert(9);
...
m.delete(9);
...
MultiSet n = ...; // initialisation de
n...
m.union(n);
```



appels sur une instance
et pas sur une classe

Diagramme de classe



- Les méthodes non **`static`** ne sont pas soulignées

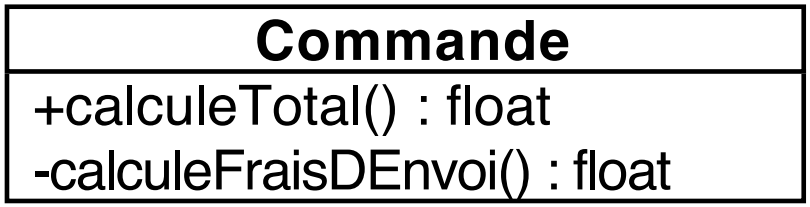
Utilisation des classes pour l'abstraction de données

- Exemple 2 : la classe `Commande` de Amazon.com

```
class Commande{
  public float calculeMontant() {
    ...
  }
  private float calculeFraisDEnvoi() {
    ...
  }
...
}
```

- `Commande` devient un nouveau type de données pouvant être utilisé dans le programme
- Idem pour `Client`, `Article`,...
- Les méthodes **in**visibles pour les autres classes sont précédées de `private`
- Une méthode `private` (privée) est dite *encapsulée* dans la classe

Diagramme de classe



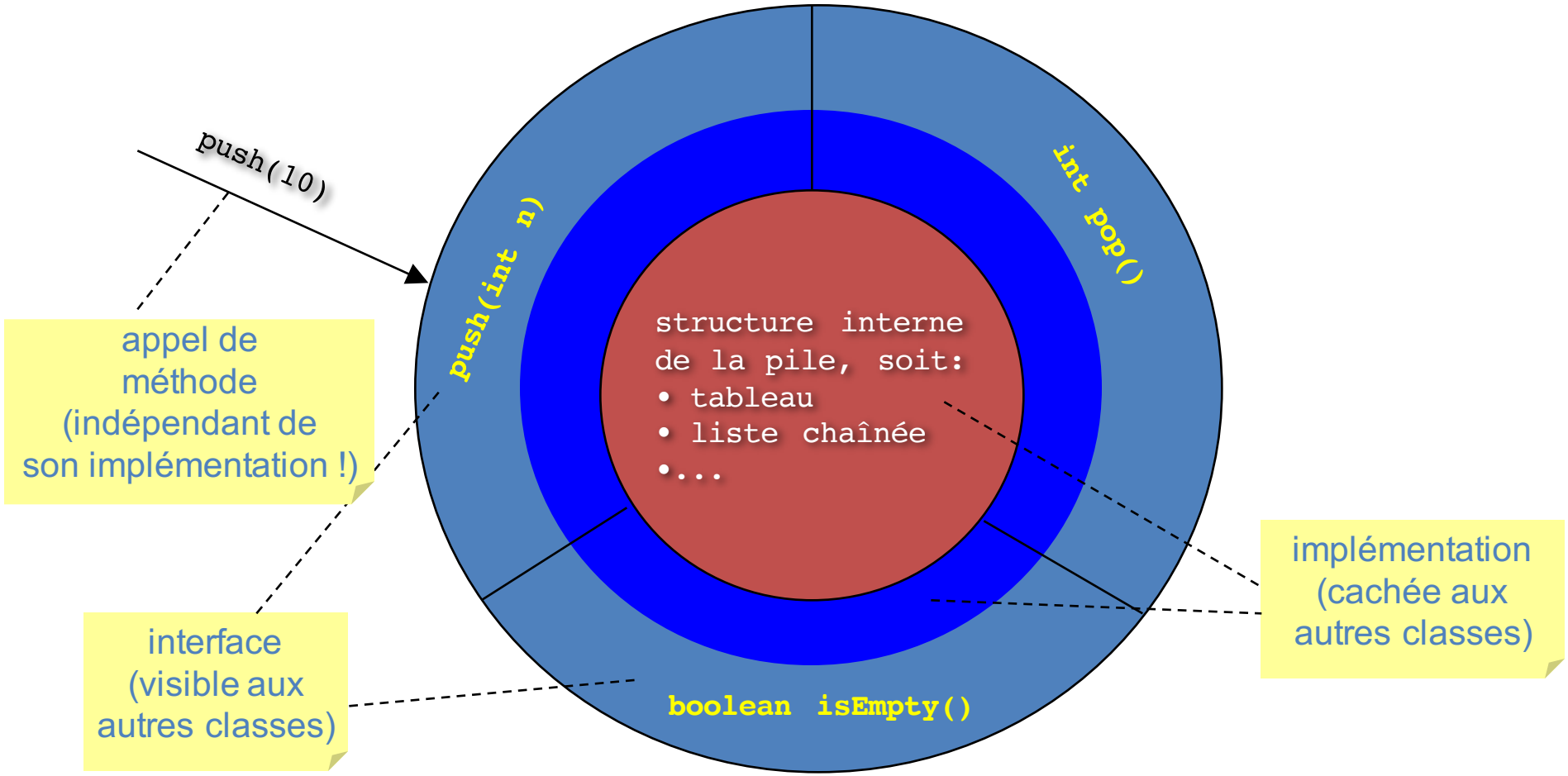
- Les méthodes **`private`** sont précédées d'un **–**

Encapsulation

- Les **types abstraits** (classes d'abstraction de données) sont dotés
  - d'une **interface** spécifiant les méthodes qui peuvent être exécutées par une instance du type et qui représentent ses **responsabilités**
  - d'une **implémentation** précisant
    - les **structures de données** (qui encodent l'état d'une instance)
    - le **corps des méthodes**
    - des **méthodes auxiliaires**
- L'abstraction de données est une forme d'**encapsulation** :
  - seule **l'interface** d'un objet est **accessible** depuis son environnement
  - **l'implémentation** est **cachée**

Encapsulation

- Exemple : le type `Pile`



Principe de Parnas

- « La définition d'une classe doit fournir à l'environnement toutes les informations nécessaires pour manipuler correctement une instance de la classe, et *rien d'autre.* »

- « L'implémentation d'une méthode doit se baser sur toutes les informations nécessaires à l'accomplissement de sa tâche, et sur *rien d'autre* »

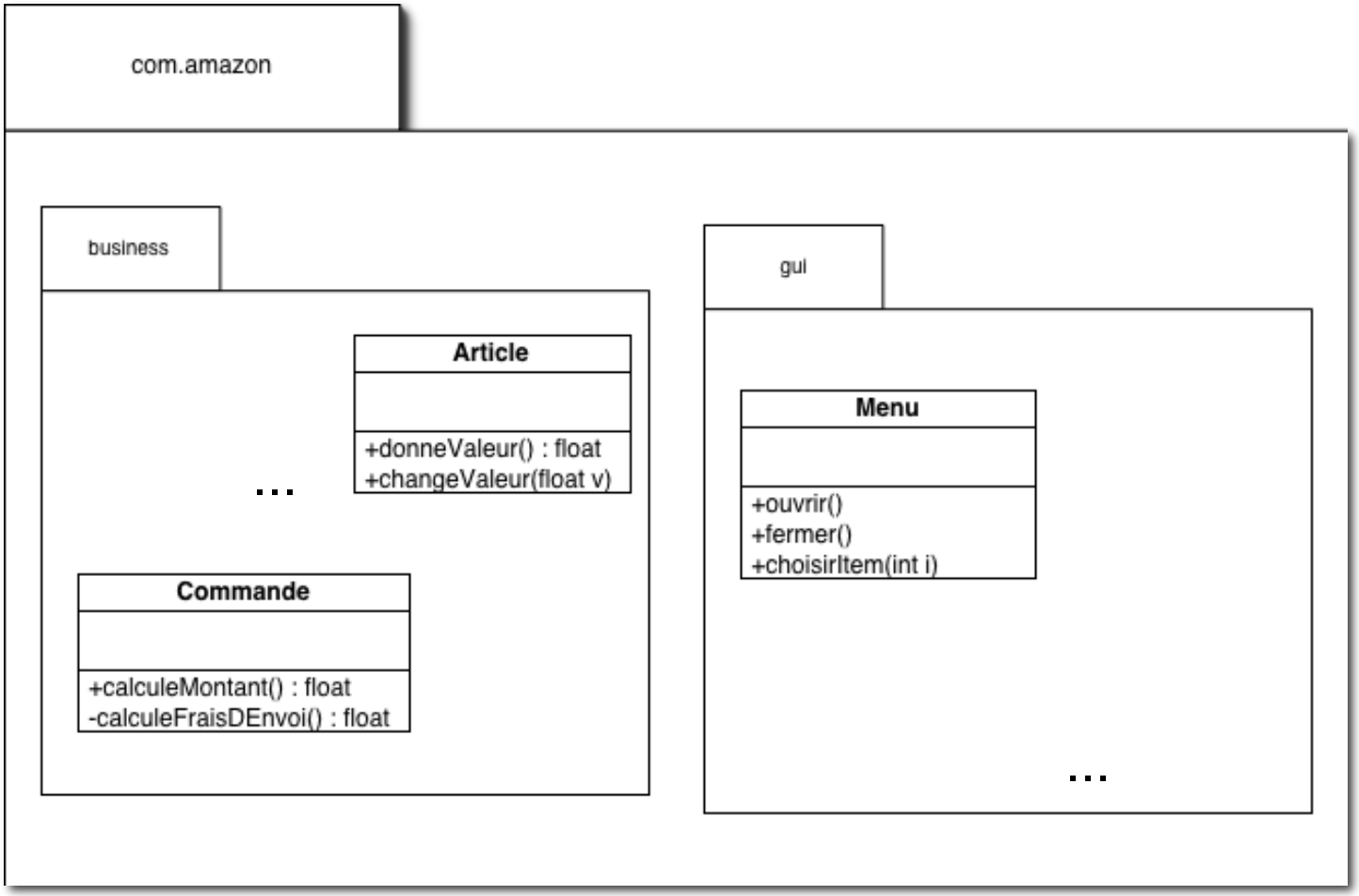
- Quelles sont ces informations ?

Visibilité

- Dans la définition d'une classe, la distinction entre interface et implémentation s'exprime par un marqueur de **visibilité** associé aux éléments, i.e. les méthodes (et *variables membres*) de la classe :
  - un élément `public` (noté +) est rendu accessible à tout le code du programme
  - un élément `private` (noté -) n'est accessible que par le code de la classe elle-même
- NB : Il y a d'autres formes de visibilité plus élaborées (voir plus loin)

Packages et visibilité

- Les classes sont regroupées en **packages** (paquetages)
- Exemple



Packages et visibilité

- Les packages sont tout d'abord un **mécanisme d'encapsulation**
  - les classes, tout comme les méthodes, ont une **visibilité**
  - seules les classes déclarées `public` (notée +) peuvent être utilisées par du code se trouvant dans d'autres packages
  - les classes qui **ne** sont **pas** déclarées `public` sont utilisables uniquement par le code du package (par défaut)

- Syntaxe (simplifiée) des déclarations de classes

```
[package NomPackage;]
[public] class NomClasse {...}
```

Packages et visibilité

Exemple :

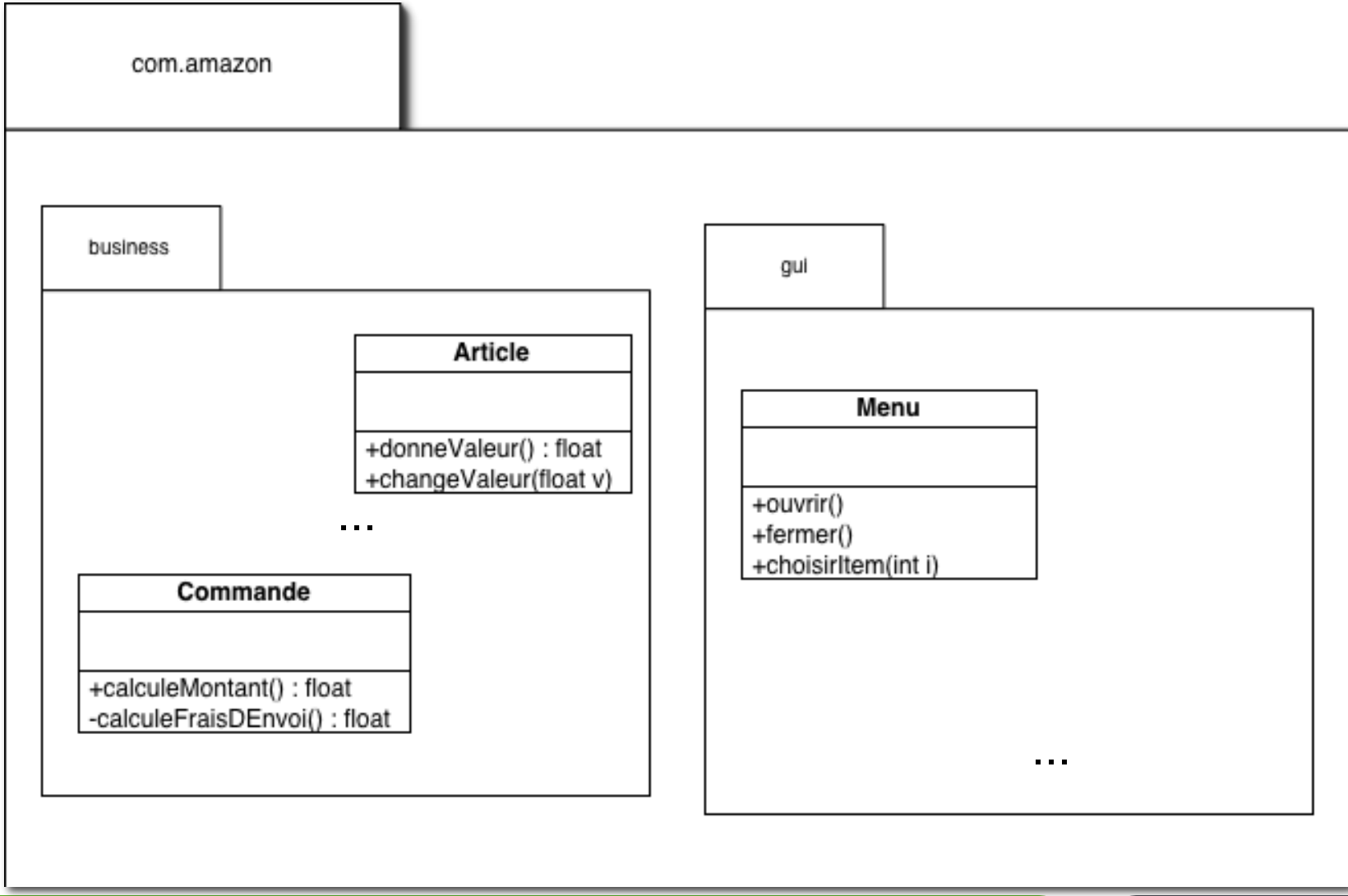
```
package com.amazon.business;

public class Commande {
   ...
}
```

Packages et visibilité

- Conséquences :
  - seuls les éléments déclarés `public` d'une classe déclarée `public` sont accessibles de l'extérieur du package (on verra une exception plus tard avec la visibilité `protected`)
  - nouvelle visibilité : les éléments déclarés **`package`** sont accessibles au code (des autres classes) du package, quelle que soit la visibilité de la classe à laquelle ils appartiennent
  - les éléments déclarés `private` restent confinés à la classe

Packages et visibilité

Exemple :



Packages et visibilité

- Les packages sont également utiles pour le nommage
  - les packages sont organisés hiérarchiquement (≠ hiérarchie de classes!)
  - chacun possède un **nom qualifié** (fully qualified name) qui reprend toute la hiérarchie du package et l'identifie de manière absolue
    - Ex: `com.amazon.business, com.amazon.gui,…`
  - un même package ne peut donc contenir deux sous-packages (directs) de même nom
  - un même package ne peut pas non plus contenir directement deux classes de même nom

Packages et visibilité

- Toute classe possède un **nom court** ou nom **relatif** (p/r au package)
  - Exemple : `Commande`
- Toute classe possède aussi un **nom qualifié** (fully qualified name) ou nom **absolu** : `nomQualPackage.nomClasse`
  - Exemple : `com.amazon.business.Commande`
- Si du code doit accéder à une classe du même package, il peut se contenter du nom court
- Si du code doit accéder à une classe d'un autre package, il doit utiliser:
  - soit le nom qualifié,
  - soit **importer** la classe ou le package dans laquelle elle se trouve

Packages et visibilité

```
package com.amazon.gui;

class Menu {
   ...
   void choisirItem(int i) {
        ...
        com.amazon.business.Commande maCom = ...;
        float total = maCom.calculeMontant();
   }
   ...
}
```

Packages et visibilité

```
package com.amazon.gui;
import com.amazon.business.Commande;

class Menu {
   ...
   void choisirItem(int i) {
        ...
        Commande maCom = ...;
        float total = maCom.calculeMontant();
   }
   ...
}
```

Packages et visibilité

```
package com.amazon.gui;
import com.amazon.business.*;

class Menu {
   ...
   void choisirItem(int i) {
        ...
        Commande maCom = ...;
        float total = maCom.calculeMontant();
            ...
            Article monArticle = ...;
        float val = monArticle.donneValeur();
   }
   ...
}
```

Packages et visibilité

```
package com.amazon.gui;
import com.amazon.business.Commande;

class Menu {
   ...
   void choisirItem(int i) {
    ...
    Commande maCom = ...;
    float total = maCom.calculeMontant();
   }
   ...
}
```

possibilité de **conflit de noms** si `com.amazon.gui` possède également une classe nommée `Commande`

Packages et visibilité

```
package com.amazon.gui;
import com.amazon.business.Commande;
import com.amazon.db;

class Menu {
   ...
   void choisirItem(int i) {
        ...
        Commande maCom = ...;
        float total = maCom.calculeMontant();
   }
   ...
}
```

possibilité de **conflit de noms** si on importe une autre classe de même nom
Ex: `Menu` importe `com.amazon.db` qui possède également une classe nommée `Commande`

Packages et visibilité

- Syntaxe (révisée mais toujours simplifiée) des déclarations de classes :

```
[package NomPackage;]
[import NomPackage [.NomPackage]* [.NomClasse]; ]*
[public] class NomClasse {...}
```

Objets et variables

- Les données du programme sont stockées dans des **variables**
- Toute variable possède un **type**
- Un type est soit
  - un **type primitif**
    - ex : `int, boolean, char`,…
  - un **type d'objets**
    - ex : `String, MultiSet, Commande, int[],...`

Types primitifs

- Un type primitif définit un ensemble de **valeurs**
  - ex : `int` définit un ensemble de valeurs telles que `0`, `12`, `544226`,...
- En Java, il existe 8 types primitifs. Ils sont **prédéfinis** dans le langage.
- L'utilisateur ne peut pas en définir de nouveaux.

Types primitifs

Type	Contains	Default	Size	Range
`boolean`	`true` or `false`	`false`	1 bit	NA
`char`	Unicode character	`\u0000`	16 bits	`\u0000` to `\uFFFF`
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

Types d'objets

- Un type d'objet définit un ensemble d'**objets**
  - ex : Le type `Commande` définit l'ensemble des commandes qu'Amazon.com est susceptible de traiter
  - ex : Le type `MultiSet` définit l'ensemble des multi-ensembles d'entiers
- Le programmeur peut définir autant de nouveaux types d'objets qu'il le désire
  - c'est en cela que Java et les autres langages OO facilitent l'abstraction par les données !
- Divers types d'objets sont néanmoins fournis au programmeur dans des packages définis par d'autres
  - ex: le package `java.lang` fournit le type `String`
  - NB : contrairement à la règle, `java.lang` ne doit pas être importé pour que l'on puisse en utiliser les types via leurs noms courts

Variables primitives

- Une **variable d'un type primitif** (aka *variable de base* aka *variable primitive*) contient une **valeur de ce type**
  - ex: `boolean laVieEstBelle = true;`
  - ex: `int i = 6 + 9;`
    - `6 + 9` est une expression dont l'évaluation donne `15`
      i.e. une valeur de type `int`

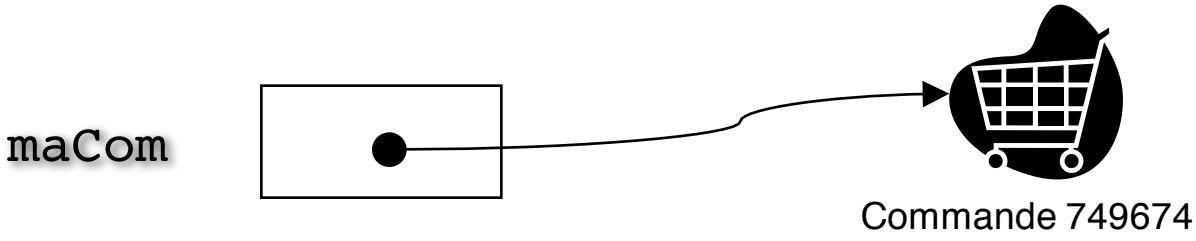
- Règles d'initialisation :
  - Toute variable **peut** être initialisée au moment de sa déclaration
  - Elle **doit** l'être avant toute utilisation
    - ex : `int j; int i = j + 2;` génère une erreur de compilation

Variables de référence

- Une **variable de type objet** (= toute variable non primitive) contient une **référence vers un objet de ce type**
  - rappel: les `String` et les tableaux sont des types d'objets !

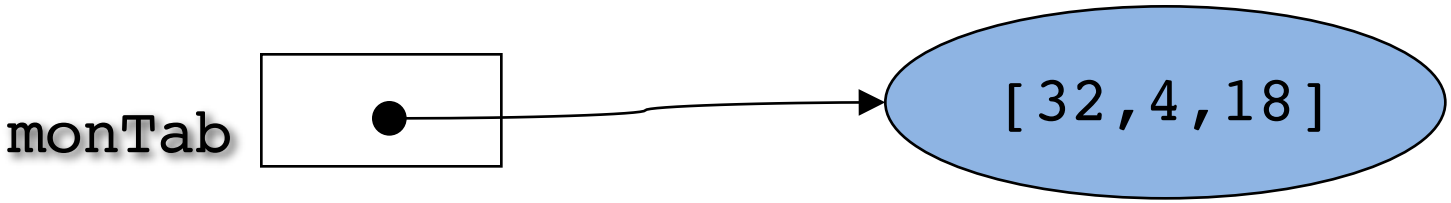
- Une telle variable est également appelée **variable de référence** ou, dans certains langages, *pointeur*

- Ex : `Commande maCom = <expr>` où `<expr>` est une expression dont l'évaluation fournit une référence vers une Commande



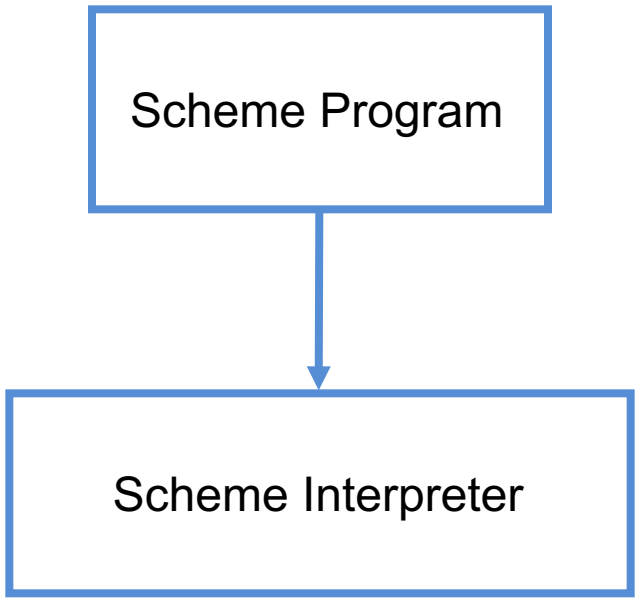
Variables de référence

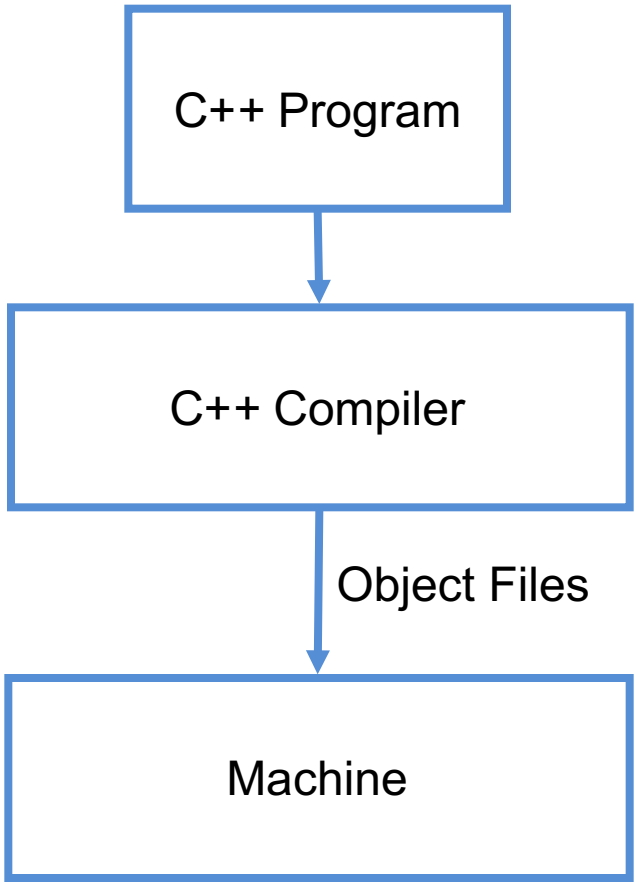
- Ex : `int[] monTab = <expr>` où <expr> est une expression dont l'évaluation fournit une référence vers un tableau d'entiers



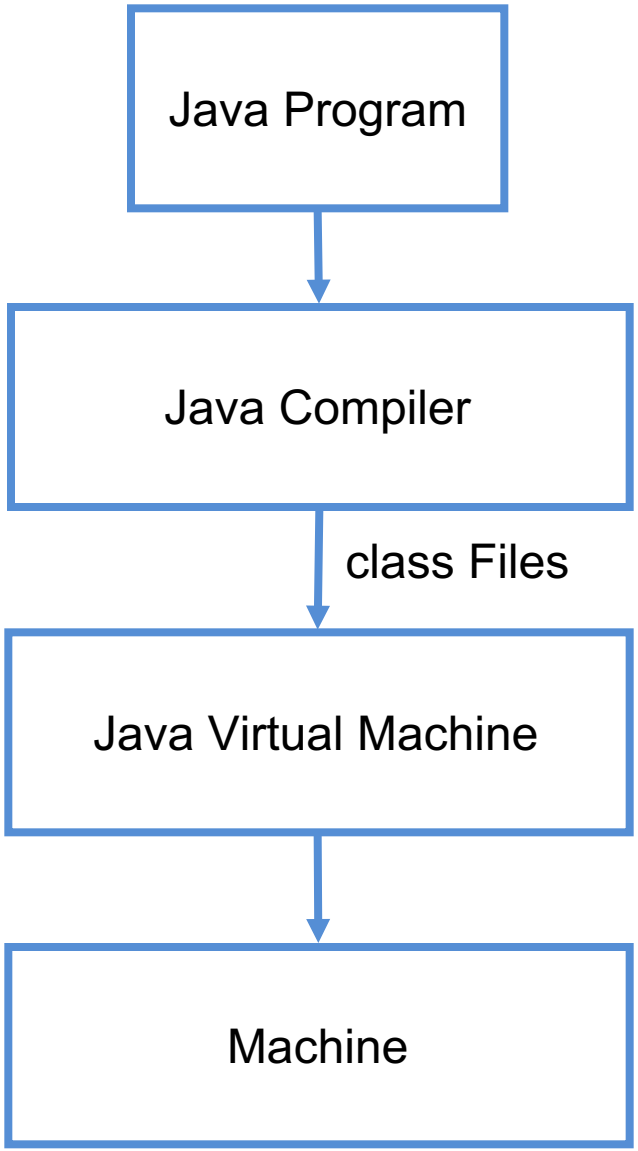
- Donc, en Java, **toute variable non primitive est un pointeur** !
- Les règles d'initialisation s'appliquent aussi aux variables de référence
- Une variable de référence qui ne pointe vers aucun objet a la valeur spéciale `null` :

Systèmes de programmation





Java VM



- Portabilité
  - si une implémentation de la Java VM existe pour l'architecture de votre machine, vous pouvez alors exécuter tous les programmes Java
- Securité
  - une VM peut restreindre l'accès du programme à la machine réelle
- Simplicité
  - les instructions de la VM instructions peuvent être plus simple que les instructions machines

Exécution d'un programme Java

- L'exécution d'un programme Java est une séquence d'exécutions de méthodes (en commençant par l'appel à la procédure `public static void main(String[] args)`)
- Au début de l'exécution d'une méthode, un **activation record** est **empilé** sur la **pile** (**stack**)
  - une entrée pour chaque paramètre de méthode
  - une entrée pour chaque variable locale
- L'activation record est **dépilé** de la pile quand la méthode se termine
- Vu que les appels de méthode peuvent être imbriqués, ce processus est **récursif**

Stack et heap

- La machine virtuelle Java possède deux mémoires :
  - le **stack** (pile)
  - le **heap** (tas)
- Les **variables locales** (typiquement, celles qui sont déclarées à l'intérieur des méthodes) **résident sur le stack**, **qu'elles contiennent une valeur ou une référence**
- Les **objets résident dans le heap**

Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```

Stack et heap

```
public static void doThing()
{
    int i = 6;
    int j;
    int[] a = {1,3,5,7,9};
    int[] b = new int[3];
    String s = "abcdef";
    String t = null;
    …
}
```

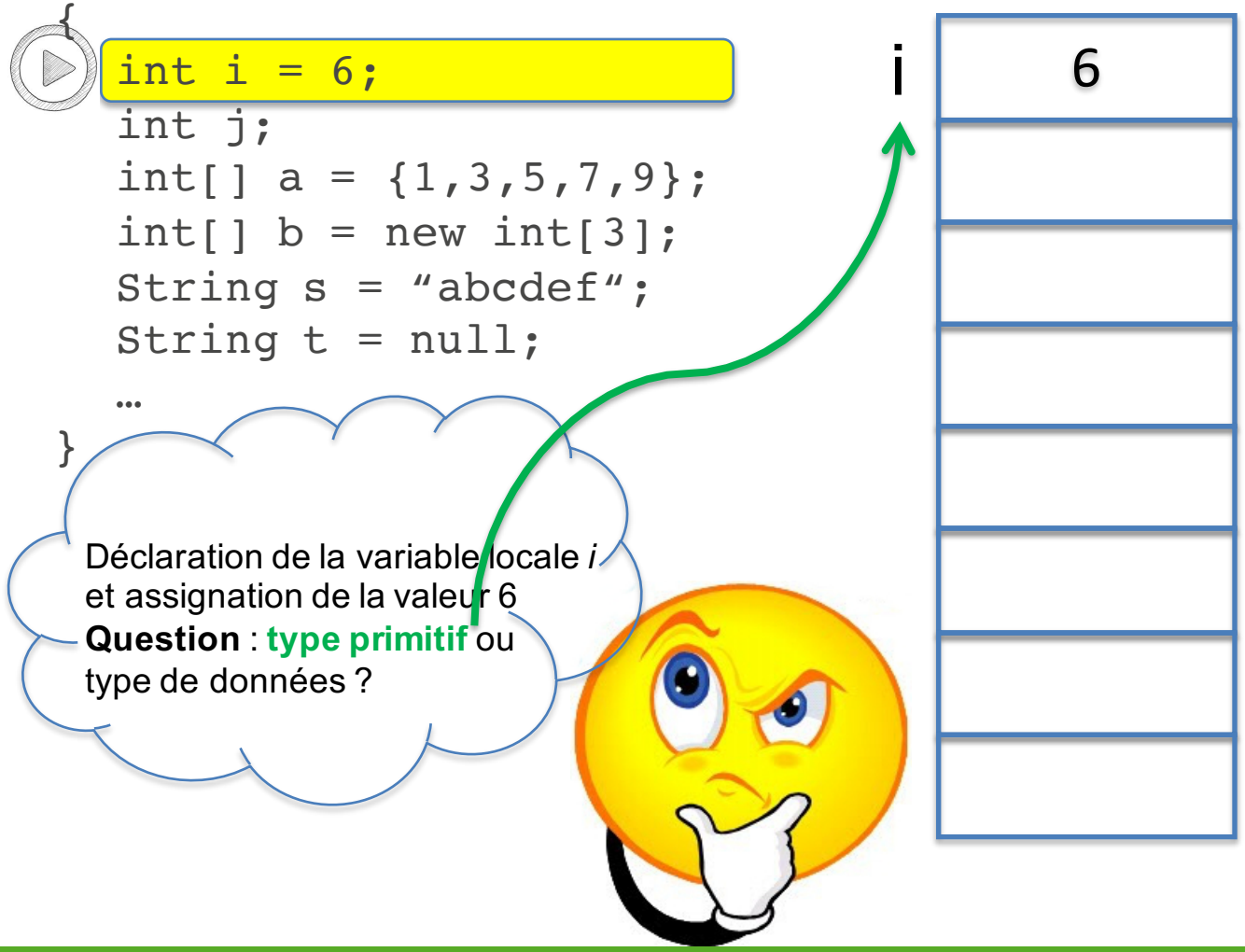
Déclaration de la variable locale *i* et assignation de la valeur 6
Question : type primitif ou type de données ?

stack

heap

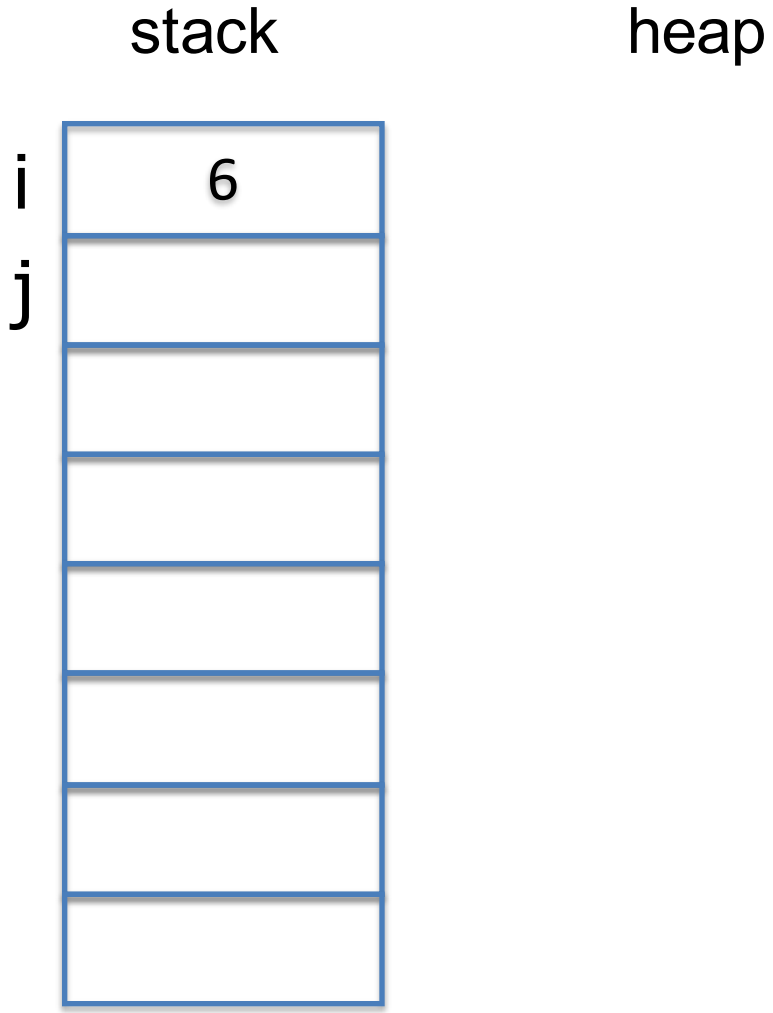
Stack et heap

```
public static void doThing()
{
    int i = 6;
    int j;
    int[] a = {1,3,5,7,9};
    int[] b = new int[3];
    String s = "abcdef";
    String t = null;
    …
}
```



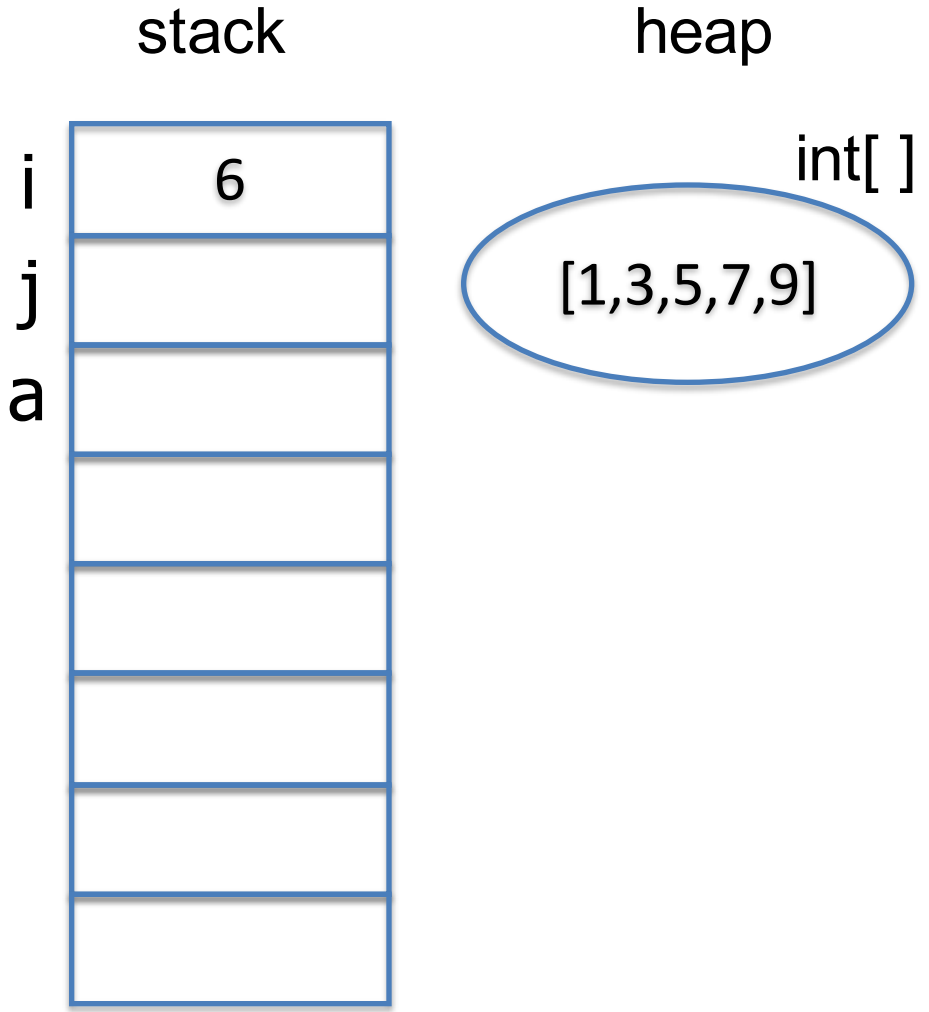
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



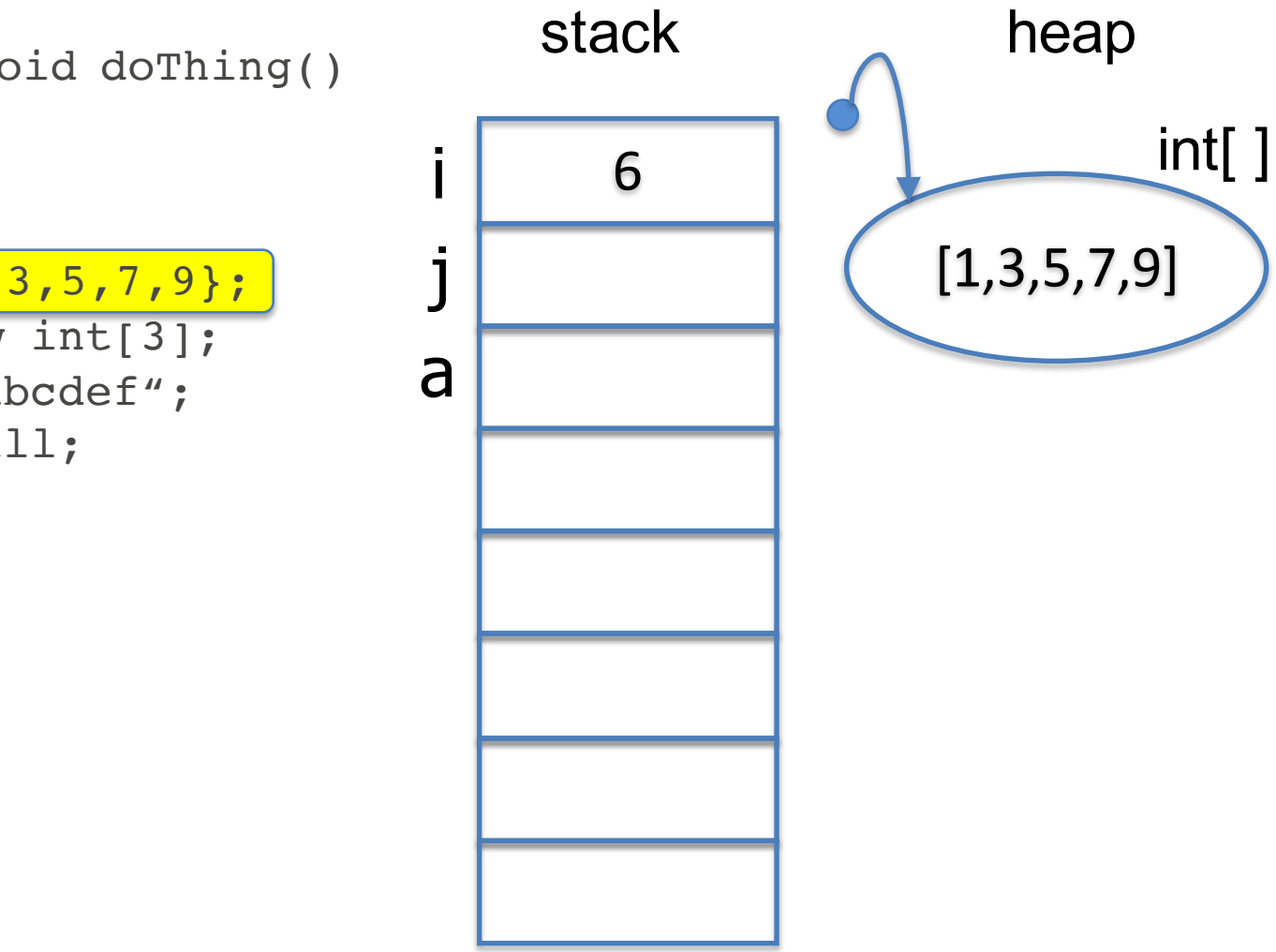
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



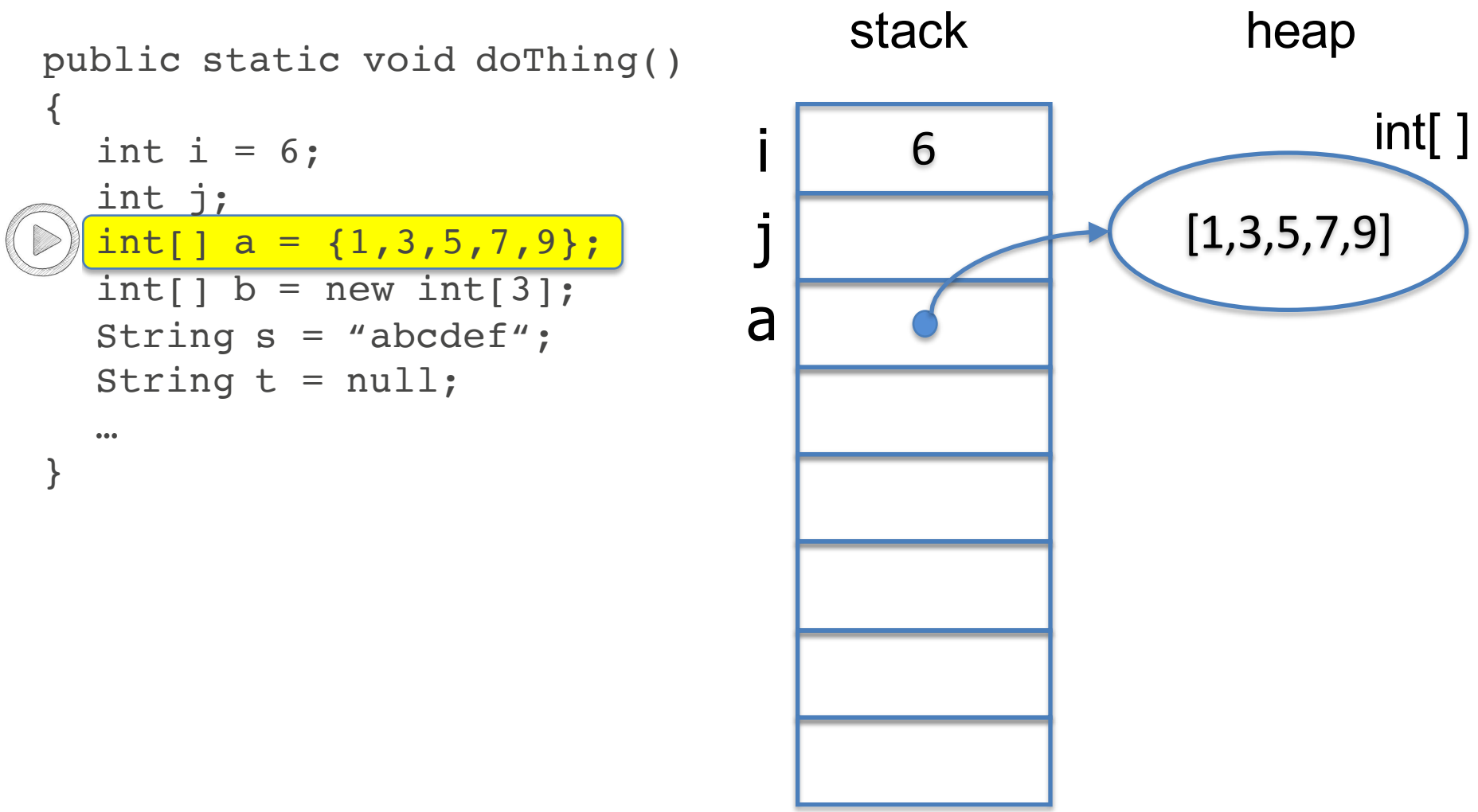
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



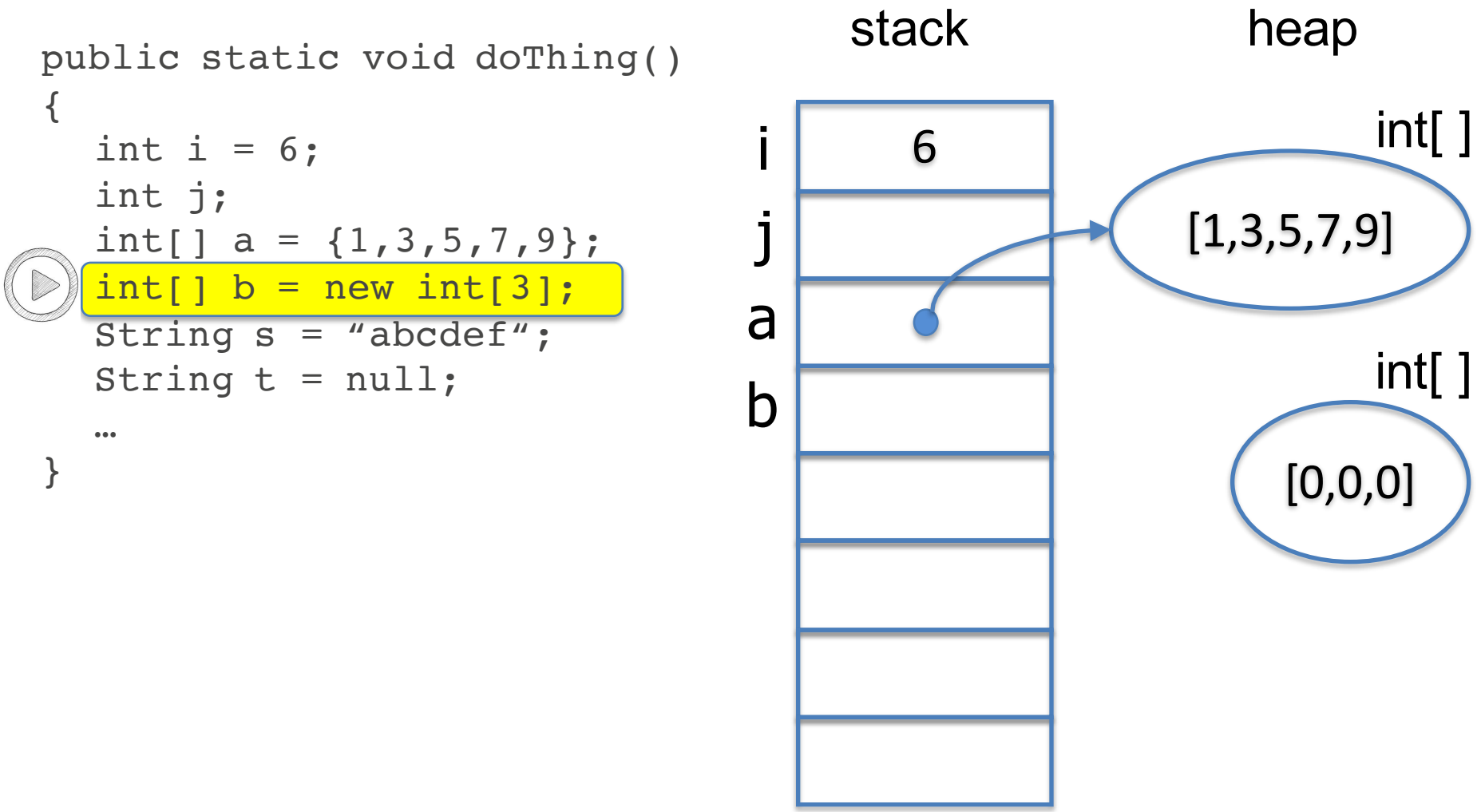
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



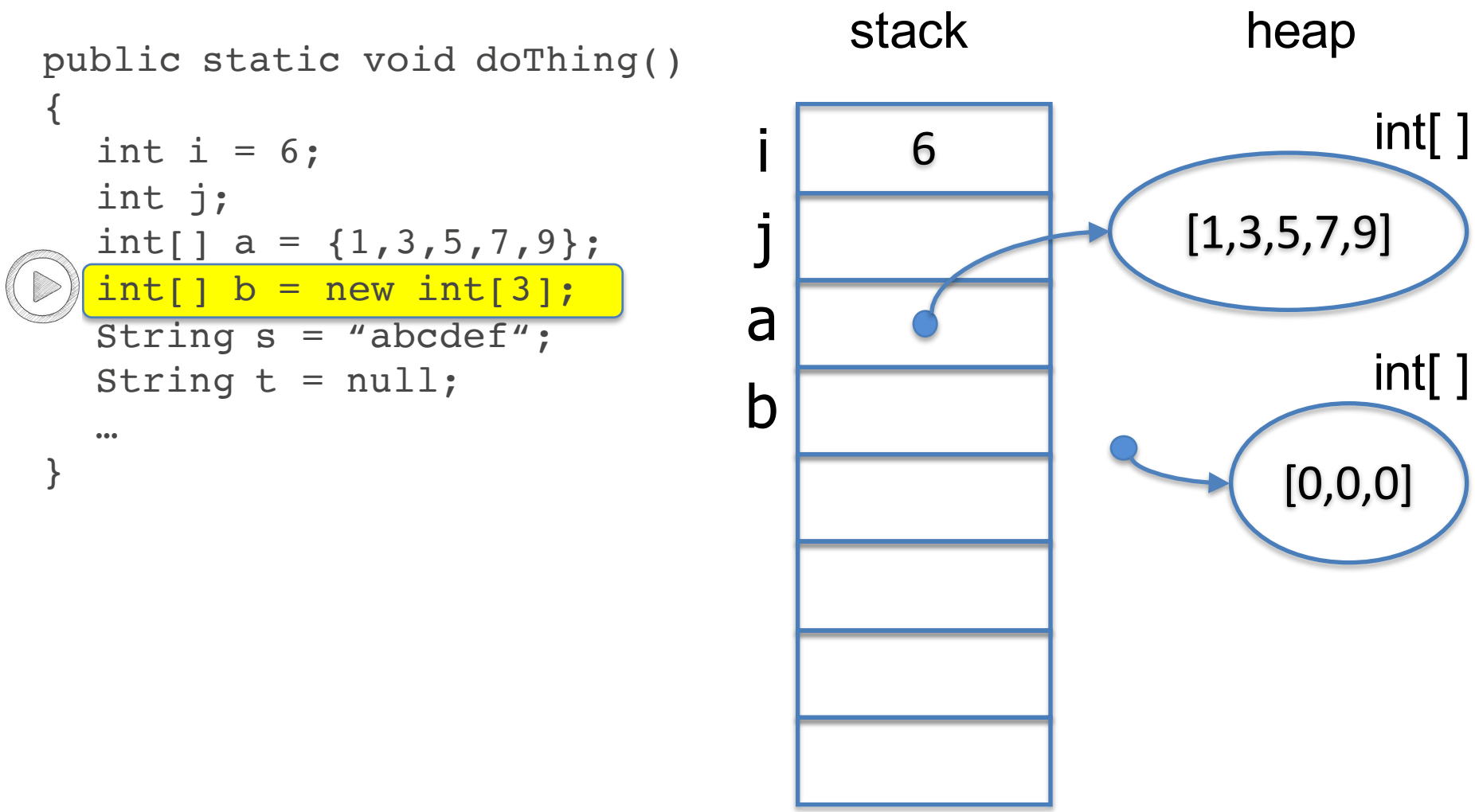
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



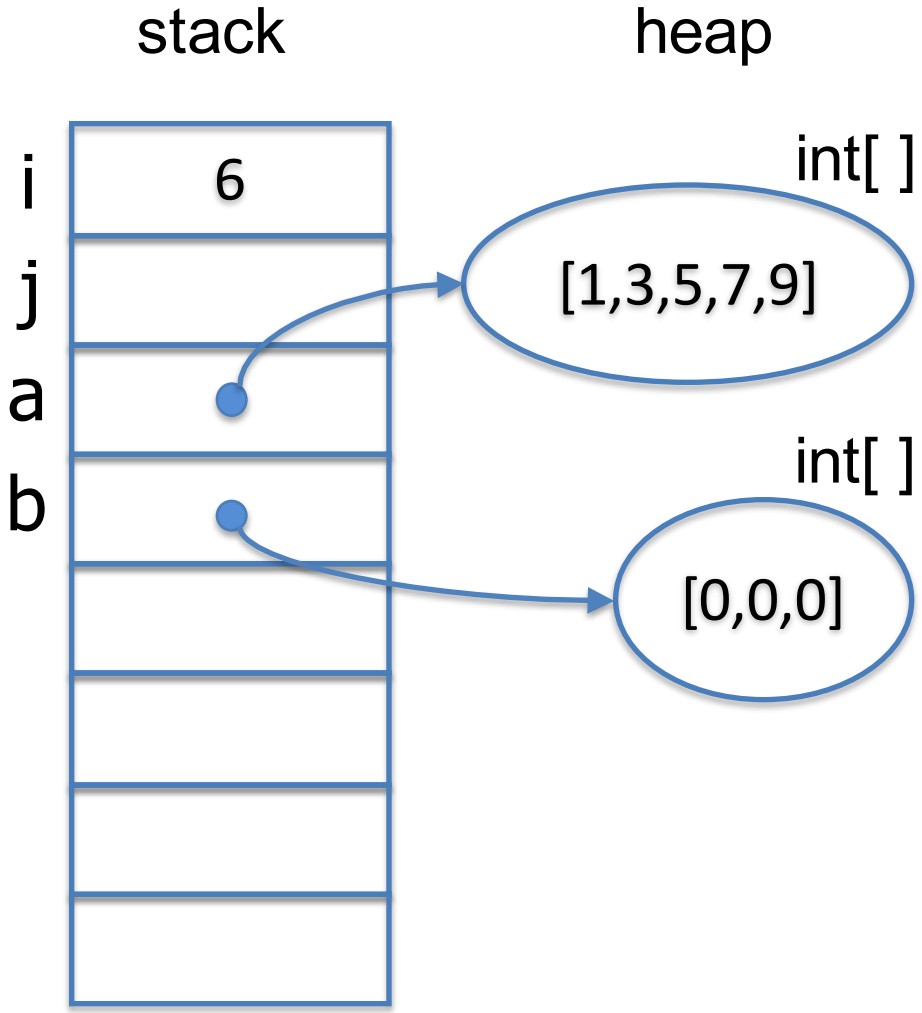
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



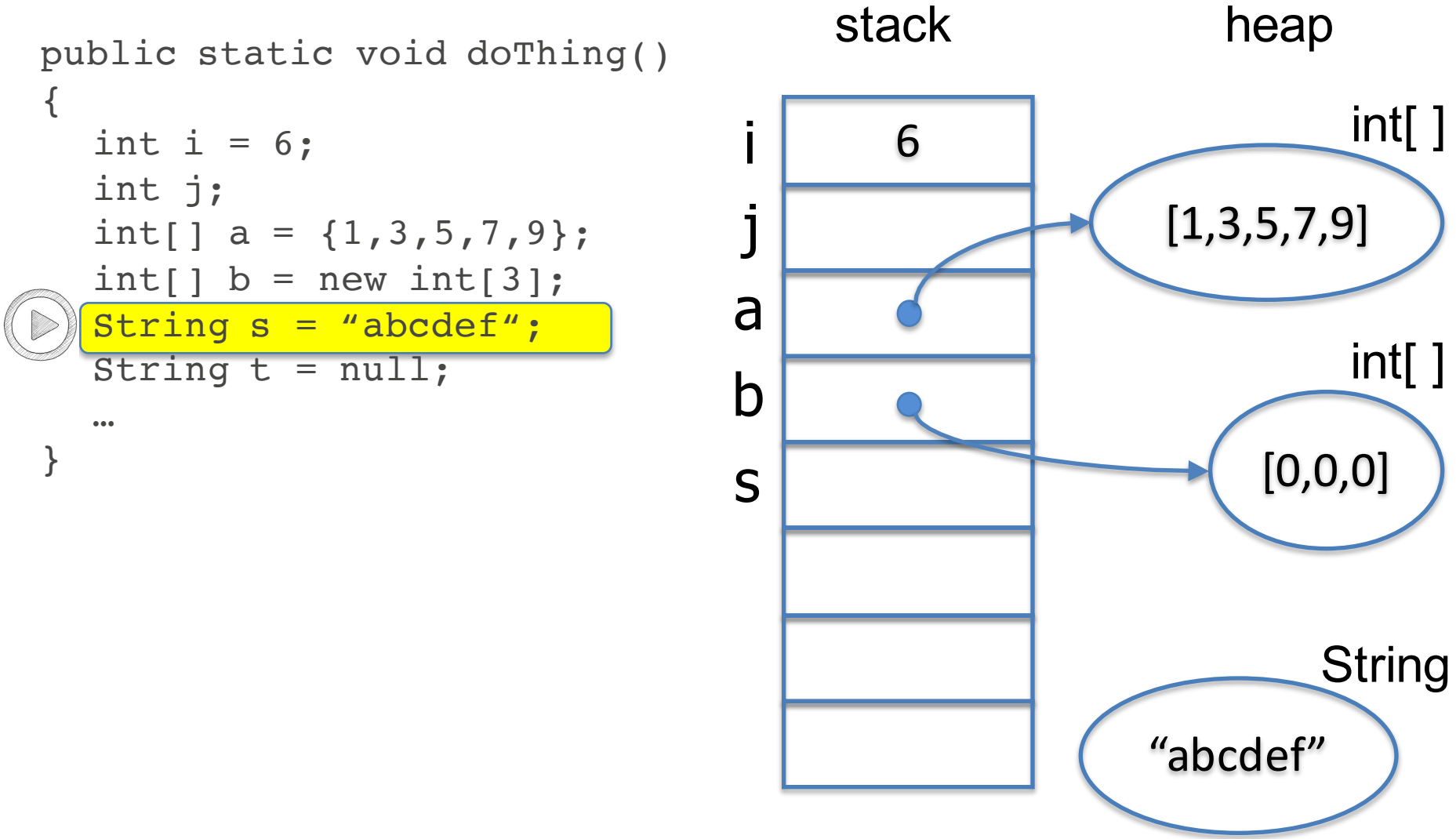
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



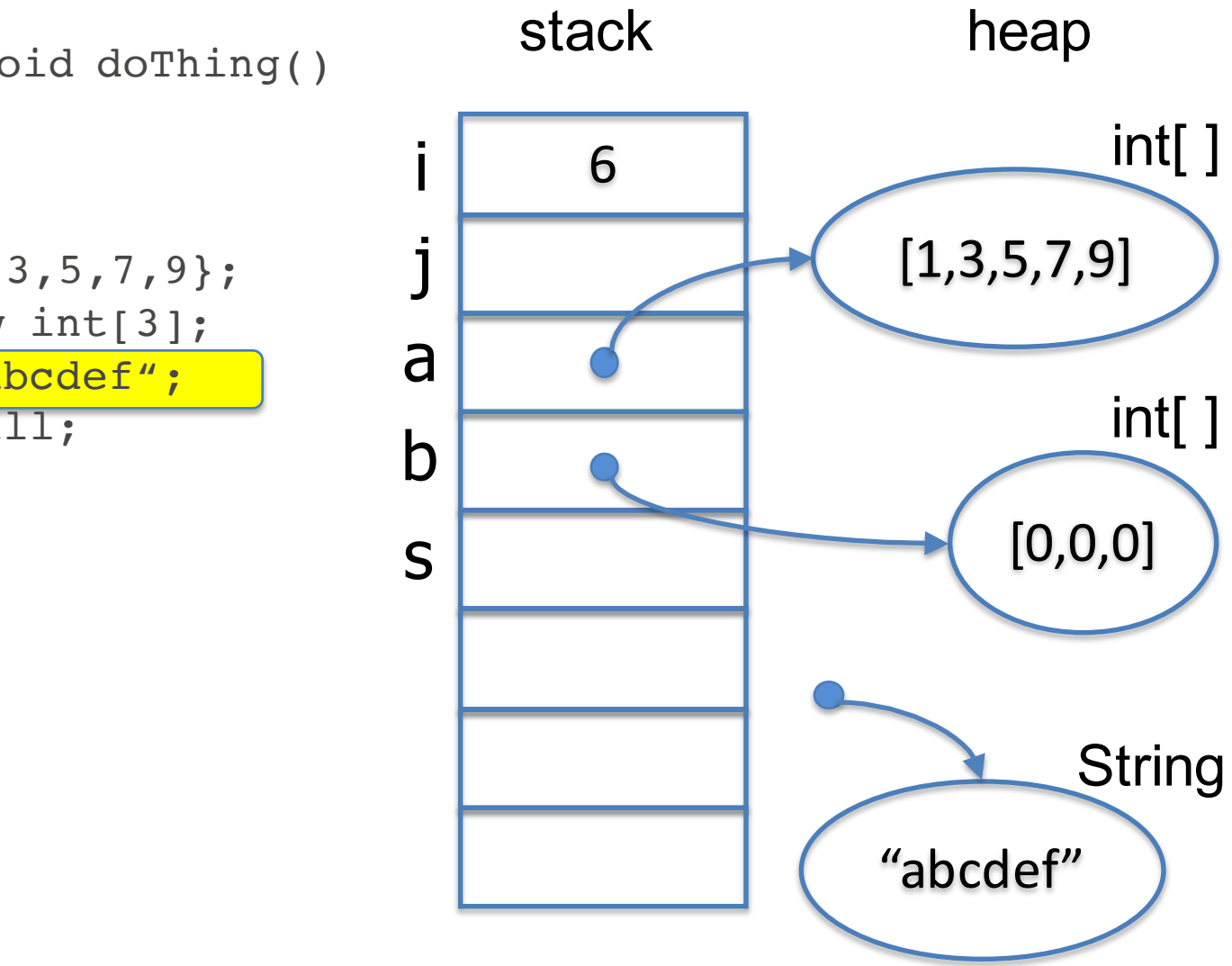
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = “abcdef“;
   String t = null;
   …
}
```



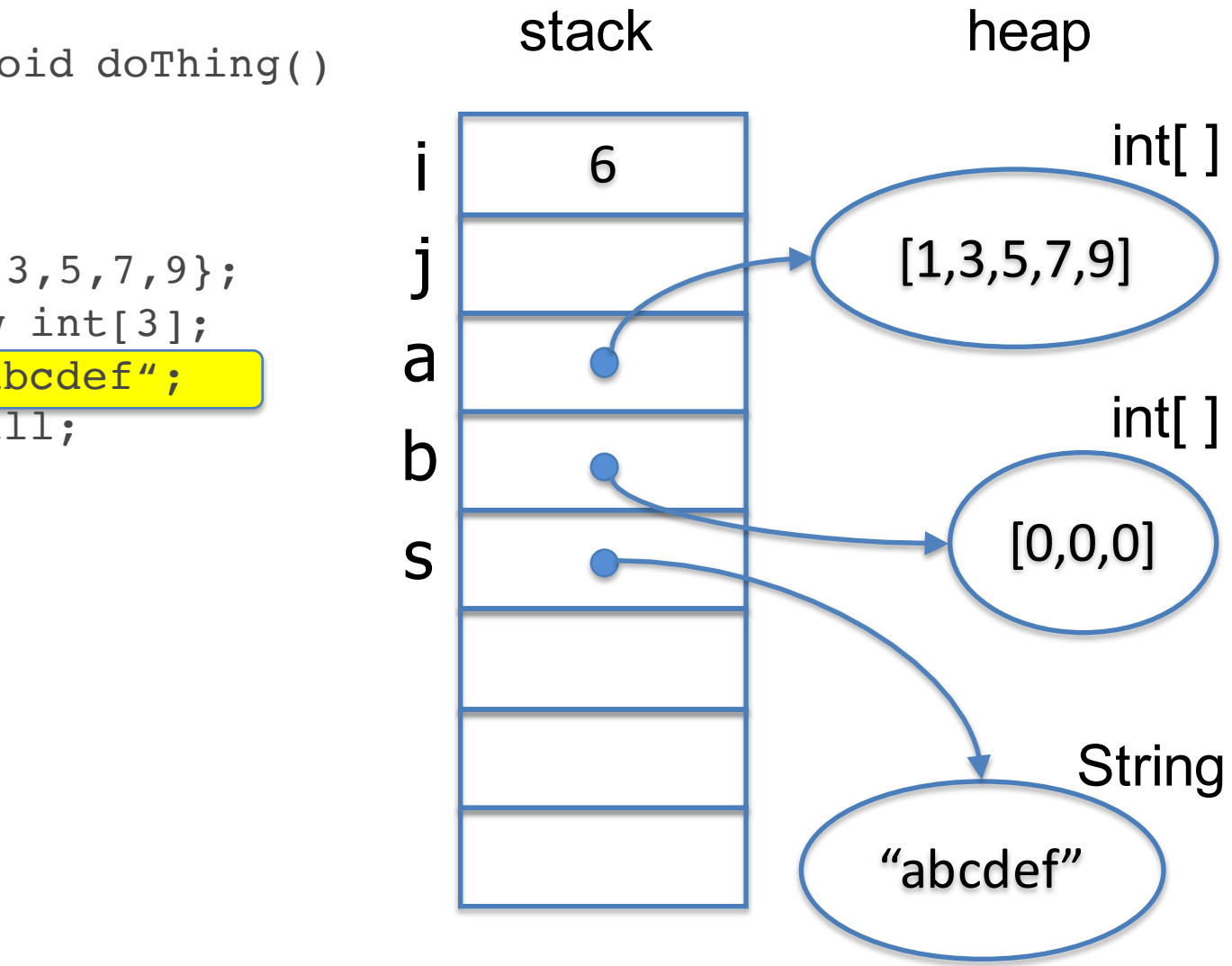
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = "abcdef";
   String t = null;
   …
}
```



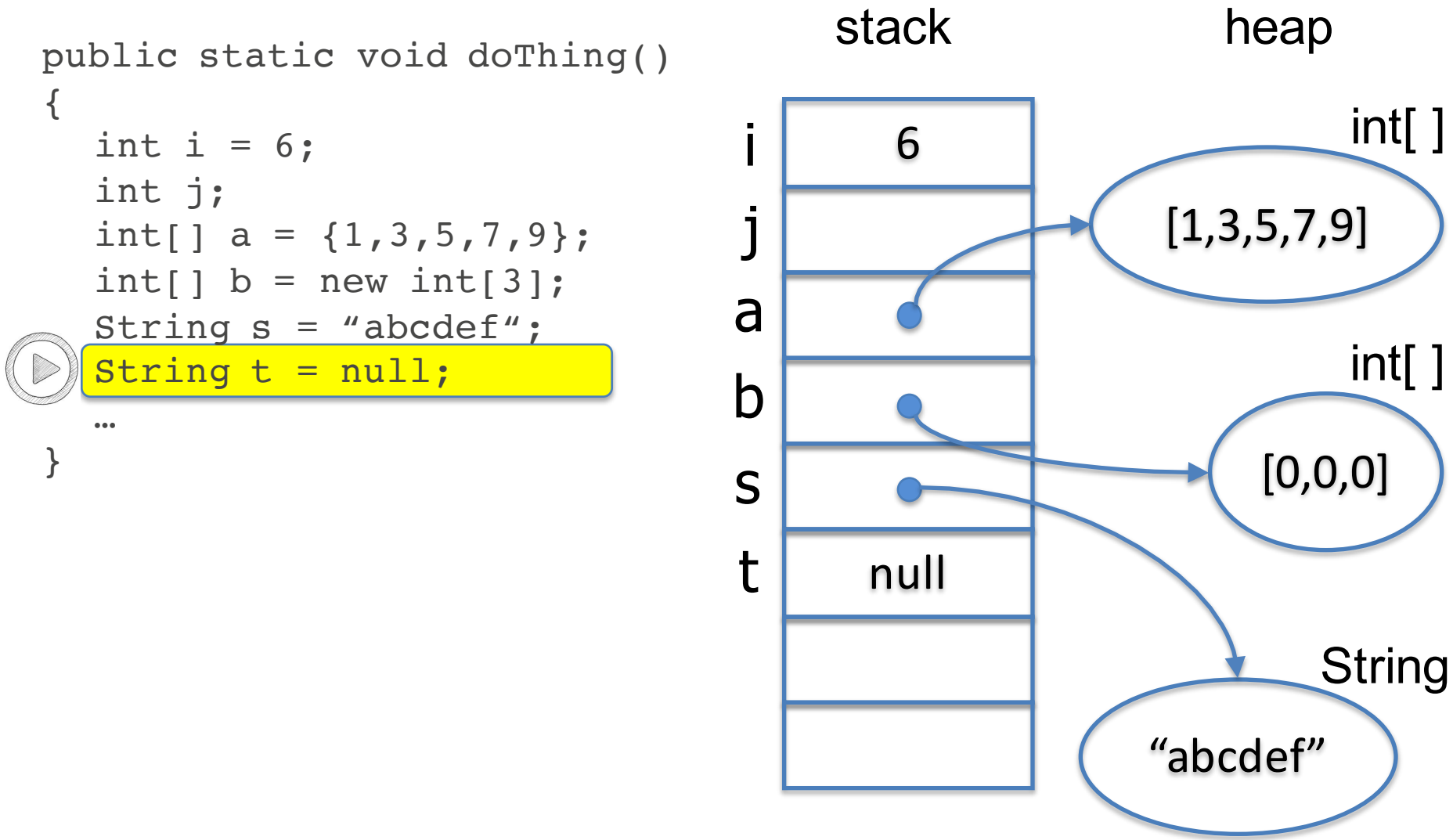
Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = "abcdef";
   String t = null;
   …
}
```



Stack et heap

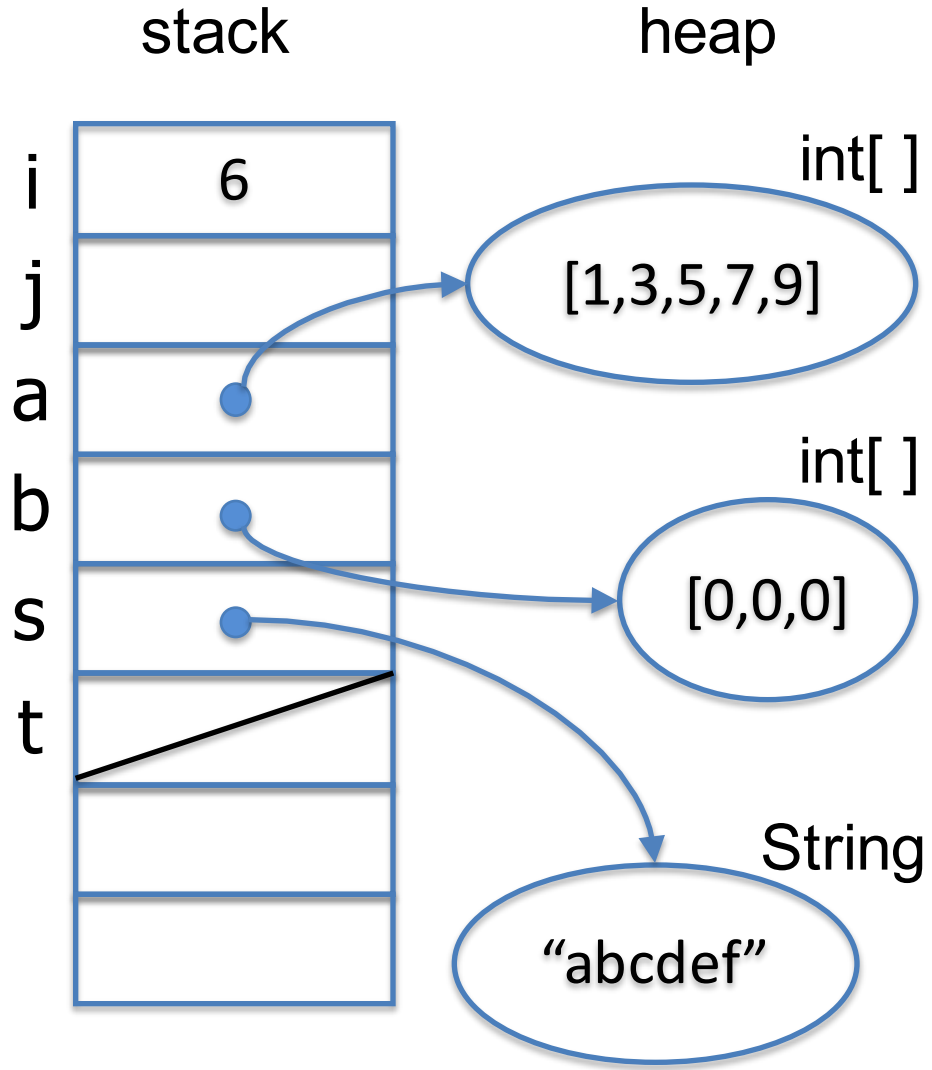
```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = "abcdef";
   String t = null;
   …
}
```



Stack et heap

```
public static void doThing()
{
   int i = 6;
   int j;
   int[] a = {1,3,5,7,9};
   int[] b = new int[3];
   String s = "abcdef";
   String t = null;
   …
}
```

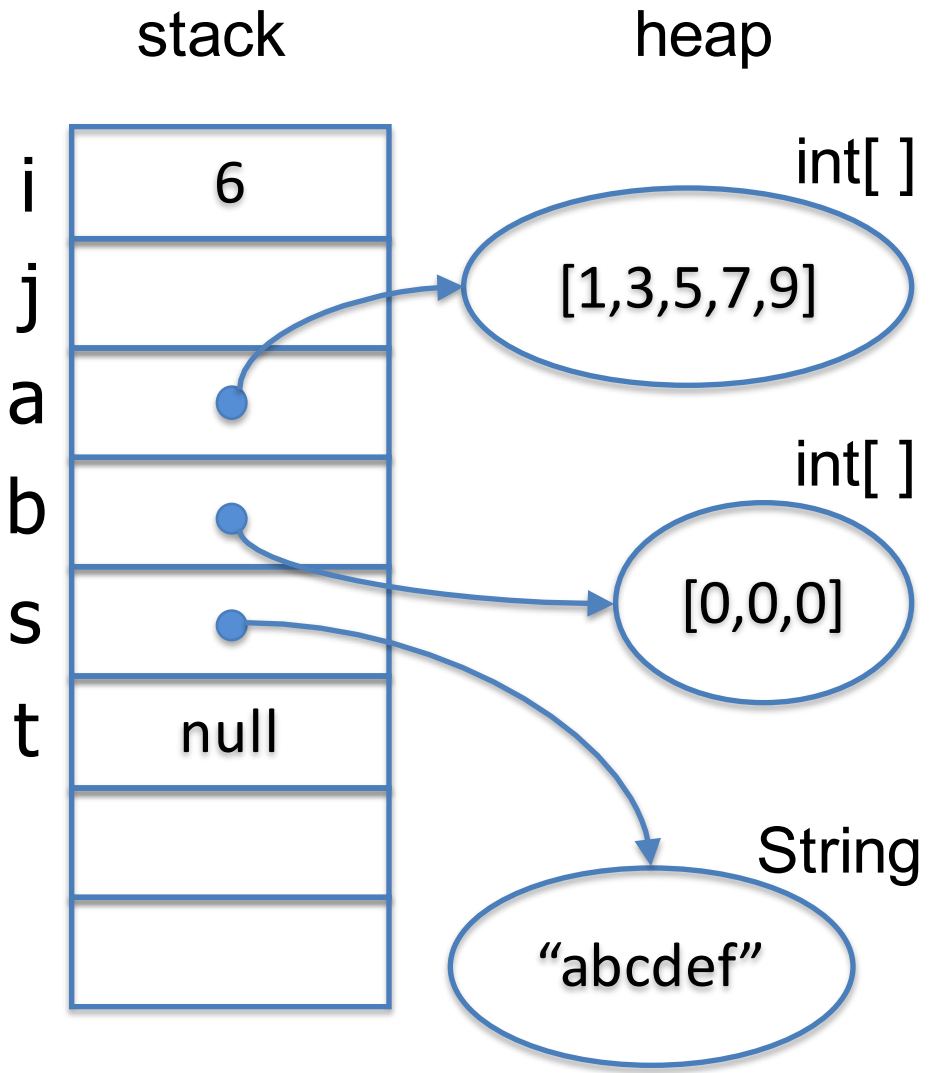
ALTERNATIVE



Stack et heap

```
public static void doThing()
{
    int i = 6;
    int j;
    int[] a = {1,3,5,7,9};
    int[] b = new int[3];
    String s = "abcdef";
    String t = null;
    …
}
```

Après l'exécution de ces instructions, le stack et le heap contiennent :



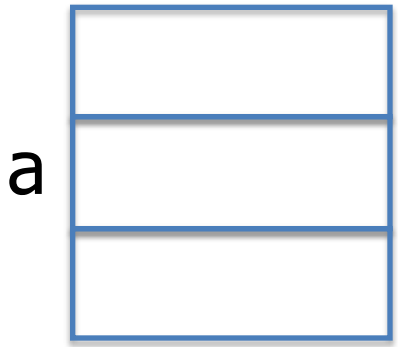
Création d'objets

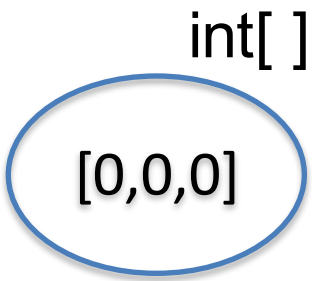
- La façon standard de créer un nouvel objet dans le heap est d'utiliser l'instruction **`new`**
  - `ex : int[] a = new int[3];`
- `new` a pour effet
  - d'**allouer une partie du heap** au stockage d'un objet du type indiqué
    - un tableau d'entiers de longueur 3 dans l'exemple

Création d'objets

- `new` a pour effet (suite)
  - d'**initialiser l'objet** (pas la variable de référence!) en appelant une *méthode particulière* du type appelée **constructeur** (cf. Create de CRUD)
    - dans l'exemple, le constructeur initialise le contenu du tableau à `[0,0,0]` (contenu par défaut)

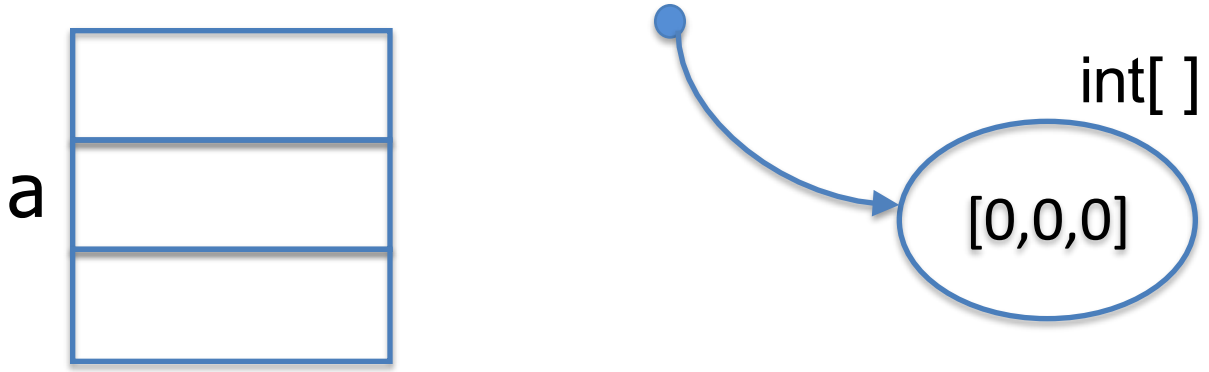
- NB: `new` peut également être utilisé avec des paramètres pour initialiser le contenu de l'objet à d'autres valeurs que celles par défaut





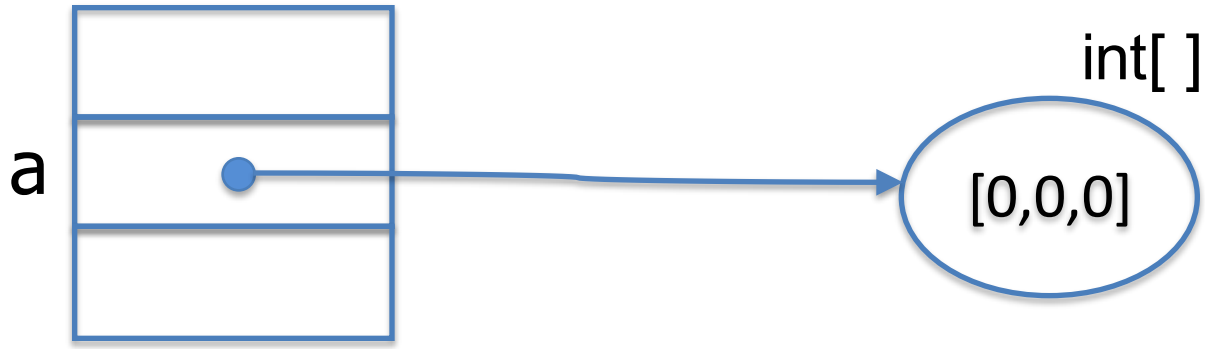
Création d'objets

- `new` a pour effet (suite)
  - d'assigner une (**valeur de) référence** à l'objet nouvellement créé, qui le distingue de tous les autres objets présents dans le heap. On parle aussi d'*identifiant* de l'objet (*object ID*).



Création d'objets

- `new` a pour effet (suite et fin)
  - de **retourner cette valeur de référence**
    - dans l'exemple, cette valeur est *ensuite* assignée à la variable `a` via le symbole `=`



Création d'objets

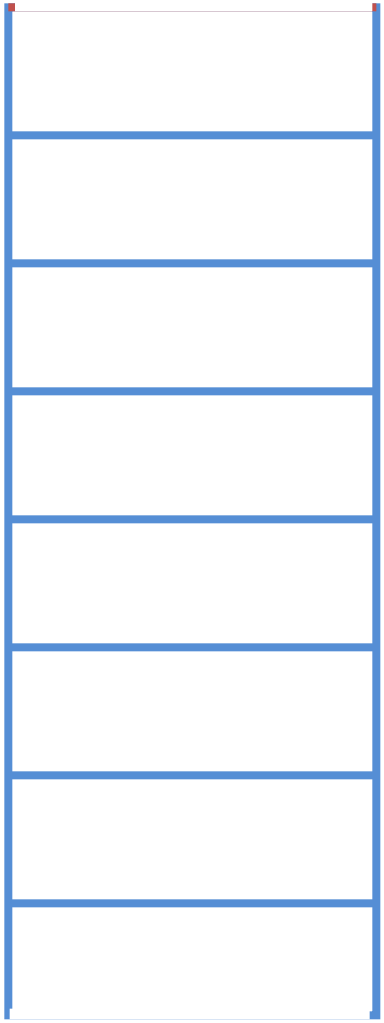
- Il existe 2 formes particulières de création d'objets
  - pour les tableaux, par ex :
    - `int[] a = {1,3,5,7,9};`
  - pour les `String`, par ex :
    - `String s = "abcdef";`
- Ces formes particulières sont des raccourcis syntaxiques définis dans le langage
- Ils ont (presqu') exactement le même effet qu'un `new`
  - mais un `new` paramétré i.e. qui initialiserait le contenu de l'objet à l'aide des valeurs indiquées et non avec des valeurs par défaut

Partage de références

- Si une variable de référence se voit donner une valeur de référence déjà assignée à une autre variable, **les deux variables partagent (la référence vers) l'objet**
- L'objet devient alors accessible par ces deux variables
- Partage de références = **aliasing** (en anglais)

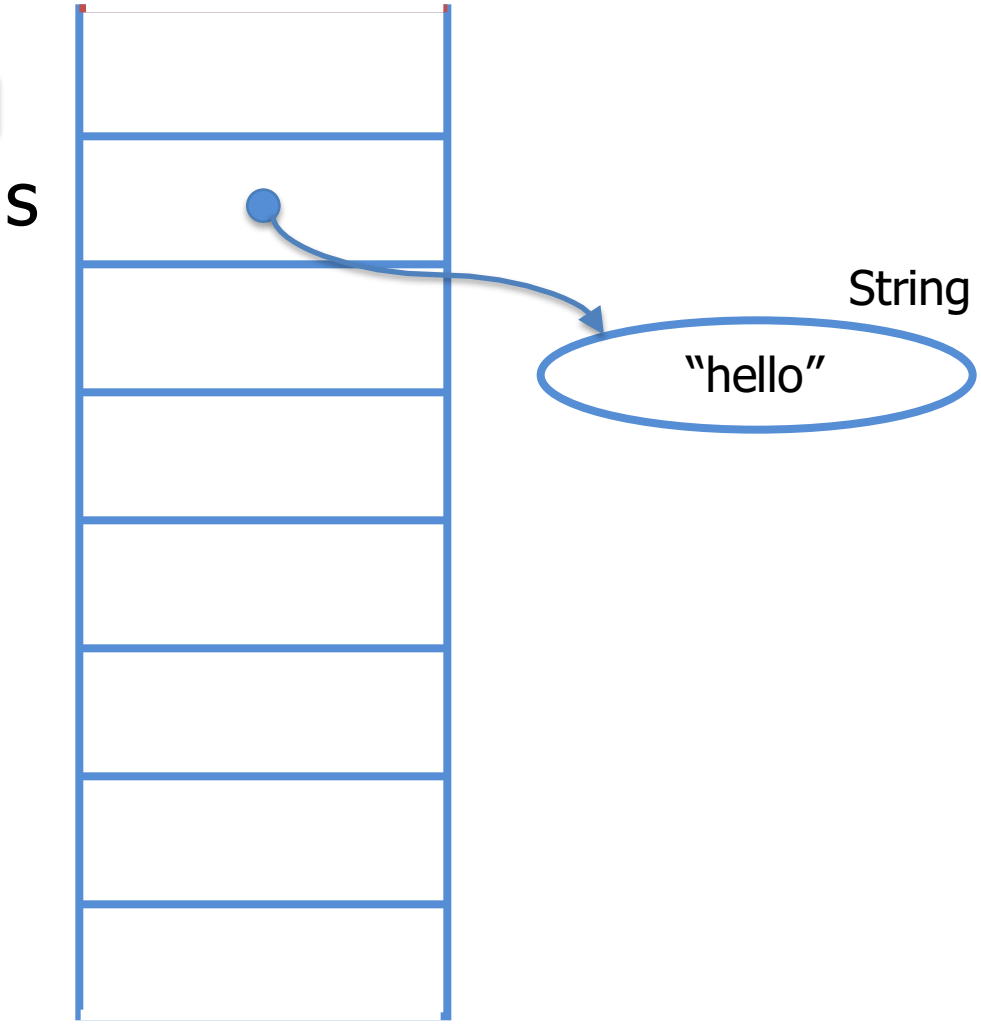
Partage de références

```
public static void doThing() {
    …
    String s = new String (“hello”);
    String t = s;
    …
}
```



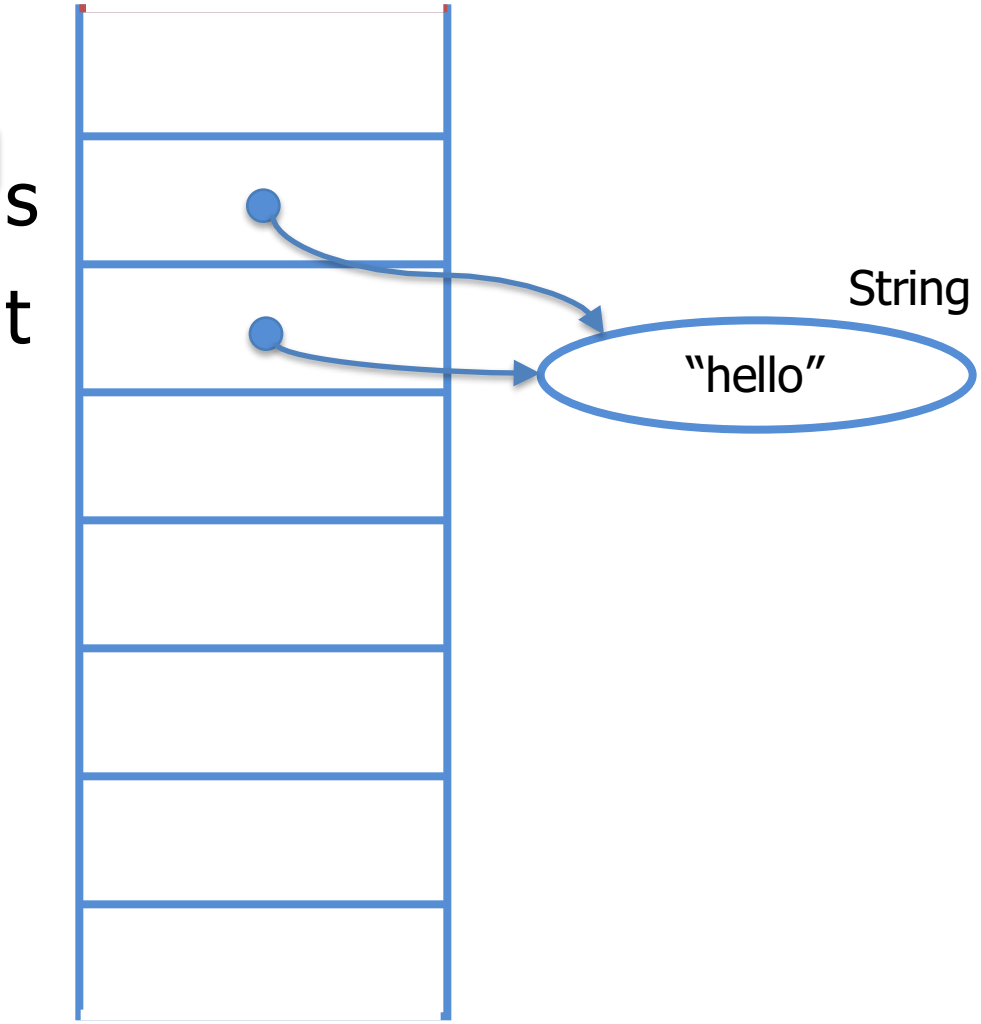
Partage de références

```
public static void doThing() {
    …
    String s = new String (“hello”);
    String t = s;
    …
}
```



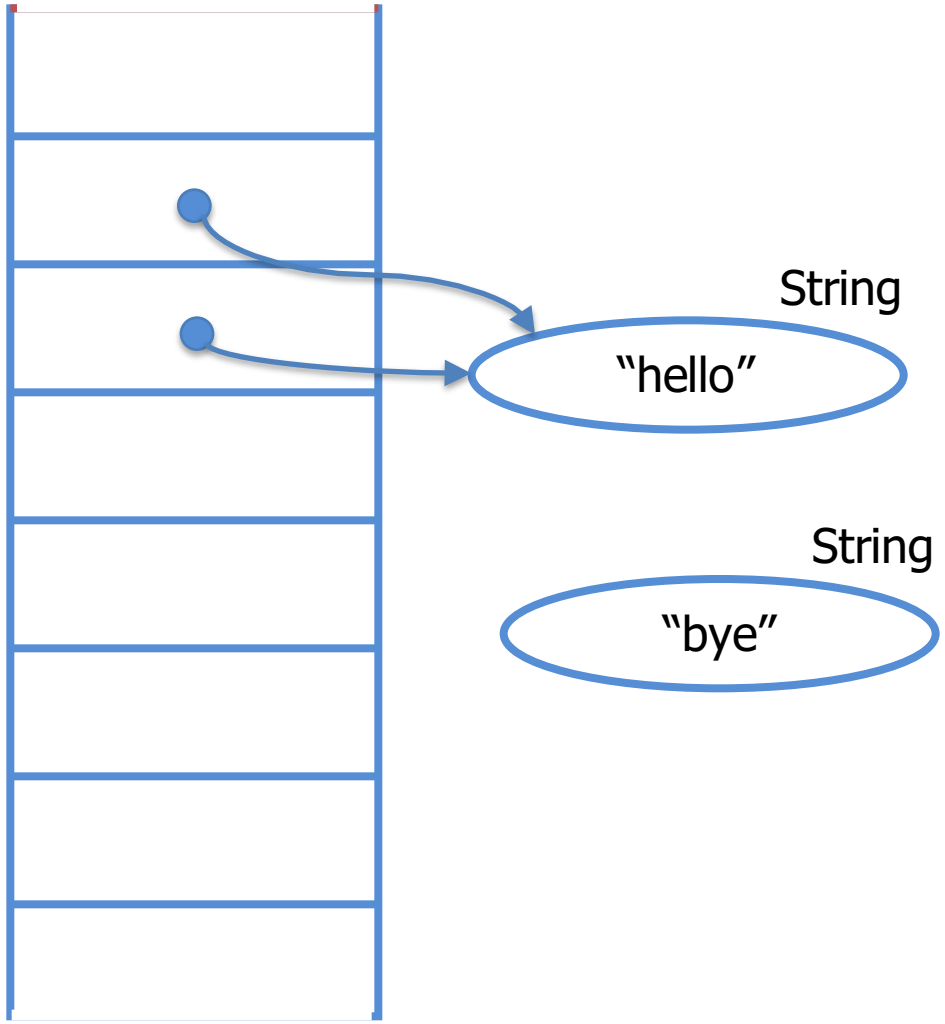
Partage de références

```
public static void doThing() {
    …
    String s = new String (“hello”);
    String t = s;
    …
}
```



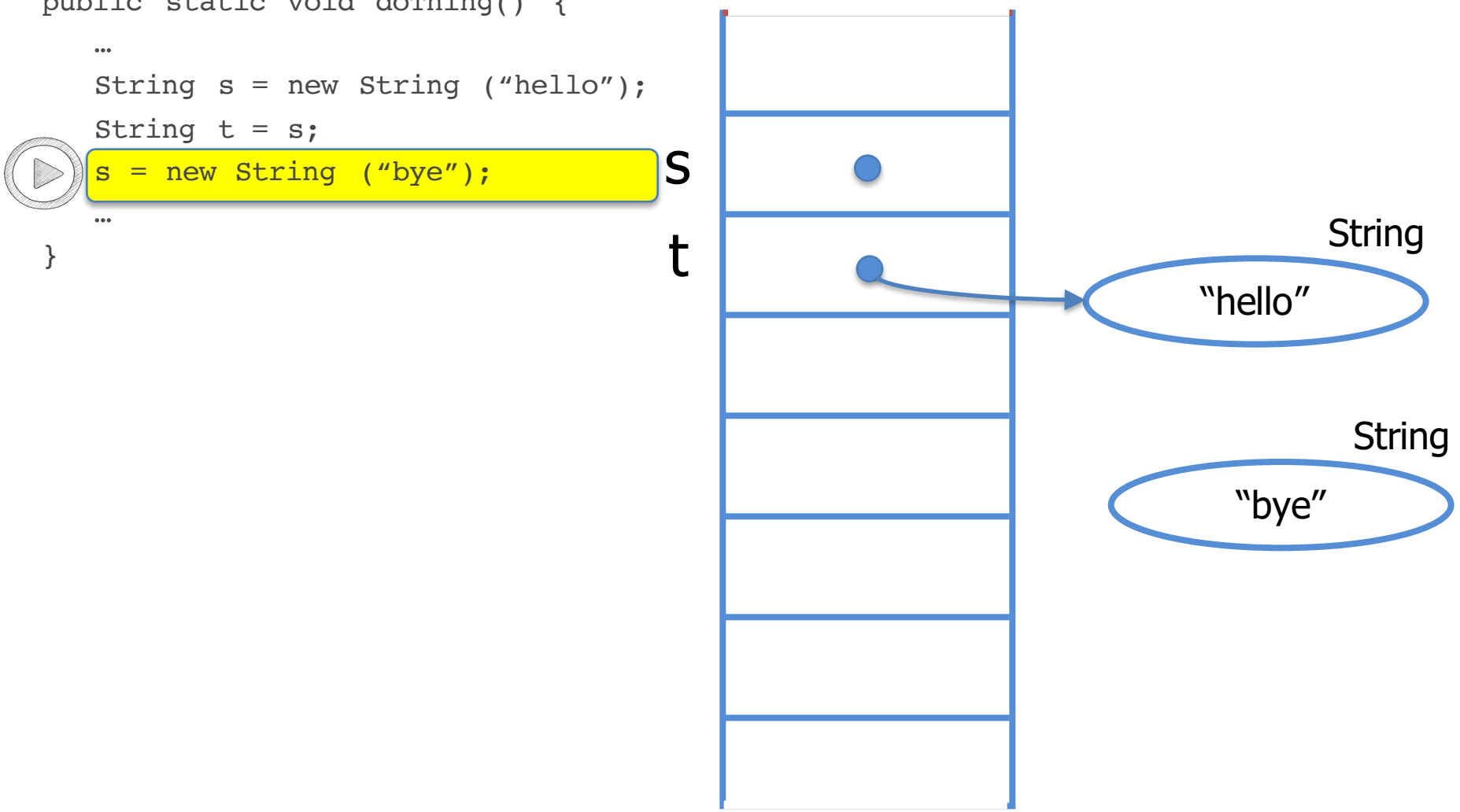
Partage de références

```
public static void doThing() {
    …
    String s = new String (“hello”);
    String t = s;
    s = new String (“bye”);
    …
}
```



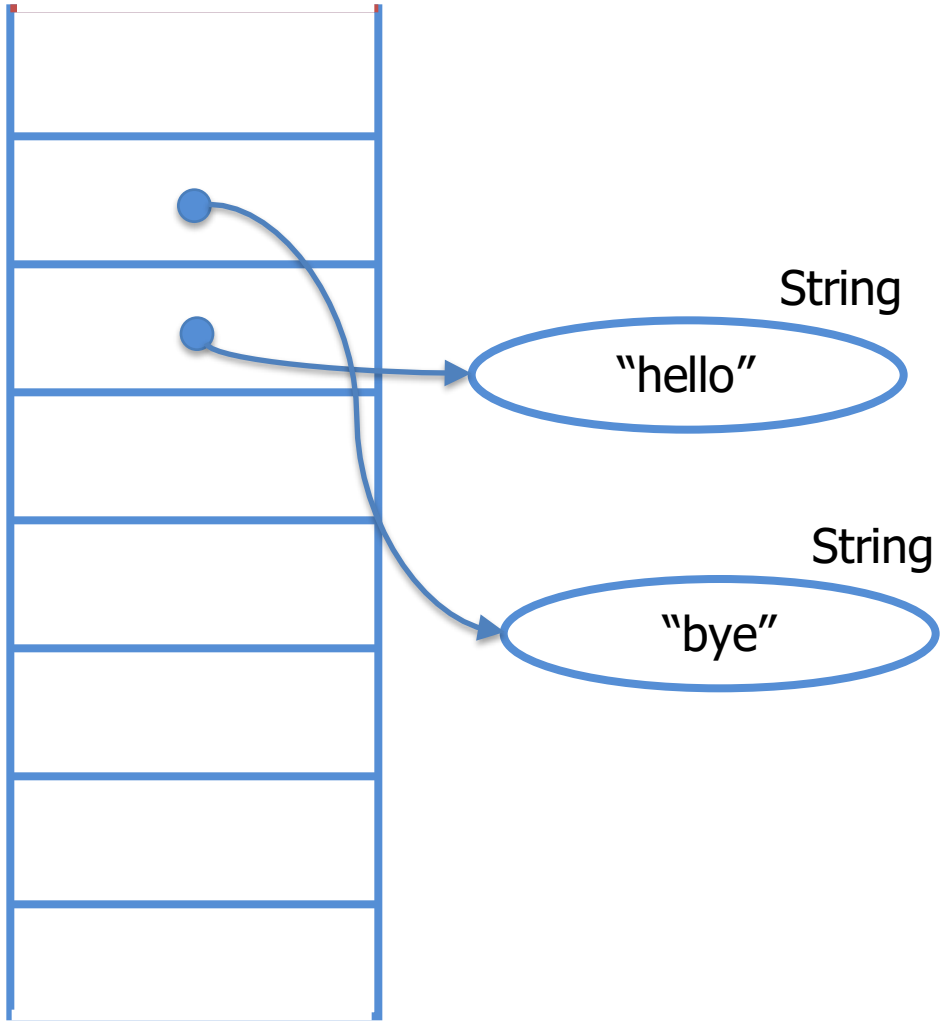
Partage de références

```
public static void doThing() {
    …
    String s = new String ("hello");
    String t = s;
    s = new String ("bye");
    …
}
```



Partage de références

```
public static void doThing() {
    …
    String s = new String (“hello”);
    String t = s;
    s = new String (“bye”);
    …
}
```

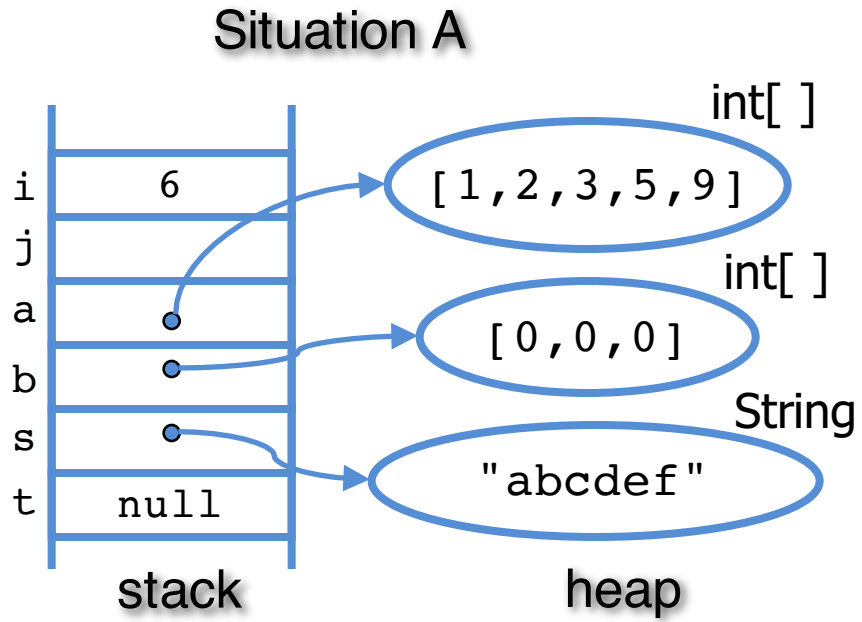


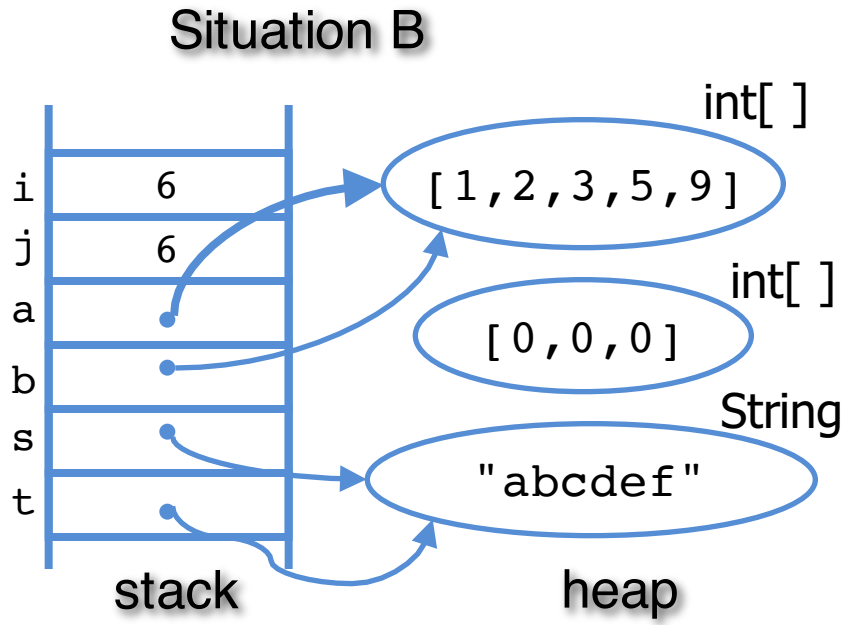
Partage de références

Autre exemple:

- Si ces instructions (voir ci-dessous) sont exécutées à partir de la situation A, on obtient la situation B:

```
j = i;
b = a;
t = s;
```



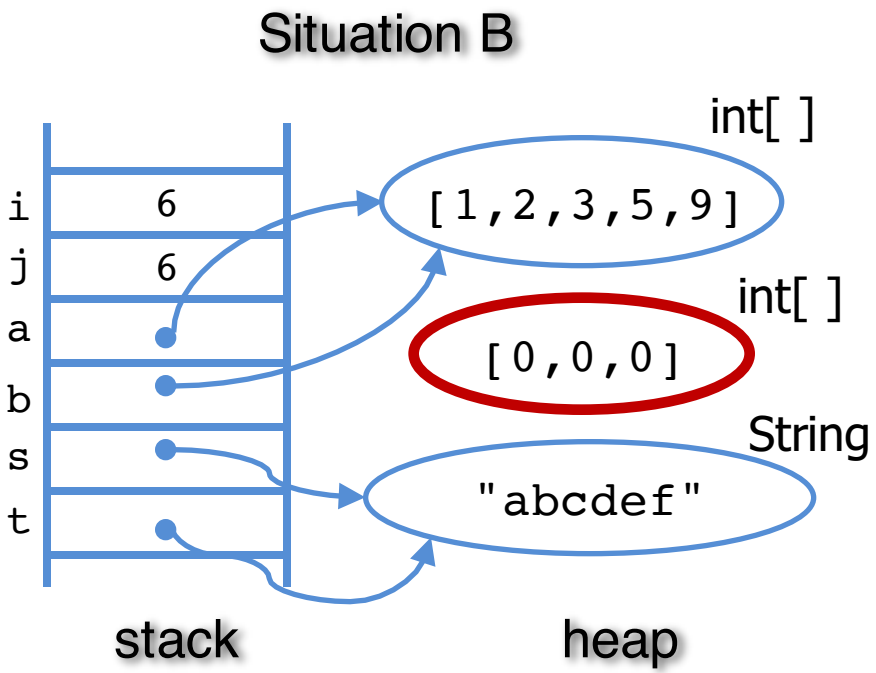


L'opérateur ==

- L'opérateur `==` détermine si deux variables ont :
  - pour les types de données, la même **référence**
  - pour les types primitifs, la même **valeur**
- L'évaluation de `<expr1> == <expr2>` retourne une valeur booléenne
- Utilisations typiques
  - Deux variables primitives ont-elles la même valeur?
    - Ex : `i == j`
  - Une variable de référence pointe-t-elle vers un objet ou non?
    - Ex : `t == null`
  - Deux variables partagent-elles une référence?
    - Ex : `a == b`

Garbage collection

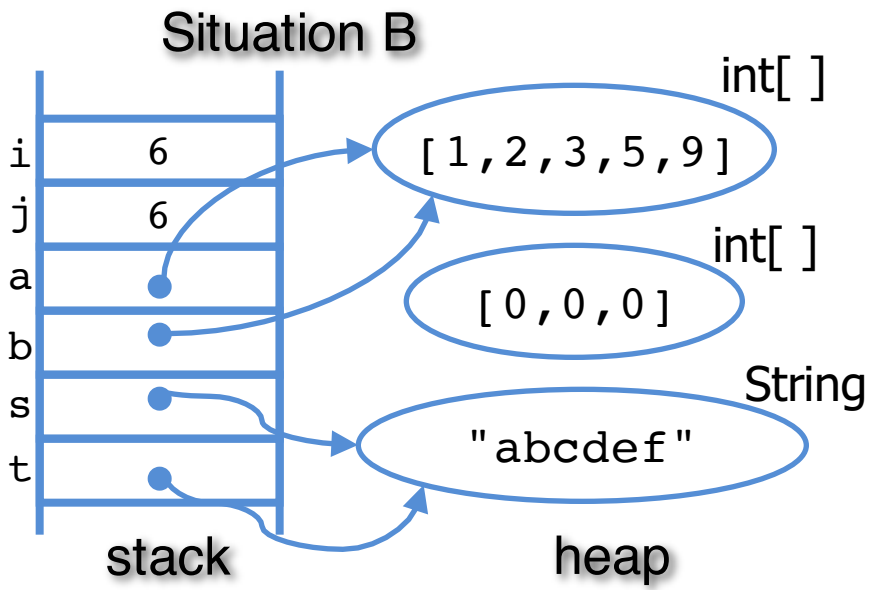
- Le **garbage collector** (ou **ramasse-miettes**) est la partie de la machine virtuelle Java chargée de libérer la mémoire (heap) occupée par les objets qui ne sont plus référencés
- Le processus de garbage collection se déclenche périodiquement ou lorsque la mémoire est saturée

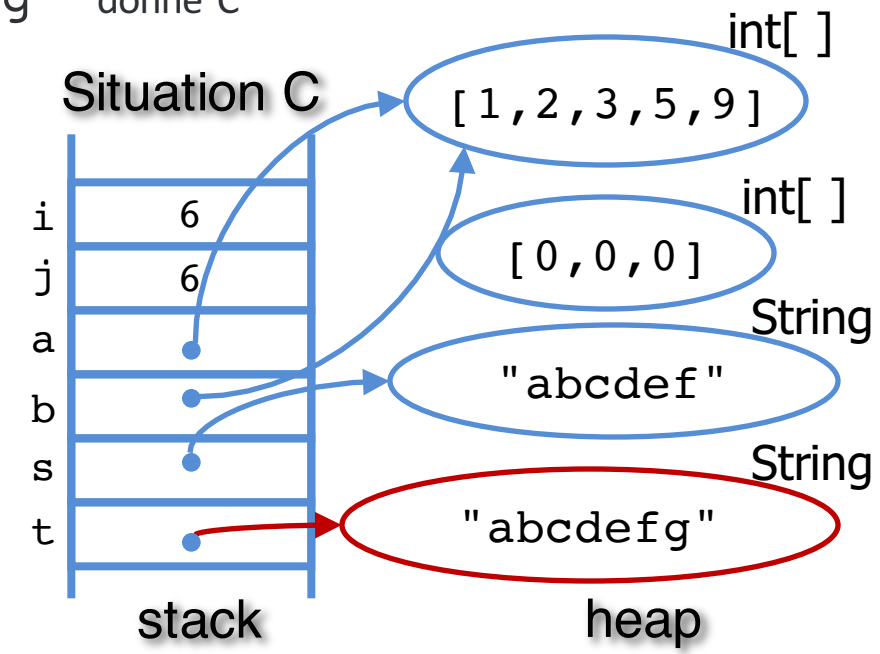


- Si le GC se déclenche dans la situation B, l'espace occupé par le tableau contenant `[0,0,0]` sera libéré.
- Le GC s'occupe de la destruction (Delete de CRUD) des objets. Le programmeur ne doit donc (généralement) pas s'en occuper.

Mutabilité

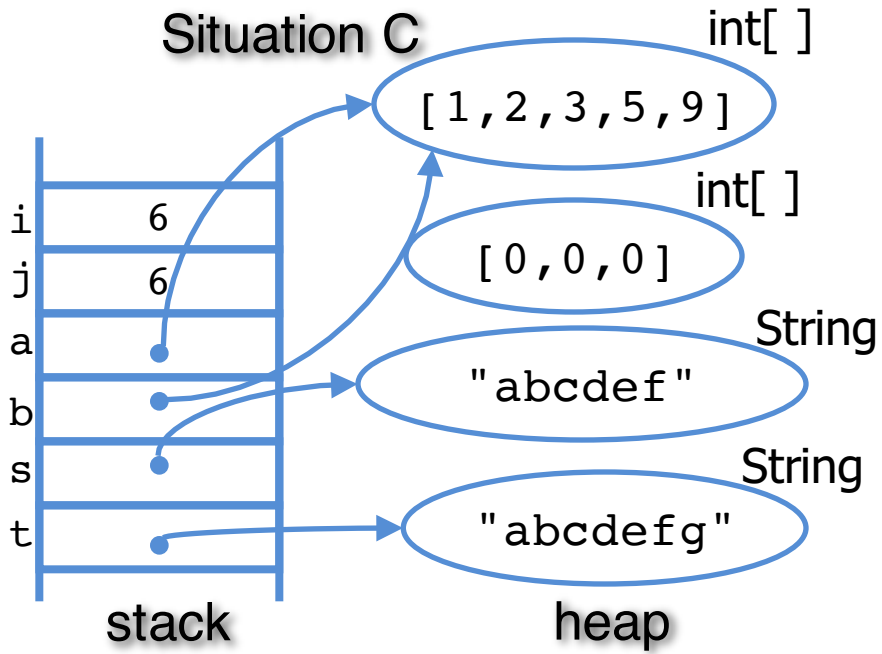
- Un objet est **mutable** ssi son **état** peut changer
- Dans le cas contraire, il est **immutable**
- Il appartient au **concepteur** de la classe de décider de sa mutabilité
- **Les `String` sont immutables**
- Il n'existe pas de méthode ou d'opérateur qui modifie (Update) une `String` une fois qu'elle a été créée
  - Ex : A partir de B, exécuter `t = t + "g"` donne C

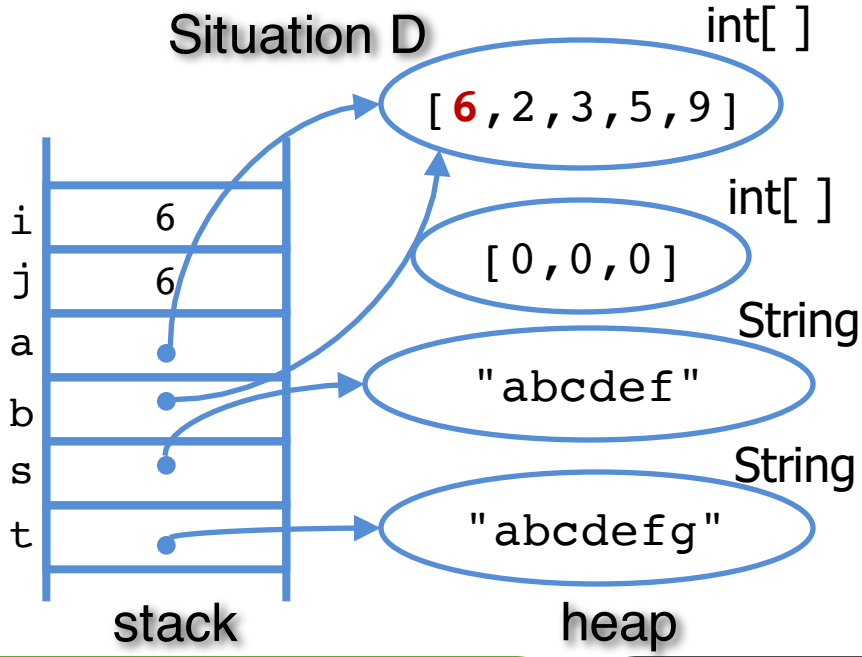




Mutabilité

- Par contre, **les tableaux sont mutables**
- `monTab [<expr1>] = <expr2>` a pour effet d'assigner à la `<expr1>`ème cellule du tableau la valeur de `<expr2>` (pour peu que `<expr1>` s'évalue à une valeur d'indice comprise dans les bornes du tableau et que `<expr2>` soit du type requis)
  - Ex : A partir de C, exécuter `b[0] = i` donne D





Mutabilité

- String est immutable, alors que StringBuilder est mutable
  - String.concat crée un nouvel objet String
  - StringBuilder.append mute l'ancien objet de type StringBuilder

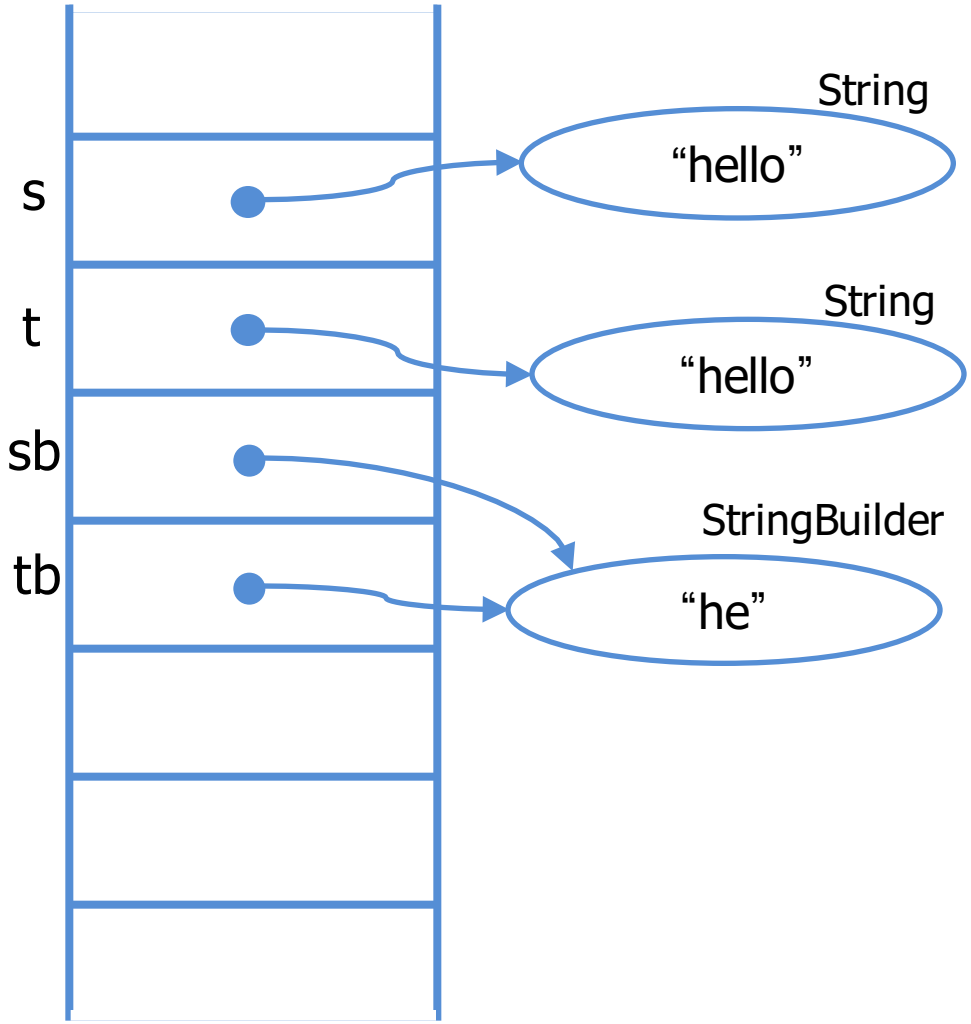
A string literal is a reference to an instance of class `String` (§4.3.1, §4.3.3).

Moreover, a string literal always refers to the *same* instance of class `String`. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method `String.intern`.

Mutabilité

String et StringBuilder

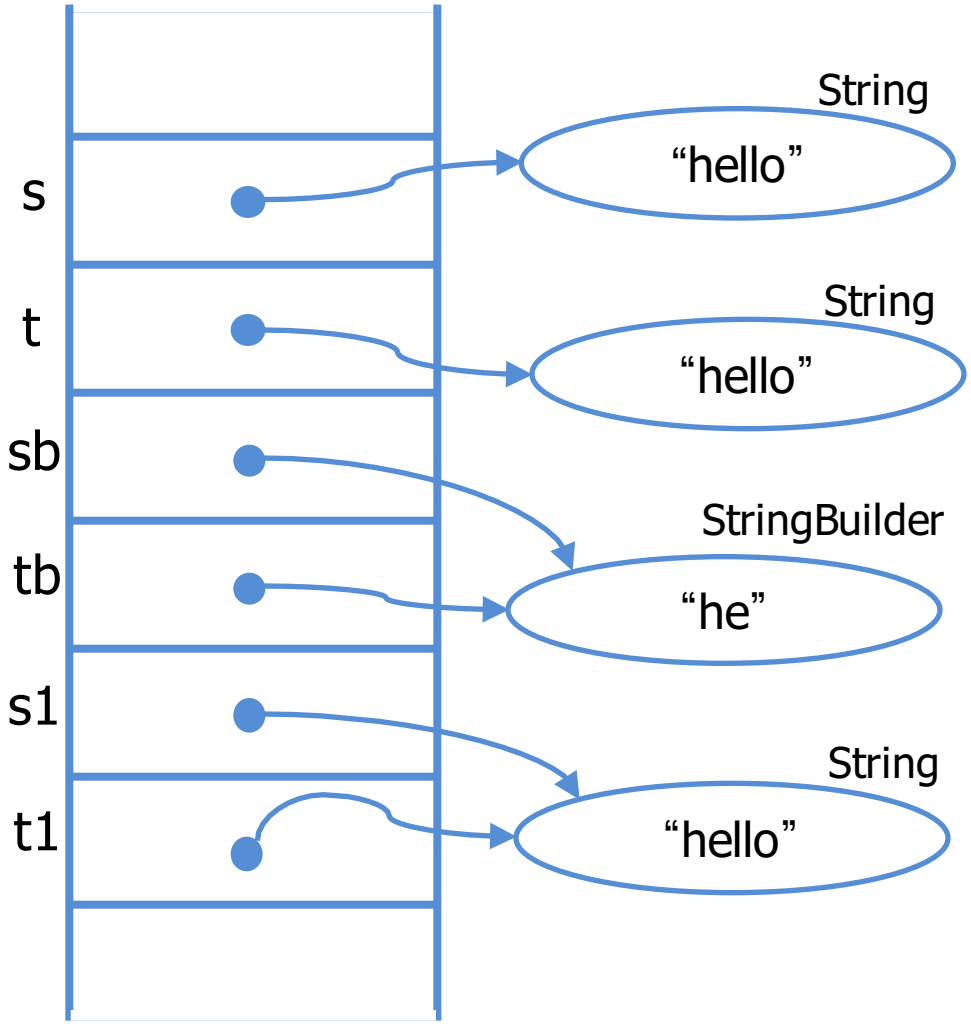
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

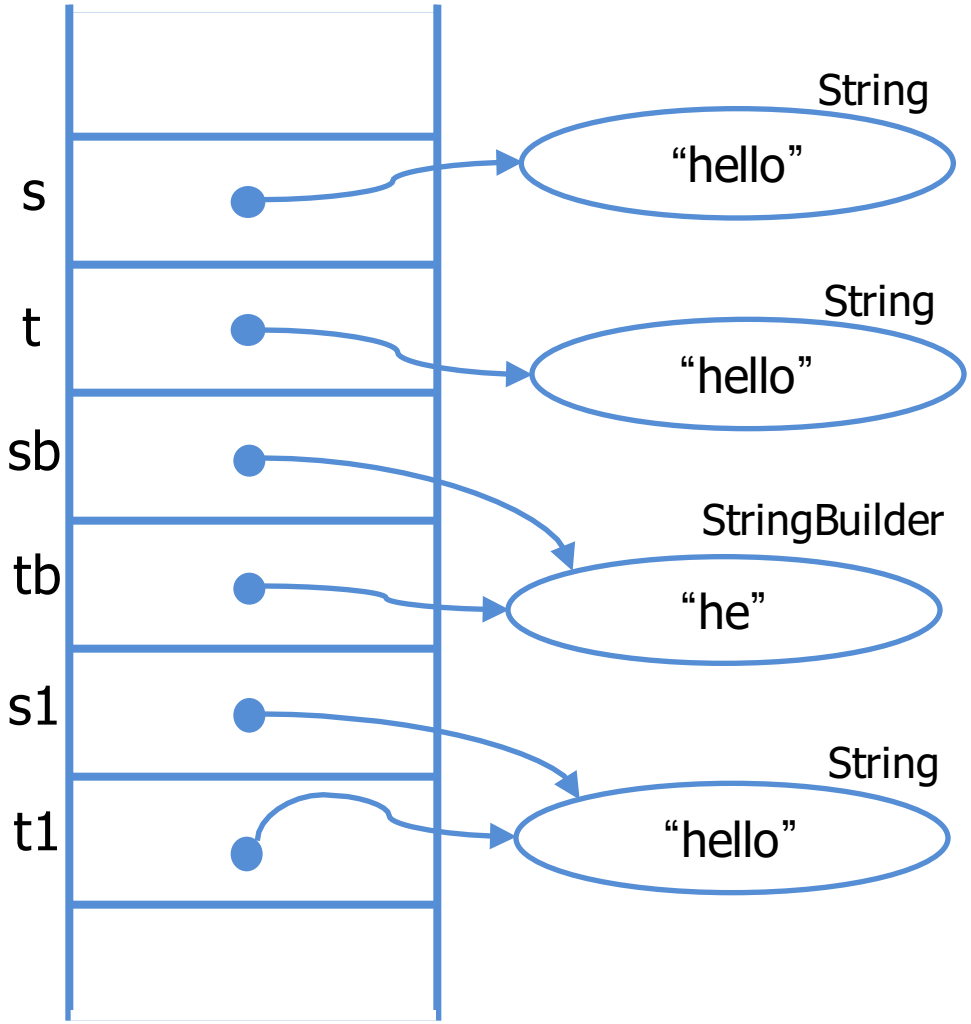
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



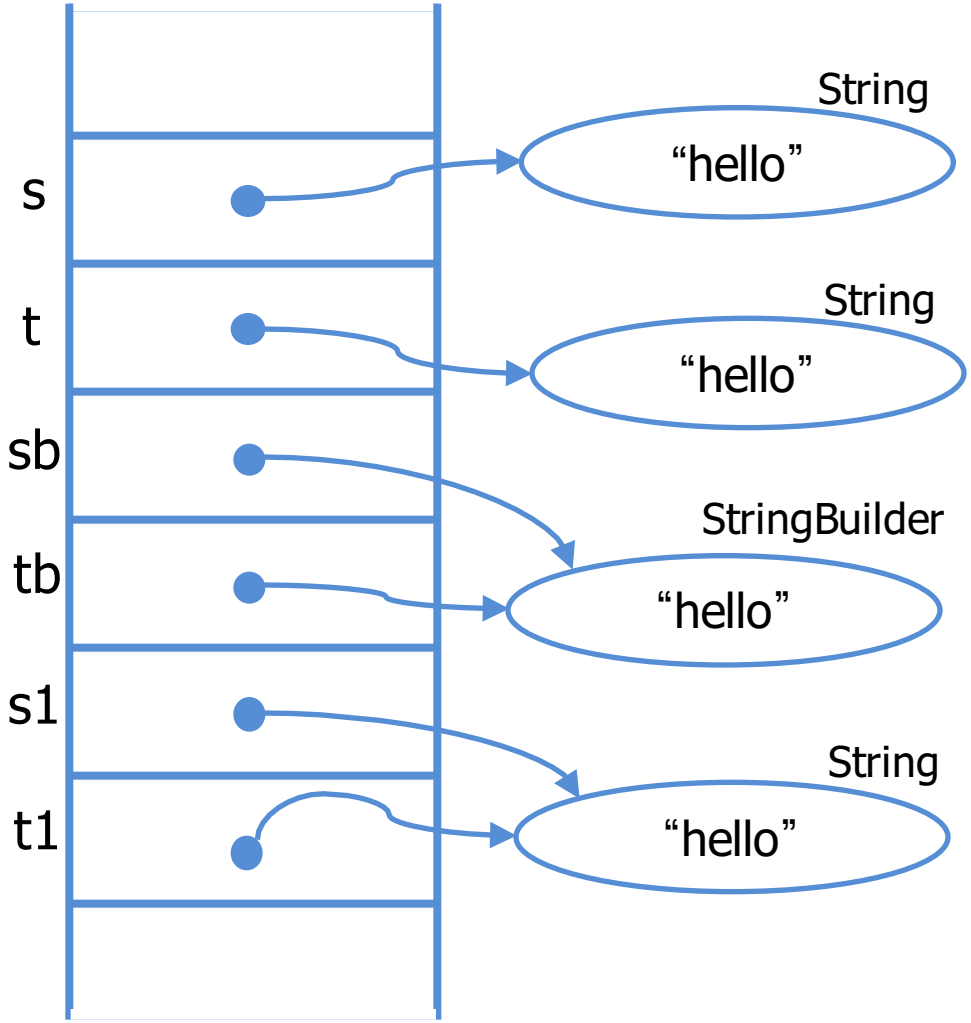
Les spécifications de String ne sont pas suffisantes pour déterminer si s, t, s1 et t1 référencent le même objet.

Et c’est tout-à-fait acceptable de faire ainsi !

Mutabilité

String et StringBuilder

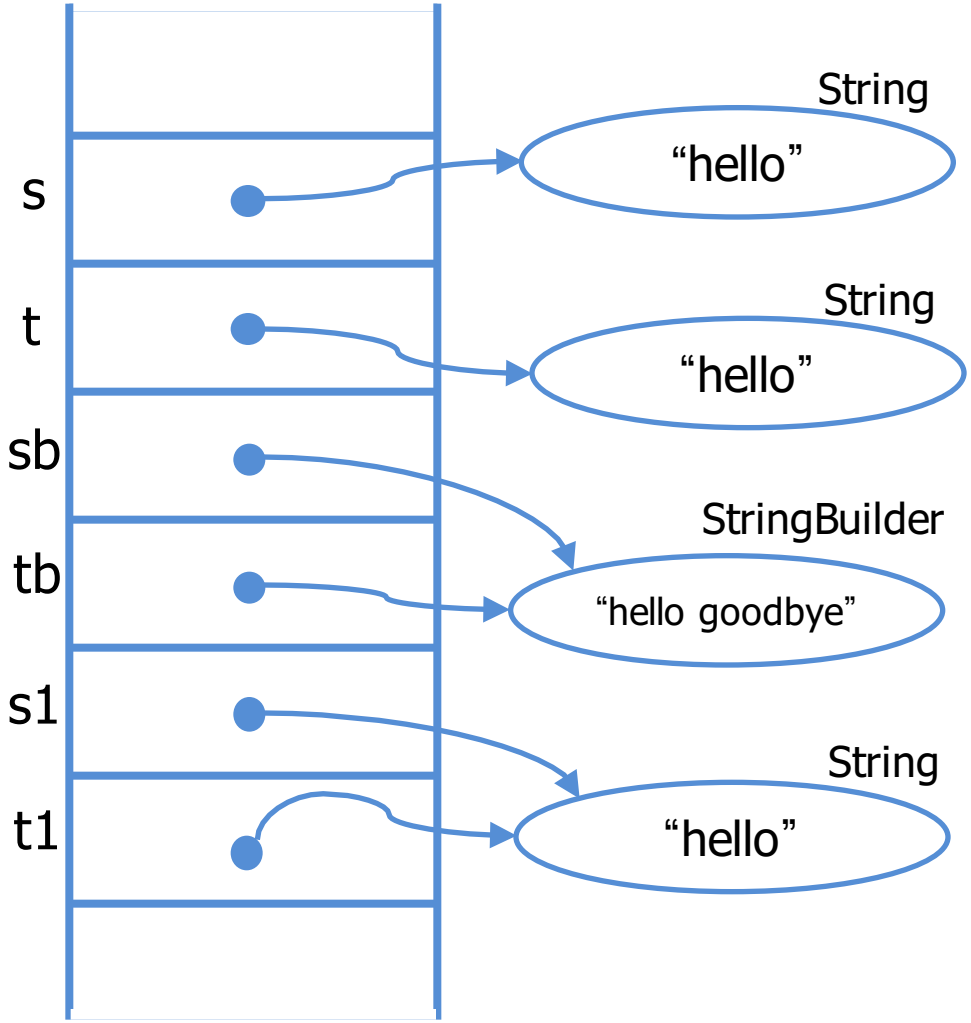
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

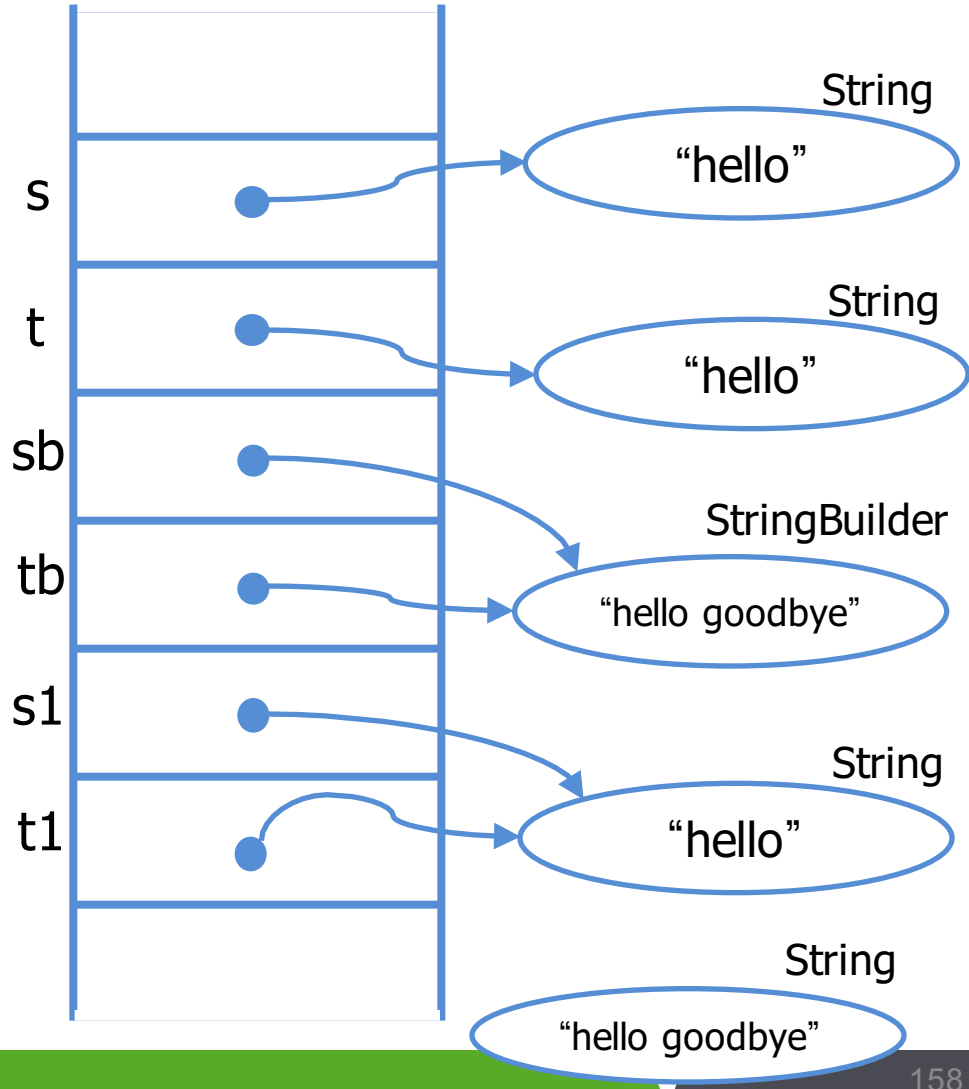
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

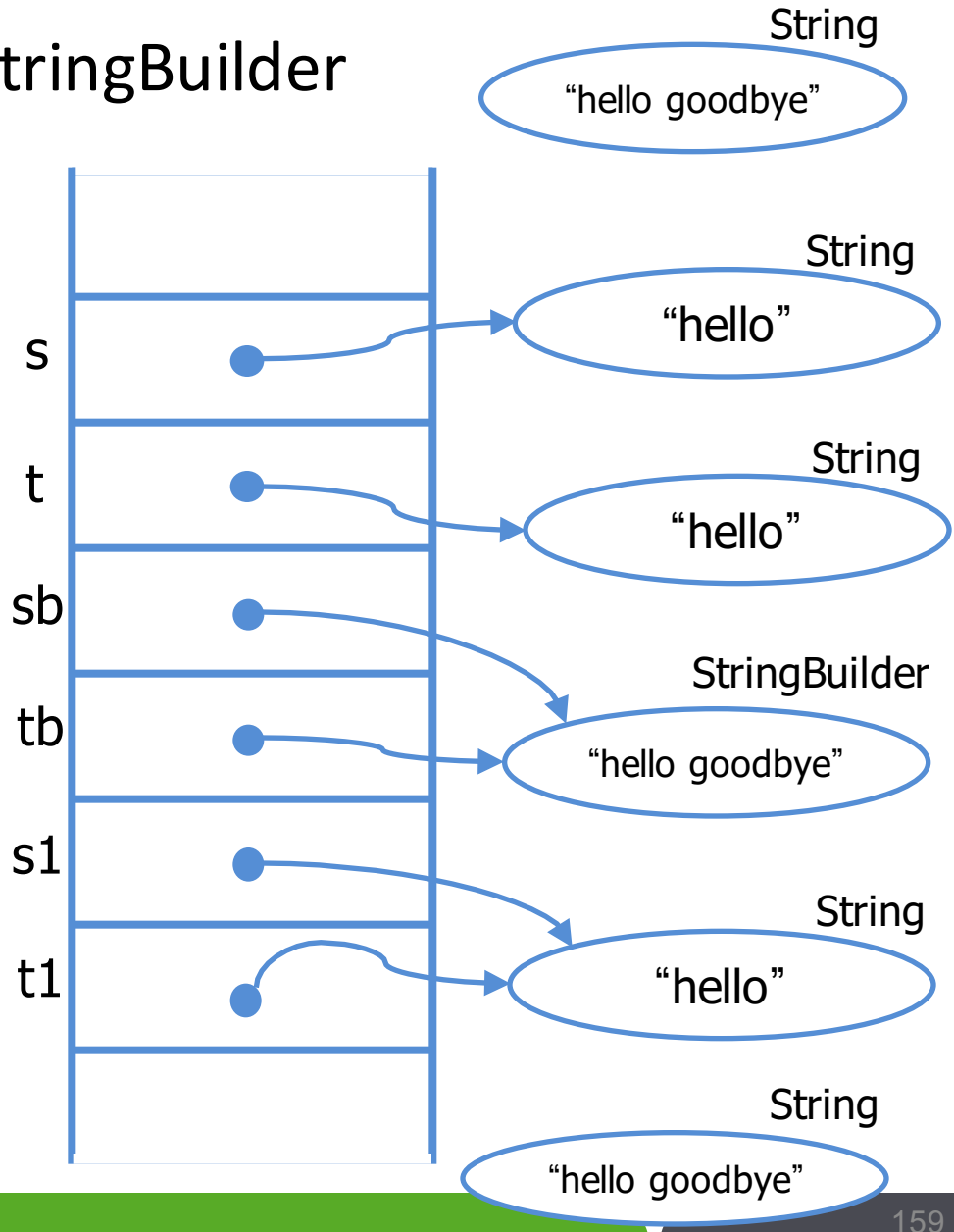
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

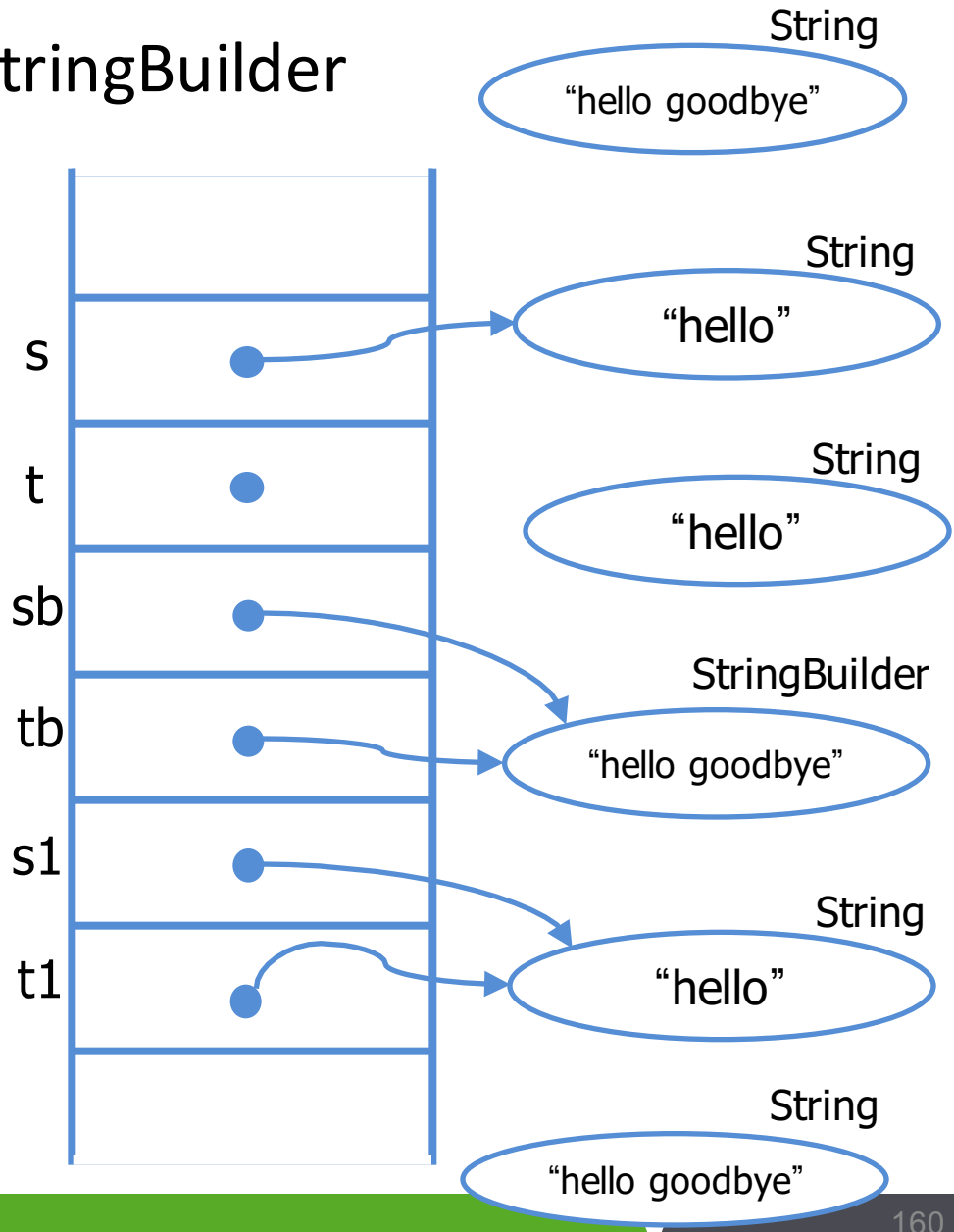
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

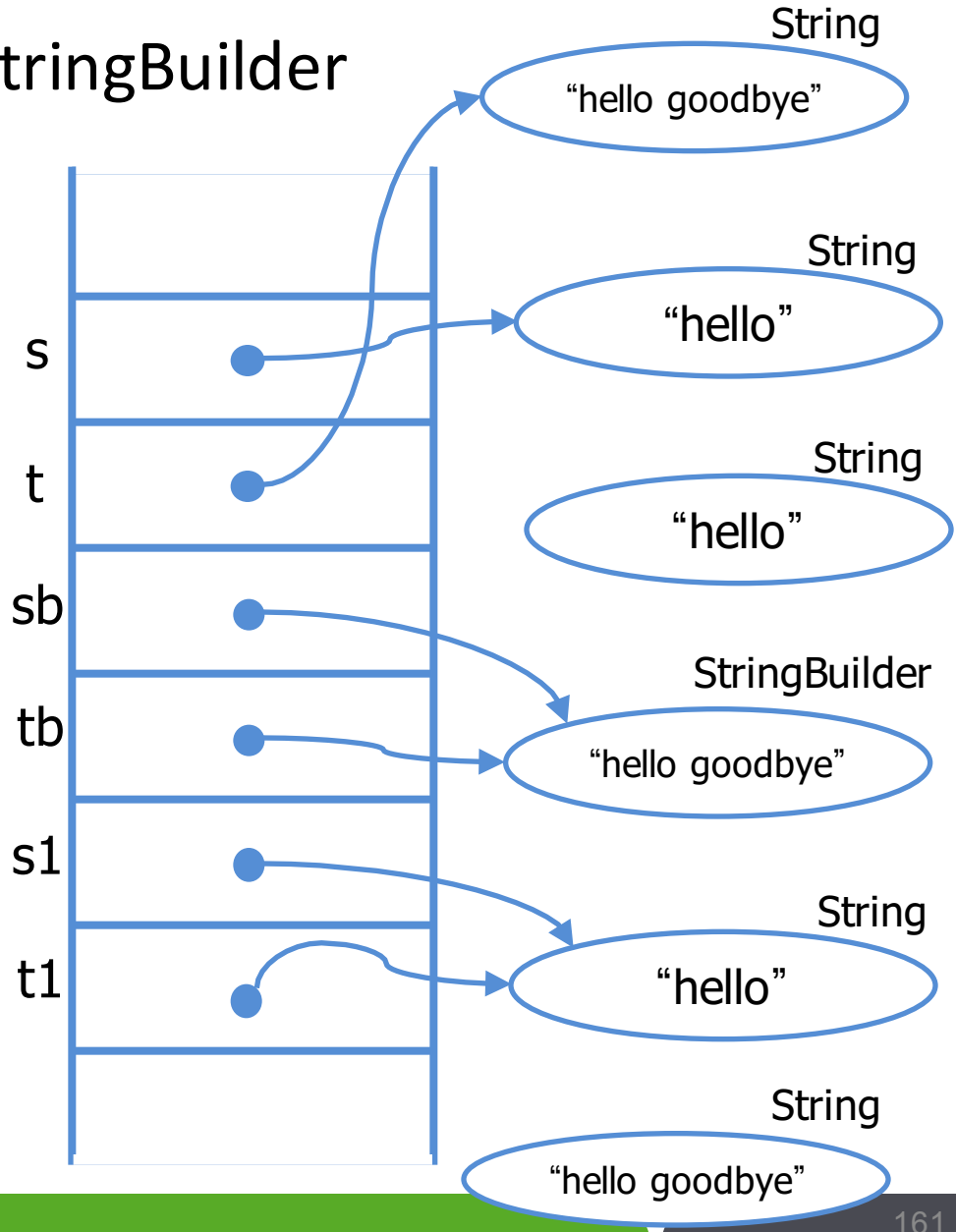
```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité

String et StringBuilder

```
public class Strings {
  public static void test (…) {
    String s = new String ("hello");
    String t = new String ("hello");
    StringBuilder sb =
            new StringBuilder("he");
    StringBuilder tb = sb;
    String s1 = "hello";
    String t1 = "hello";

    sb.append (“llo");
    tb.append (" goodbye!");
    s.concat (" goodbye!");
    t = s.concat (" goodbye!");
  }
}
```



Mutabilité et partage

- **Si un objet mutable est partagé par plusieurs variables, toute modification de l'objet effectuée à partir d'une des variables est visible au travers des autres**
- Ex : dans la situation D (voir slide 151), `a[0] == i` est évaluée à `true`, tout comme `b[0] == i`, bien que la modification du tableau ait été faite via `b`

Appels de méthodes

- En Java, l'appel à une méthode `m` se fait via l'instruction **`<expr>.m(<expr1>, <expr2>,...,<exprn>)`** et est traité comme suit:
  - `<expr>` est évaluée de manière à fournir la classe (si méthode statique) ou l'objet dont on invoque la méthode
    - si `<expr>` est évaluée à `null`, l'exécution sera interrompue par une `NullPointerException`
  - les expressions `<expr1>, <expr2>,...<exprn>` sont évaluées (de gauche à droite) de manière à fournir les paramètres effectifs de l'appel
  - un **activation record** est créé dans le stack réservant de la mémoire pour les paramètres formels et variables locales de la méthode
  - les paramètres formels prennent les paramètres effectifs pour valeurs dans le stack (passage de paramètres par **valeur**)
    - i.e. les valeurs des paramètres effectifs sont **copiées** dans l'activation record
  - le contrôle est passé à la méthode. C'est le **dispatching**. Ce mécanisme complexe sera abordé ultérieurement.

Appels de méthodes

- Exemple

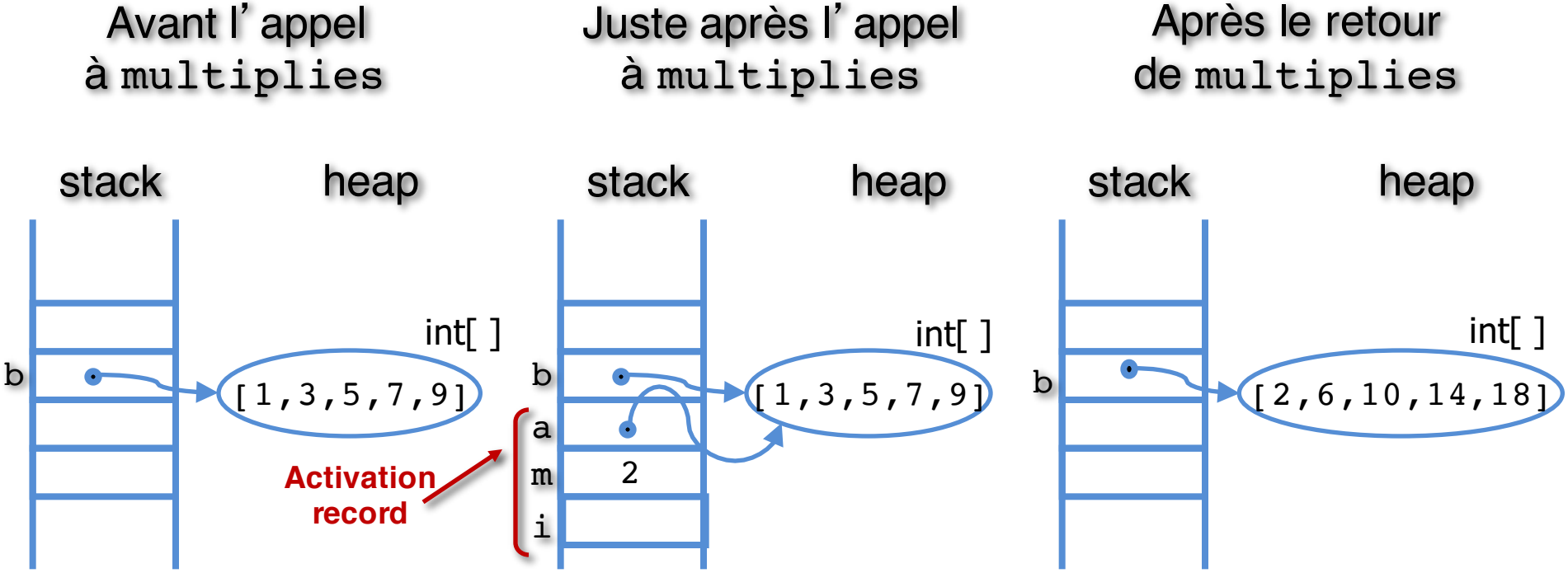
```
class Arrays {
   ...
   public static void multiplies(int[] a, int m) {
    // multiplie par m chaque élément du tableau d'entiers a
    if (a == null) return;
    for (int i = 0 ; i < a.length; i++) a[i]=a[i]*m;
   }
   ...
}
```

Appels de méthodes

- Exemple (suite).

Soit l'appel :

```
int[] b = {1,3,5,7,9};
Arrays.multiplies(b,2);
```



Type checking

- Java est un langage **fortement typé**
  - le **compilateur** vérifie que toutes les **assignations** et que tous les **appels** sont corrects du point de vue du typage
  - toute erreur à ce niveau est sanctionnée par une erreur de compilation
- La vérification du typage (**type checking**) requiert que
  - toute déclaration de variable indique le type de celle-ci
  - toute déclaration de méthode indique sa **signature**
    - les types des paramètres
    - le type du résultat
    - (les types d'exceptions renvoyés -- voir plus loin)

Type checking

- Ces informations de typage permettent au compilateur de déduire le **type apparent** de toute expression
- Le compilateur utilise alors le type apparent pour vérifier la correction des assignations et des appels
- Exemple

```
int y = 7;
int z = 3;
int x = Num.gcd(z,y);
```

Correspondance des types apparents des paramètres effectifs et des paramètres formels

```
class Num {
    public static int gcd(int n, int d) {
      while (n != d)
      if (n > d) n = n - d; else d = d - n;
      return n;
...
}
```

Type checking et type safety

- Le type checking à la compilation est un des mécanismes de Java permettant de garantir que les programmes (compilés!) sont **type safe**
  - i.e. aucune erreur de typage (type mismatch) ne peut survenir durant son exécution
  - i.e. il n'est pas possible que le programme manipule des données d'un type alors qu'il s'attend à ce qu'elles soient d'un autre type
    - exemple : `String s = new Commande(); // non valide`
- Au total, ces mécanismes sont au nombre de 3. Les deux autres sont
  - la gestion automatique de la mémoire aka **automatic storage management**
  - la vérification des (non-)dépassements de limites dans les tableaux aka **array bound checking**

Type checking et type safety

Type safety

=

type checking

\+ **automatic storage management**

\+ **array bound checking**

Automatic storage management

- Il s'agit de l'allocation et la désallocation automatique de la mémoire dans le heap
- En Java, c'est la machine virtuelle qui s'occupe de tout cela sans que le programmeur doive s'en soucier
  - elle s'occupe de l'allocation lors de chaque création d'objet (voir le traitement du `new`, ci-avant)
  - elle s'occupe de la désallocation lorsque l'objet n'est plus référencé (voir garbage collection, ci-avant)
- En C et C++, ces mécanismes n'existent pas. C'est au programmeur qu'il incombe d'écrire lui-même le code d'allocation et de désallocation de la mémoire

Automatic storage management

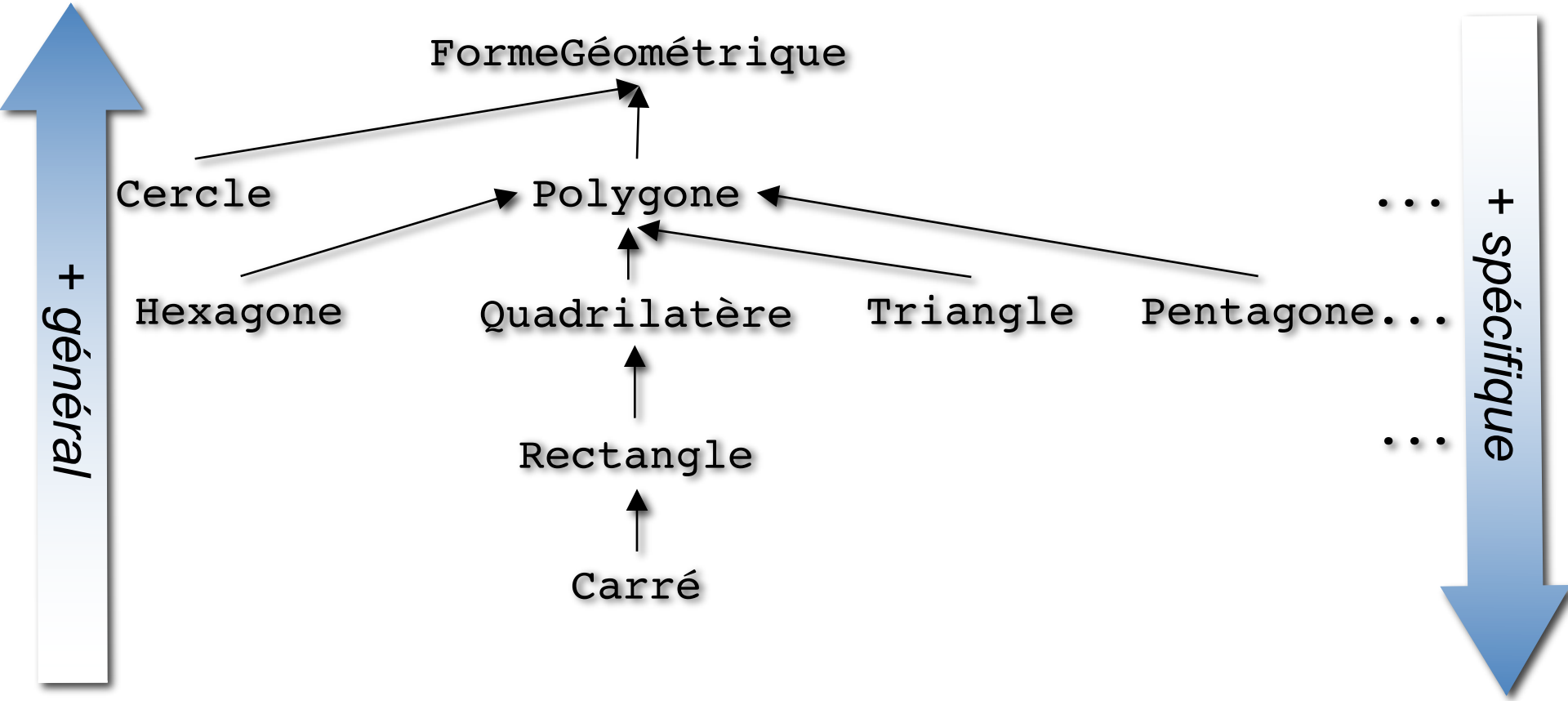
- Exemple d'allocation dynamique de mémoire en C

```
struct foo *ptr;
...
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

- La désallocation se fait via un `free (ptr);`
- Cette façon de faire comportait deux problèmes majeurs (et corrélés)
  - complexification du code
  - risques d'introduction d'erreurs, dont les fameuses « dangling references »
    - i.e. désallocation d'espace mémoire toujours référencé par le programme (si partage de références)

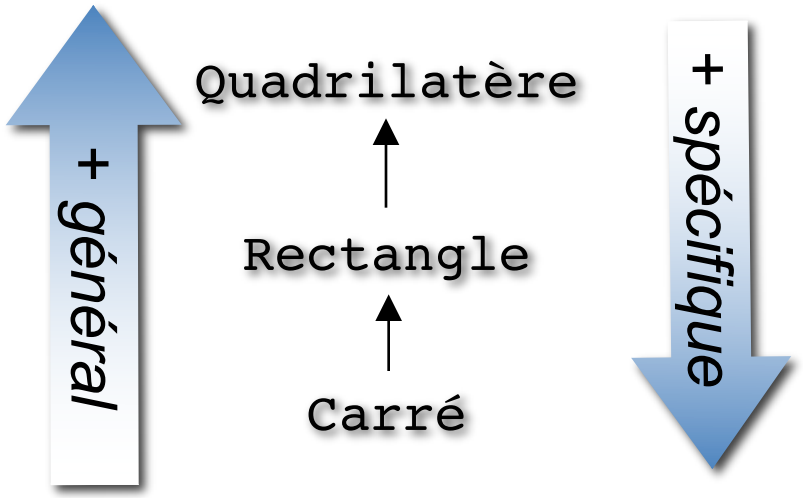
Hiérarchie des types

- Souvenez-vous de l'exemple introductif…



Hiérarchie des types

- En Java, les types non primitifs (i.e. les types d'objets) sont organisés de manière **hierarchique**
- Un type donné T peut avoir plusieurs **supertypes**, ses *ancêtres* dans la hiérarchie
- Il est dit être un **sous-type** (subtype) de chacun de ses supertypes
- Exemple
  - `Quadrilatère` un **supertype** de `Rectangle` et `Carré` qui en sont des **sous-types**
  - `Rectangle` est un **supertype** de `Carré`
  - `Rectangle` et `Carré` sont des **sous-types** de `Quadrilatère`



Hiérarchie des types

- Dans la littérature, on trouve parfois d'autres terminologies
  - **superclasse** (superclass) ≈ supertype
  - **sous-classe** (subclass) ≈ sous-type
  - « un type T1 **étend** (extends) un type T2 » ≈ « T1 est un sous-type de T2 »

- L'organisation hiérarchique des types est un **mécanisme d'abstraction** permettant, quand c'est approprié, d'**ignorer les différences** entre types pour ne plus voir que ce que leurs comportements ont de **commun** et qui se trouve dans leurs **supertypes communs**.

Hiérarchie des types

- La relation de sous-typage est
  - **transitive**
    - si R est un sous-type de S et S est un sous-type de T, alors R est un sous-type de T
  - **réflexive**
    - T est un sous-type de lui-même

Hiérarchie des types
Principe de susbstitution

- **« Si S est un sous-type de T, les objets de type S doivent pouvoir être utilisés dans tout contexte qui requiert des objets de type T »**
- Implications
  - **toutes les méthodes de T doivent être disponibles dans S**
    - vérifié par le compilateur Java
  - **les appels à ces méthodes doivent produire le même comportement dans S et T**
    - **non vérifiable** par le compilateur puisque cela reviendrait à pouvoir prouver que deux programmes ont le même comportement
    - doit être garanti par le programmeur
- Nous verrons ultérieurement comment construire des types qui vérifient le principe de substitution
- Pour l'instant, nous nous contenterons d'utiliser des types prédéfinis supposés offrir ces garanties

Hiérarchie des types
`Object`

- En java, il existe un type `Object` qui est supertype de tous les types (prédéfinis ou non).
- C'est l'ancêtre commun, le sommet de la hiérarchie
- `String, StringBuilder` et les types de tableaux sont donc, comme tous les autres, des sous-types d'`Object`
- `Object` possède notamment deux méthodes
  - `boolean equals(Object o)`
  - `String toString()`
- L'utilité de ces méthodes sera discutée plus avant mais, à ce stade, retenons qu'elles peuvent être invoquées sur tout objet Java
  - ex : `boolean lesMemes = s.equals(t)` est valide quels que soient les types respectifs de `s` et `t` (sauf pour les types primitifs, bien entendu)

Hiérarchie des types
Validité des assignations

- Règle : en Java, l'assignation `v=<expr>` (avec `v` variable de référence) est valide si le type (apparent) de `<expr>` est un sous-type du type (apparent) de `v`

- Exemple

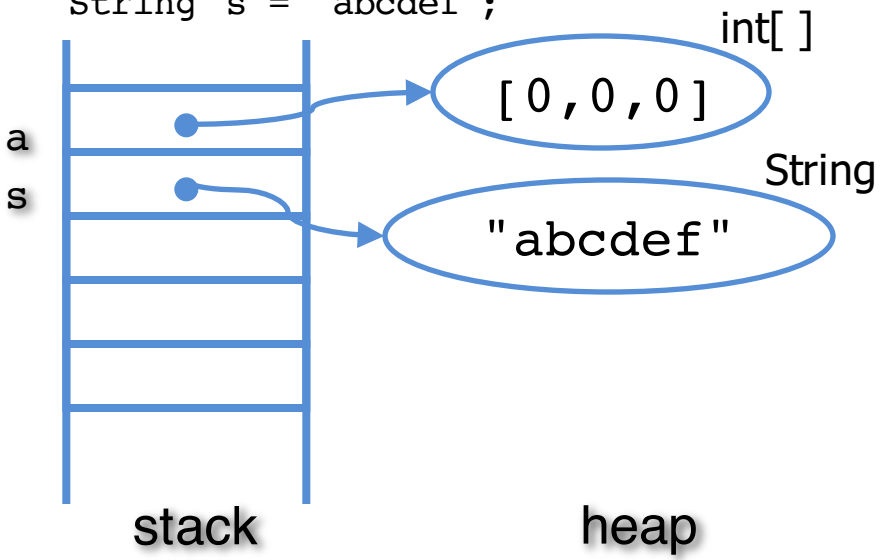
```
...
int[] a = new int[3];
String s = "abcdef";
Object o1 = a;
Object o2 = s;
...
```

Hiérarchie des types
Validité des assignations

- Cet extrait de code est parfaitement valide et produit le résultat suivant

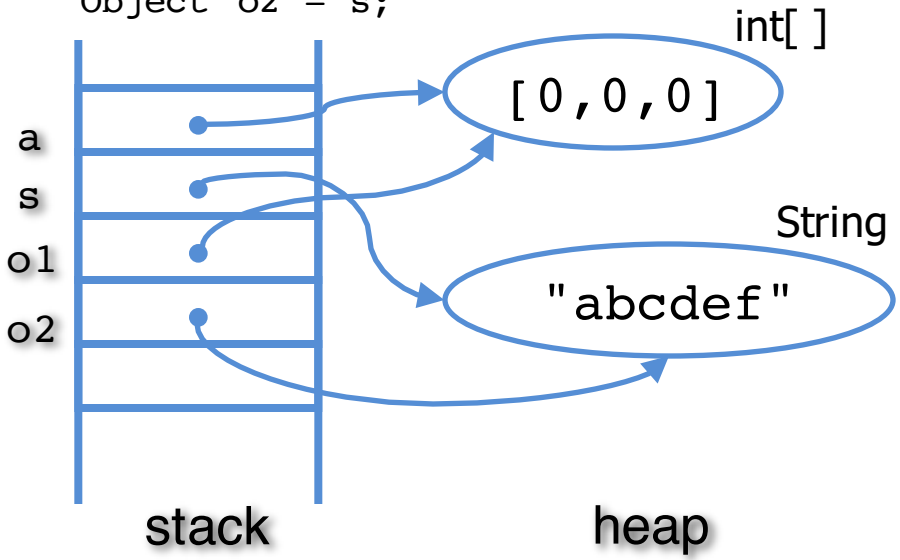
Après

```
int[] a = new int[3];
String s = “abcdef“;
```



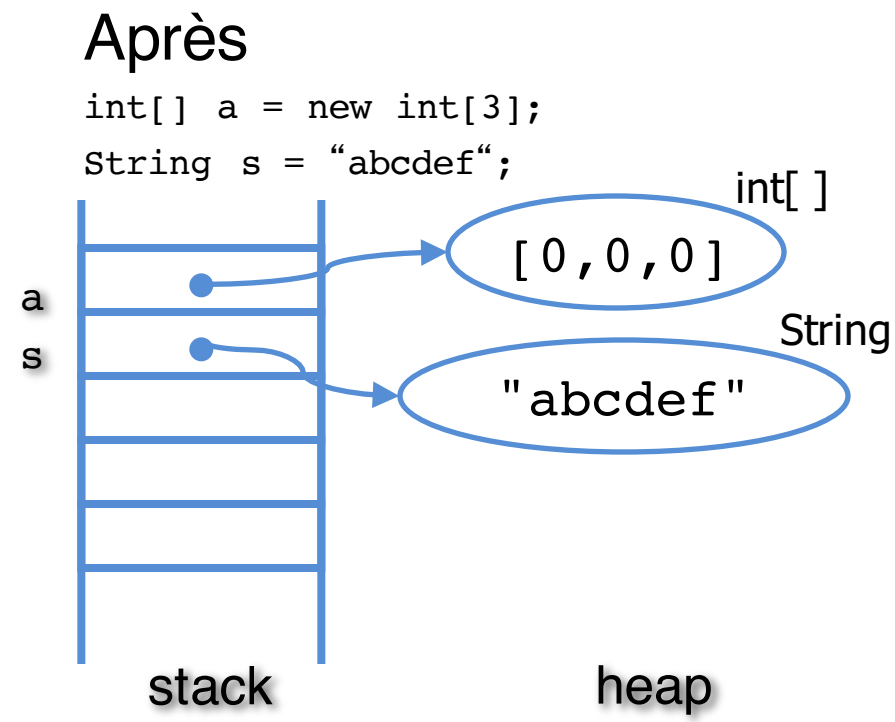
Après

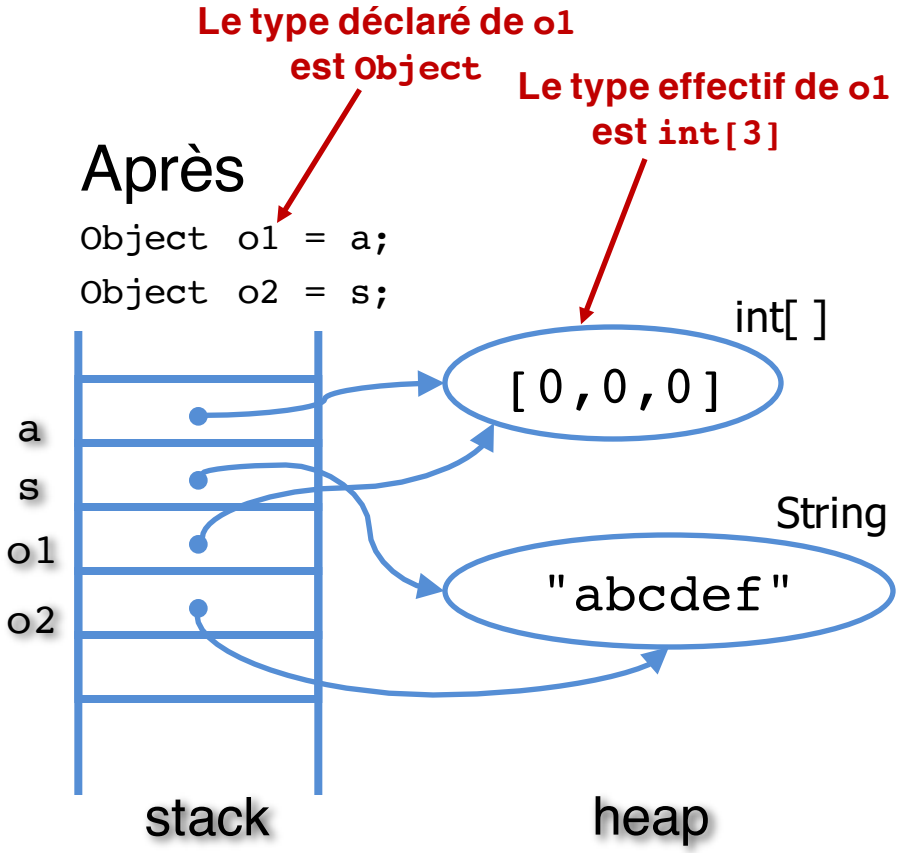
```
Object o1 = a;
Object o2 = s;
```



Hiérarchie des types
Validité des assignations

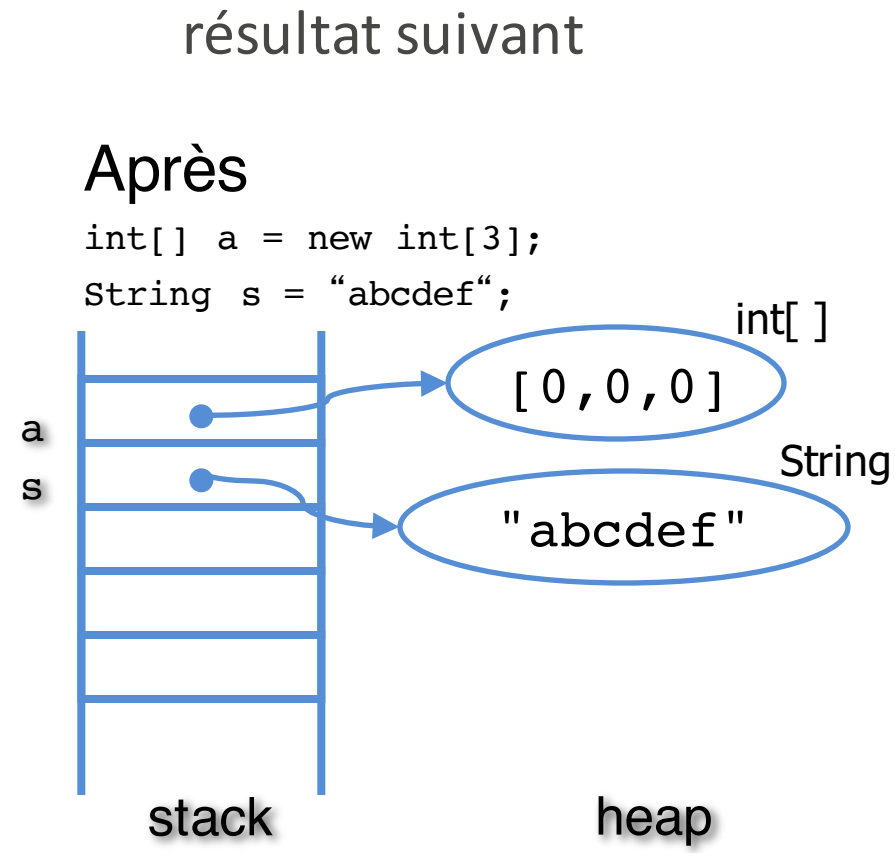
- Cet extrait de code est parfaitement valide et produit le résultat suivant

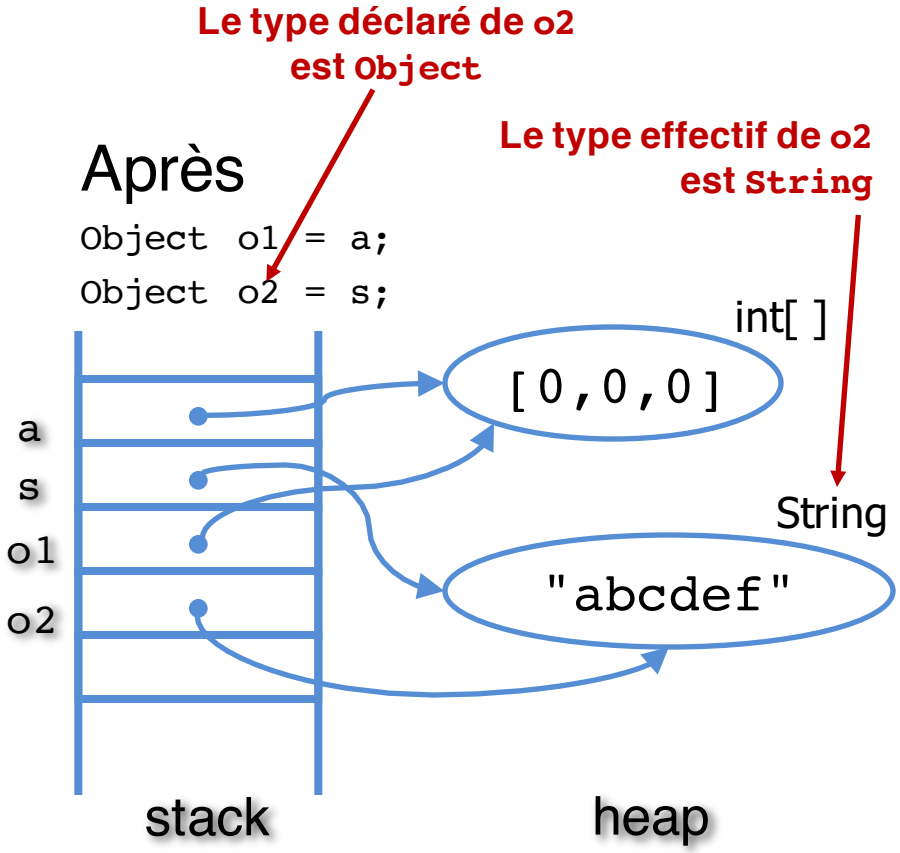




Hiérarchie des types
Validité des assignations

- Cet extrait de code est parfaitement valide et produit le résultat suivant





Hiérarchie des types
Type déclaré vs type effectif

- **Type déclaré** (apparent type, compile-time type) = type d'une variable tel qu'il est donné dans sa **déclaration** (il est le même pendant toute l'/les exécution(s))
- **Type effectif** (actual type, runtime type) = type d'une variable à un moment donné de l'**exécution**, déterminé par le type de l'objet qu'elle référence dans le heap (celui-ci est déterminé à la création de l'objet)
- Le type checking (à la compilation) est toujours effectué sur base du type **déclaré**

Hiérarchie des types
Validité des appels de méthodes

- Règle: la validité des appels de méthodes est vérifiée de manière analogue aux assignations i.e.
  - le type déclaré de l'objet sur lequel on invoque la méthode doit être un sous-type du type pour lequel la méthode a été définie
  - le type déclaré d'un paramètre effectif doit être un sous-type du type déclaré du paramètre formel
- Exemple

```
...
if (o2.equals("abc")) ... ; // valide
if (o2.length == 0) ... ; // non valide
if (s.size() == 0 ) ... ; // valide
...
```

- NB : seules les méthodes d'`Object` (i.e. `equals, toString,...`) peuvent être invoquées sur n'importe quel type d'objet

Hiérarchie des types
Type déclaré vs type effectif

- Les deux règles de validité que l'on vient de voir ont pour conséquence que, **à l'exécution, le type effectif d'une variable ou d'une expression est toujours un sous-type de son type déclaré**

Hiérarchie des types
Casting

- Il est cependant parfois utile d'indiquer dans le programme le type effectif de l'objet à l'exécution
- C'est utile, par exemple,
  - pour pouvoir invoquer une méthode sur un objet dont le type déclaré l'interdit mais dont on sait que le type effectif le permet
    - `ex:if (o2.size() == 0) ... ; // non valide`
  - pour assigner à une variable la valeur d'une expression dont le type déclaré l'interdit mais dont on sait que le type effectif le permet
    - `ex:s = o2; // non valide`

Hiérarchie des types
Casting

- Cela peut se faire via le **casting**
- Exemple

```
if (((String)o2).size() == 0) ... ; // valide
s = (String)o2; // valide
```

- Le casting provoque la survenance d'une vérification de type à l'exécution
  - `(String)o2` demande la vérification du fait que le type effectif de `o2` soit `String`
- Si la vérification réussit, les instructions sont exécutées
- Si la vérification échoue, au lieu d'exécuter les instructions, la machine virtuelle renvoie une `ClassCastException`
  - Dans notre cas, `(String)o2` réussit et le test ainsi que l'affectation sont exécutés

Conversion de types

- Java autorise certaines **conversions implicites** de valeurs d'un type en valeurs d'un autre type
- **Seuls les types primitifs** sont concernés par la conversion
- Exemple: la conversion de caractères (`char`) en types numériques (`int`, par ex)

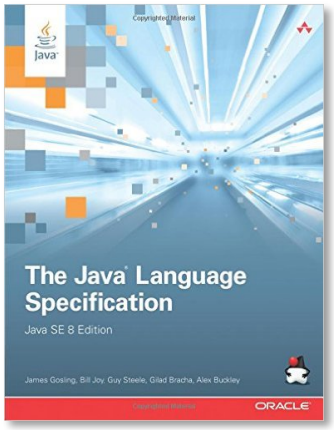
```
char c = 'a';
int n = c; // valide
```

- Lorsque le compilateur rencontre des conversions implicites, il génère les instructions qui effectuent la conversion
- On dira que les `char` peuvent être **élargis** (**widened**) en `int`
  - Plus généralement, les char peuvent être élargis à tous les types numériques (sauf `short`)

Conversion de types

- Les conversions de types implicites autorisées en Java sont données dans le chapitre 5 du « The Java Language Specification »

- Exemple : widening de types primitifs autorisés
  - `byte` *vers* `short`, `int`, `long`, `float` ou `double`
  - `short` *vers* `int`, `long`, `float` ou `double`
  - `char` *vers* `int`, `long`, `float` ou `double`
  - `int` *vers* `long`, `float` ou `double`
  - `long` *vers* `float` ou `double`
  - `float` *vers* `double`



- Pas à connaître mais vous y référer en cas de doute

Overloading (surcharge)

- En Java, comme dans la plupart des langages, les opérateurs sont ***overloadés*** (**surchargés**)
- Exemple
  - l'opérateur + est défini pour plusieurs types d'opérandes : `int + int, float + float, int + float, float + int, int + long,` etc…
- En outre, en Java, le programmeur peut overloader les méthodes qu'il définit
- Exemple: soit la classe C possédant les méthodes

```
static int comp(int, long)  // def 1
static int comp(long, int)  // def 2
static int comp(long, long) // def 3
```

Overloading (surcharge)

- On dit que la classe `C` surcharge le nom de méthode `comp`
- L'overloading est utile mais il peut être source d'ambiguïtés pour le compilateur
- Exemple :
  - soit les déclarations `int x; long y; float z;`
  - or, Java admet la conversion d'`int` en `long` ainsi que la conversion de `long` en `float`
  - dès lors, à quelle définition doit obéir l'appel `C.comp(x,y)` ?
    - def. 1 ?
      - **OK** car correspondance parfaite des types des paramètres effectifs et formels
    - def. 2 ?
      - **non**, car conversion de `long` en `int` non autorisée
    - def. 3 ?
      - **OK aussi** car `x` peut être converti en `long` !

Overloading (surcharge)

- Lorsque plusieurs définitions de méthodes surchargées peuvent convenir à un même appel de méthode, le **compilateur** résout l'ambiguïté en choisissant ***la plus spécifique***
- « Une définition de méthode `m1` est **plus spécifique** qu'une autre définition `m2` si tout appel valide à `m1` serait également un appel valide à `m2` **moyennant plus de conversions** »
- Dans l'exemple
  - la déf 1 est plus spécifique que la déf 3 car tout appel à def 1 serait valide pour def 3 moyennant la conversion de l'`int` (premier paramètre) en `long`
  - c'est donc la déf 1 qui sera utilisée pour l'appel `C.comp(x,y)`

Overloading (surcharge)

- S'il ne peut trouver la définition la plus spécifique, le compilateur signale une erreur
- Exemple
  - pour l'appel `C.comp(x,x)`, il n'y a pas de définition plus spécifique parmi les 3 : def 1 est plus spécifique que def 3; def 2 est plus spécifique que def 3; mais, de def 1 et def 2, laquelle est plus spécifique? Aucune !
- Le programmeur peut résoudre une telle ambiguïté en utilisant une *conversion explicite* de type, aka un cast
- Exemple
  - En écrivant `C.comp((long) x, x)`, le programmeur indique que c'est la def 2 qui doit être choisie

Overloading (surcharge)

- L'overloading ainsi que la règle de la définition la plus spécifique s'appliquent tant aux paramètres de types primitifs qu'aux types d'objets
- Exemple, étant données les définitions dans la classe `C` :

```
static void foo(T a, int x) // def 1

static void foo(S b, long y) // def 2
```

avec `S` sous-type de `T`, l'appel `C.foo(e,3)`, où `e` est de type (apparent!) `S`, est ambigu.

- Exercice : comment lever cette ambiguïté ?

Dispatching

- Idée: garantir que quand une méthode est appelée sur un objet donné (`objetDonné.méthode(...)`), c'est le code fourni pour cette méthode et cet objet (type effectif) qui est exécuté
- Exemple
  - `String` fournit une méthode `boolean equals(Object o)` qui renvoie `true` quand deux `Strings` (la `String` courante et le paramètre) ont le même contenu (indépendamment de leur adresse dans le heap)
    - Attention, ce n'est pas la même chose que ==
  - `Object` aussi fournit une méthode `boolean equals(Object o)` mais qui renvoie `true` quand l'`Object` courant et le paramètre sont un seul et même `Object` (même adresse dans le heap)
    - Le résultat est, pour `Object`, le même que ==

Dispatching

- Or, il se peut qu'à la compilation, on ne puisse déterminer quel code appeler car seul le type apparent est connu du compilateur
- Exemple

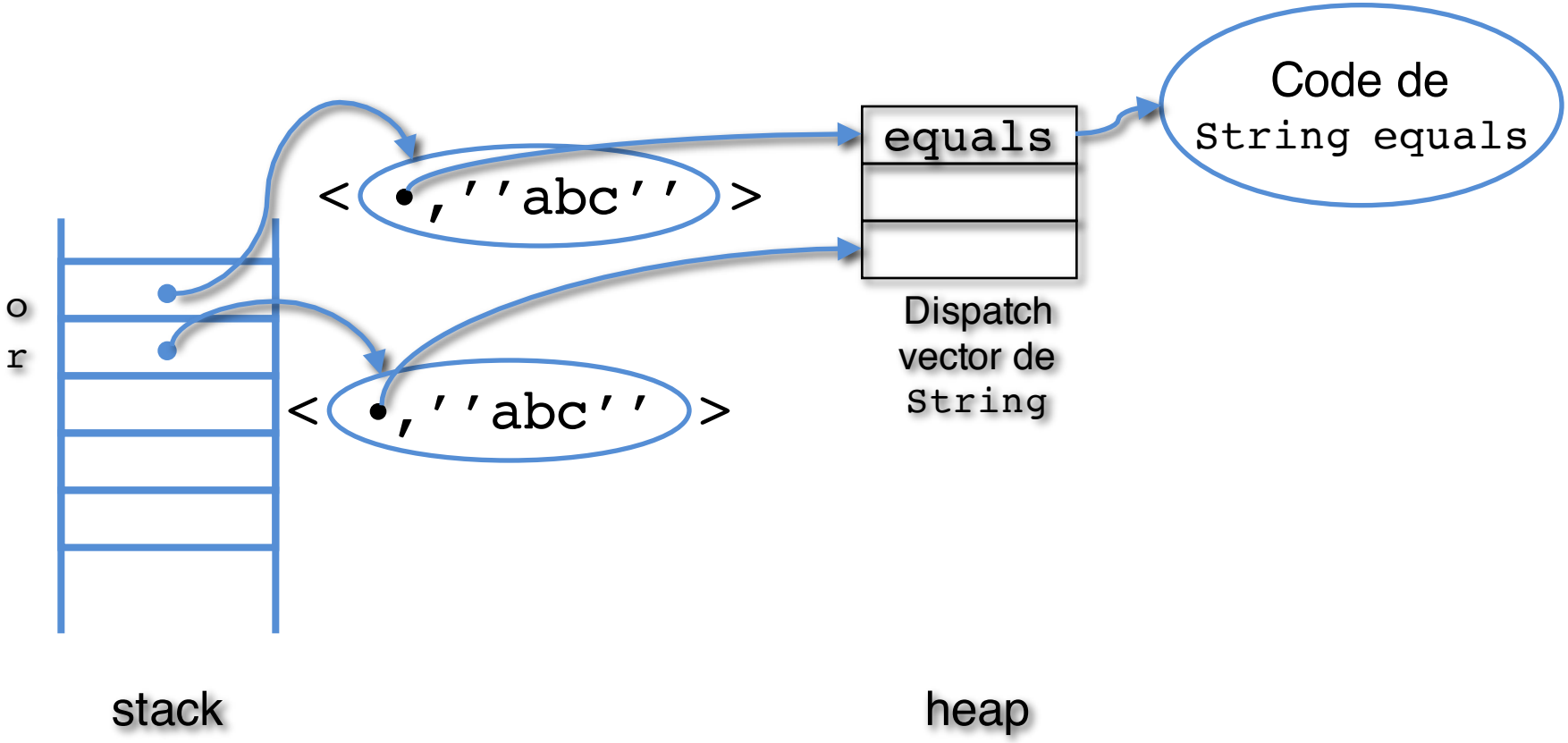
```
String t = "ab";
Object o = t + "c"; //concaténation
String r = "abc";
boolean b = o.equals(r);
```

- Si le type apparent était utilisé pour déterminer le code à appeler, `b` contiendrait `false`
- Il est donc nécessaire d'avoir un mécanisme **à l'exécution** qui fasse le dispatching vers le code souhaité (ici, celui du type effectif de `o`, c-à-d `String`).

Dispatching

- Chaque objet (dans le heap) contient une référence vers un **dispatch vector**
- Celui-ci contient une entrée pour chaque méthode d'un type d'objet donné
- Un dispatch vector n'est associé à un objet qu'à l'exécution en fonction de son type effectif
- Le compilateur ne fait que générer le code qui accède à l'entrée de la méthode dans le dispatch vector et qui se branche à l'endroit où le dispatch vector dit que le code de la méthode se trouve

Dispatching



« Types d'objets primitifs »

- Les types primitifs (`int`, `char`,...) ne sont pas des sous-types d'`Object`
- Ils ne peuvent donc être utilisés dans des contextes où on attend un `Object`
  - ce serait une violation du principe de substitution
- Ceci peut être résolu en utilisant des « **types d'objets primitifs** » ou **wrapper types**
- Pour chaque type primitif, il existe un type d'objet correspondant qui « emballe » sa valeur dans un objet (qui résidera donc dans le heap et pas dans le stack)
  - Le type d'objet `Integer` pour le type primitif `int`
  - Le type d'objet `Character` pour le type primitif `char`
  - etc...

« Types d'objets primitifs »

- Chaque type d'objet primitif fournit un constructeur qui permet d'emballer une valeur dans un objet
  - `public Integer(int x)`
- A l'inverse, chaque type d'objet primitif fournit une méthode qui renvoie la valeur contenue dans l'objet
  - `public int intValue( )`
- Idem pour les autres types d'objets primitifs e.g. `Character` possède les méthodes
  - `public Character(char x)`
  - `public char charValue( )`

« Types d'objets primitifs »

- On trouve également d'autres méthodes, par ex. celle qui construit un objet à partir de la lecture d'une `String`
  - `int n = Integer.parseInt(s);`
  - Si `s` pointe vers le `String` "`1024`", `n` prendra la valeur `1024`
- D'autres méthodes sont disponibles; elles sont décrites dans la documentation du package `java.lang` (dans lequel sont regroupés tous les « types d'objets primitifs »)
  - NB : `java.lang` est importé par défaut dans tout programme

Collections d'objets

- En Java, il existe une interface commune à toutes les collections d'objets: interface java.util.Collection
- C'est une interface, donc ne peut pas être directement instanciée
- Deux interfaces plus spécialisées héritent de `Collection` : `List` et `Set`

Interface `List`

- Collection ordonnée d'objets
- Peut contenir des doublons
  - doublon s'il existe o1, o2 appartenant à la List | o1 == o2 ou o1.equals(o2)
- C'est une interface donc ne peut pas être directement instanciée
- On utilisera habituellement le type de données `ArrayList` qui implémente l'interface `List`

Interface `Set`

- Correspond à la notion d'ensemble mathématique
- Ne peut pas contenir de doublon
  - pour tout o1, o2 qui appartiennent à l'ensemble | o1 ≠ o2 && !o1.equals(o2), et il y a au plus un élément null (mais ça n'est pas conseillé)
- C'est une interface, donc ne peut pas être directement instanciée
- Types de données implémentant l'interface `Set` : `HashSet` et `TreeSet`

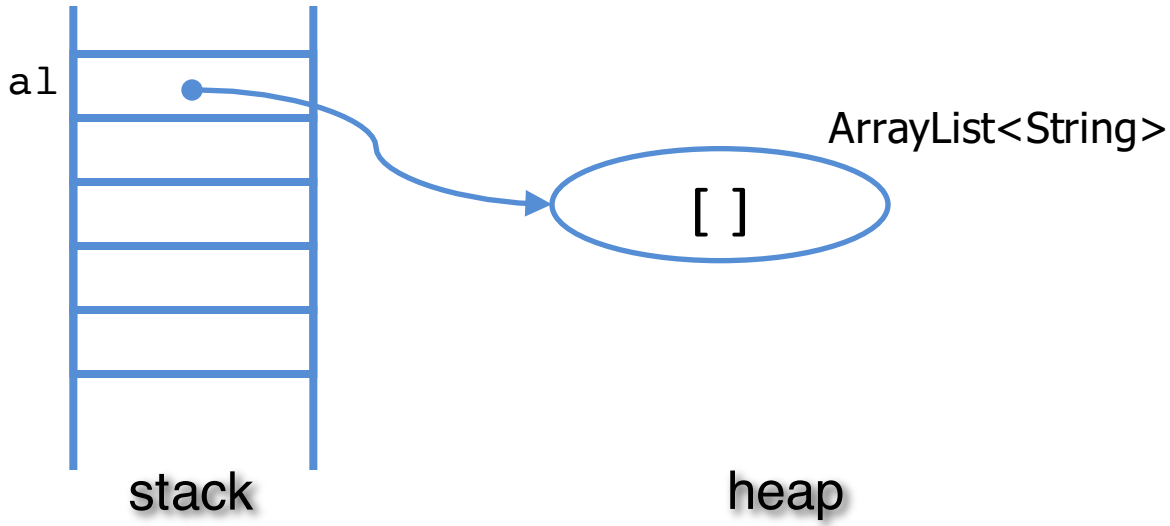
ArrayList

- Le type d'objet `ArrayList` est défini dans `java.util`
- Il sert à représenter des **tableaux d'objets de longueur variable**
  - il s'agit là de deux différences majeures par rapport aux tableaux classiques
- Les éléments d'un `ArrayList` sont numérotés de 0 à n-1 où n est la longueur courante de l'`ArrayList`
- On peut connaître la longueur courante d'un `ArrayList` en invoquant la méthode `int size()`
- Il est grandement recommandé de définir le type des éléments d'un `ArrayList` (cf. chapitre sur les Generics). Dès lors,
  - un `ArrayList` ne peut contenir des valeurs d'un type primitif, mais uniquement des (références vers des) objets (attention au mécanisme d'autoboxing)
  - un `ArrayList` ne devrait pas être constitué d'objets hétéroclites.

ArrayList

- **Créer** un nouvel `ArrayList` (de longueur 0) et destiné à contenir des objets de type `String` se fait via un `new`
- Exemple

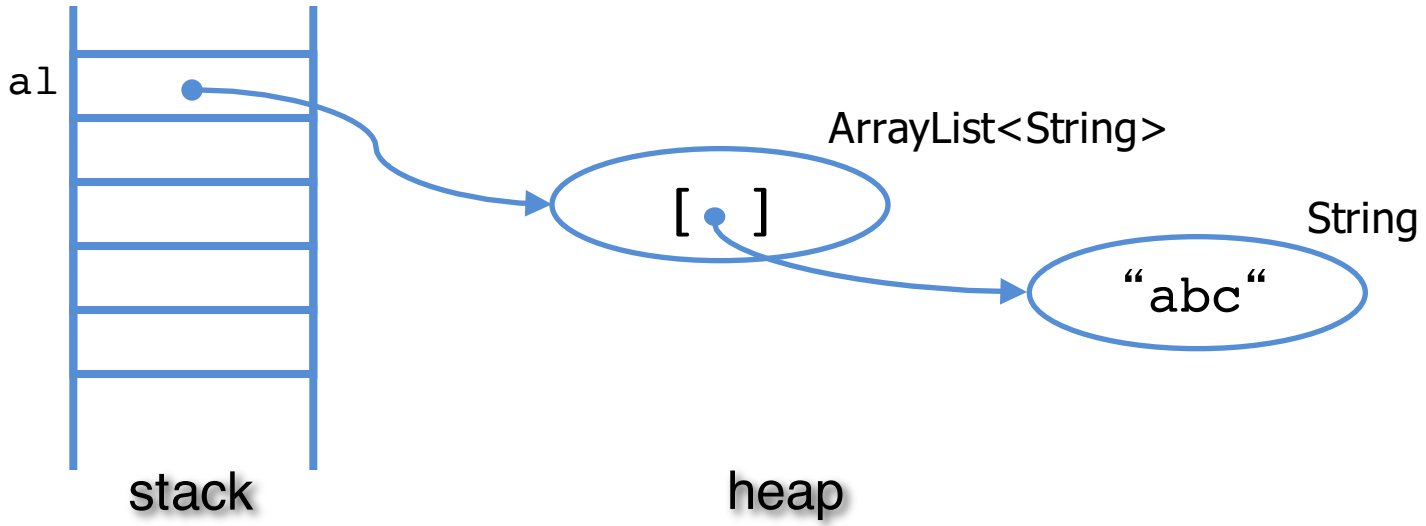
```
// crée une liste vide
   List<String> al = new ArrayList<>( );
   if (al.size() == 0 ) ...  // true, size vaut 0
```



ArrayList

- **Ajouter** un élément au contenu d'un `ArrayList<T>` (et donc augmenter sa taille) se fait via la méthode boolean `add(<T> o)`
- Exemple (suite de l'exemple du slide précédent)

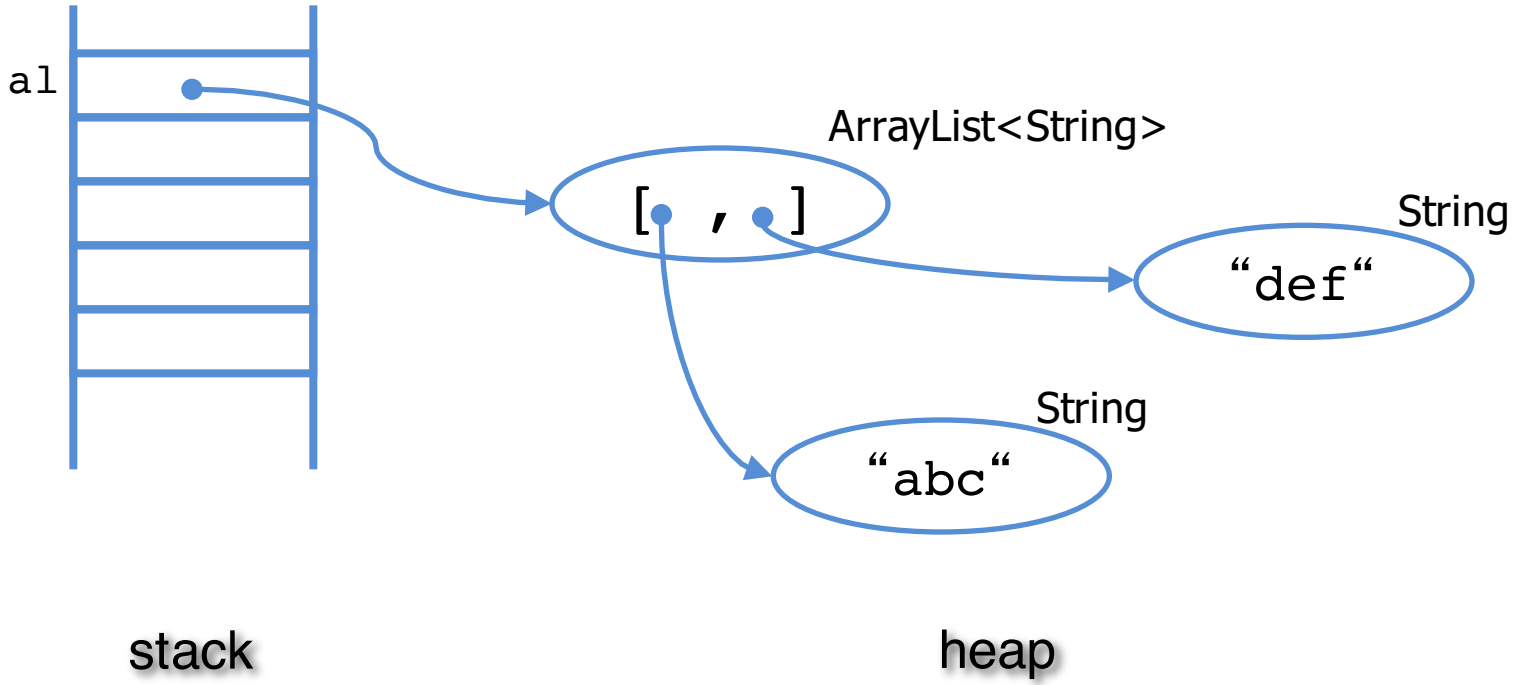
```
al.add("abc"); // ajoute une String "abc" dans al
if (al.size( ) == 0 ) ...  // false, al.size() vaut 1
```



ArrayList

- Exemple (suite de l'exemple du slide précédent)

```
al.add("def");// ajoute une String "def" dans al
```



ArrayList

- On peut **accéder** au contenu d'un `ArrayList<T>` via la méthode

  `<T> get(int index)`

- Exemple

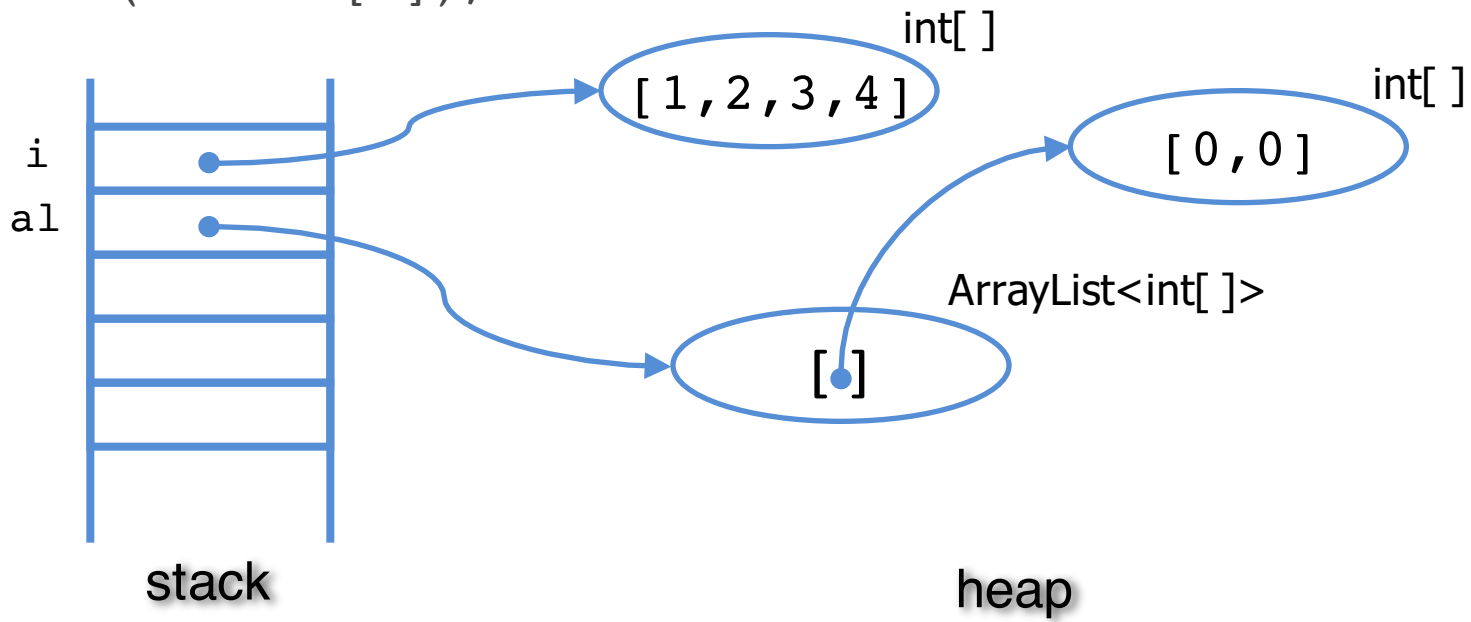
  ```
  String s = al.get(0)
  ```

- Remarques
  - Puisque l'`ArrayList` est typé pour contenir des objets de type `String` uniquement, la méthode `get(…)` renvoie directement une `String`
  - Si la valeur de l'index indiquée en paramètre dépasse la taille de l'`ArrayList`, la machine virtuelle renverra une `IndexOutOfBoundsException`
    - Exemple : `String t = (String) al.get(2)` est valide à la compilation mais échoue à l'exécution

ArrayList

- On peut **modifier le contenu** d'un `ArrayList<T>` **à un indice déterminé** via la méthode `<T> set(int i, <T> o)`
- Exemple

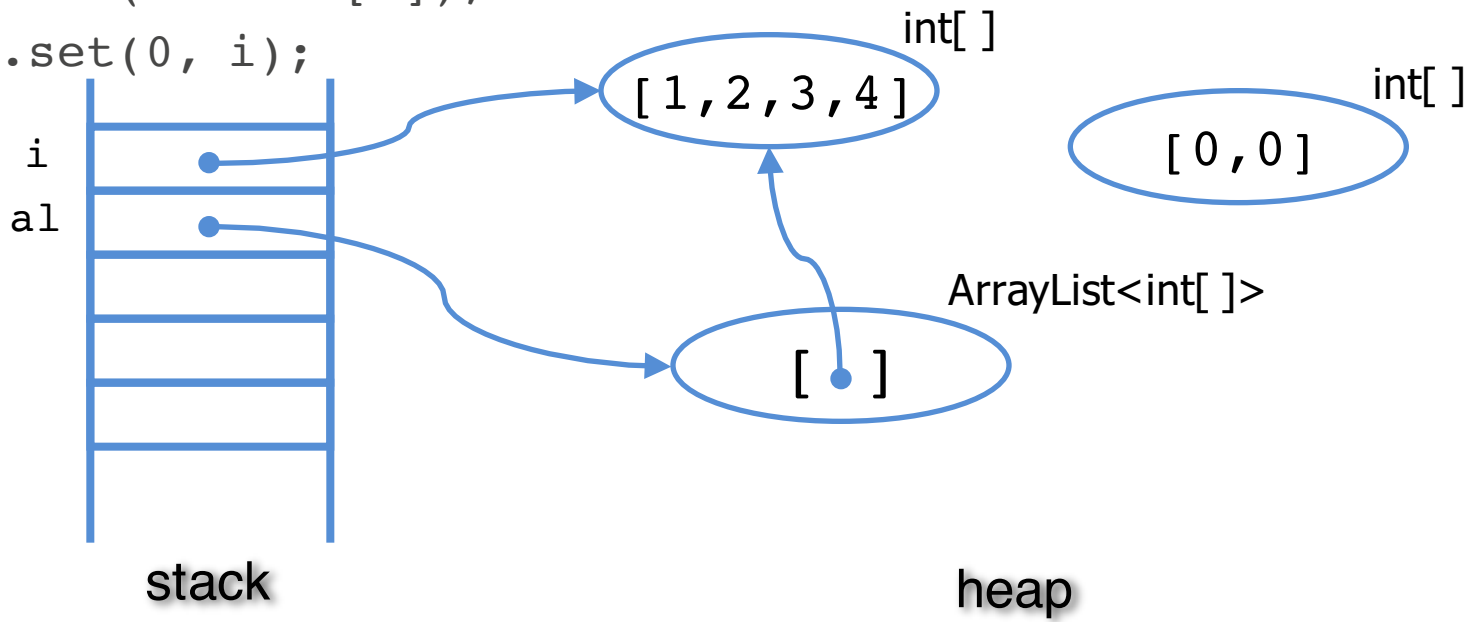
```
int[] i = {1,2,3,4};
List<int[]> al = new Arraylist<>();
al.add(new int[2]);
```



ArrayList

- On peut **modifier le contenu** d'un `ArrayList<T>` **à un indice déterminé** via la méthode `<T> set(int i, <T> o)`
- Exemple

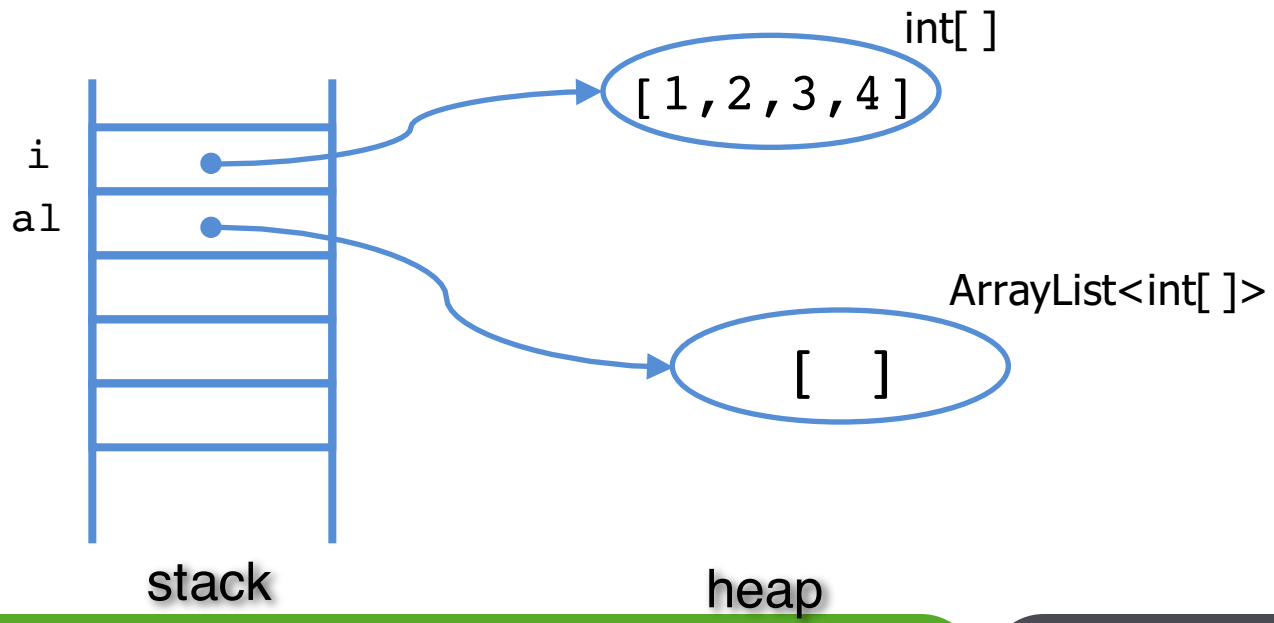
```
int[] i = {1,2,3,4};
List<int[]> al = new Arraylist<>();
al.add(new int[2]);
al.set(0, i);
```



ArrayList

- On peut également effacer le contenu d'un `ArrayList<T>` à un indice donné via la méthode `<T> remove(int i)`
- NB : Tant `set` que `remove` sont susceptibles de générer des `IndexOutOfBoundsException` à l'exécution
- Exemple (suite de l'exemple du slide précédent)

```
al.remove(0);
```



AutoBoxing

- Les collections ne peuvent contenir que des instances de types de données (donc pas de valeurs d'un type primitif).
- Java fournit le mécanisme d'autoboxing pour automatiquement « emballer » un type primitif dans un « type d'objet primitif »
- Exemple

```
List<Integer> al = new ArrayList<>( );
al.add(3); // instruction valide
```

est équivalent à

```
List<Integer> al = new ArrayList<>( );
al.add(new Integer(3));
```

Unboxing

- Java fournit le mécanisme d'unboxing pour automatiquement « transformer » un type d'objet primitif dans un type primitif
- Exemple (suite de l'exemple du slide précédent)

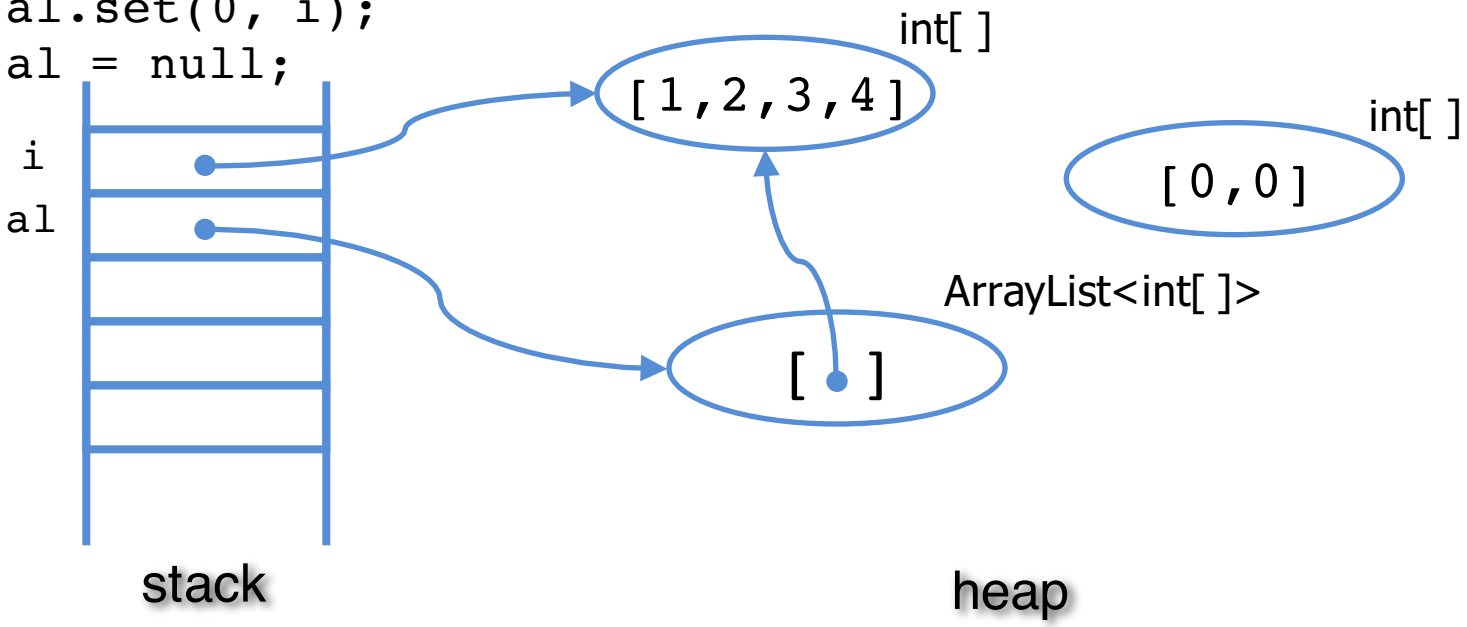
```
int x = al.get(0);
```

- Il n'est pas nécessaire de caster l'objet renvoyé par la méthode `get()`

ArrayList

- Tiens, et si on exécute `al=null` maintenant et qu'ensuite le garbage collector se déclenche, qu'emportera-t-il ?

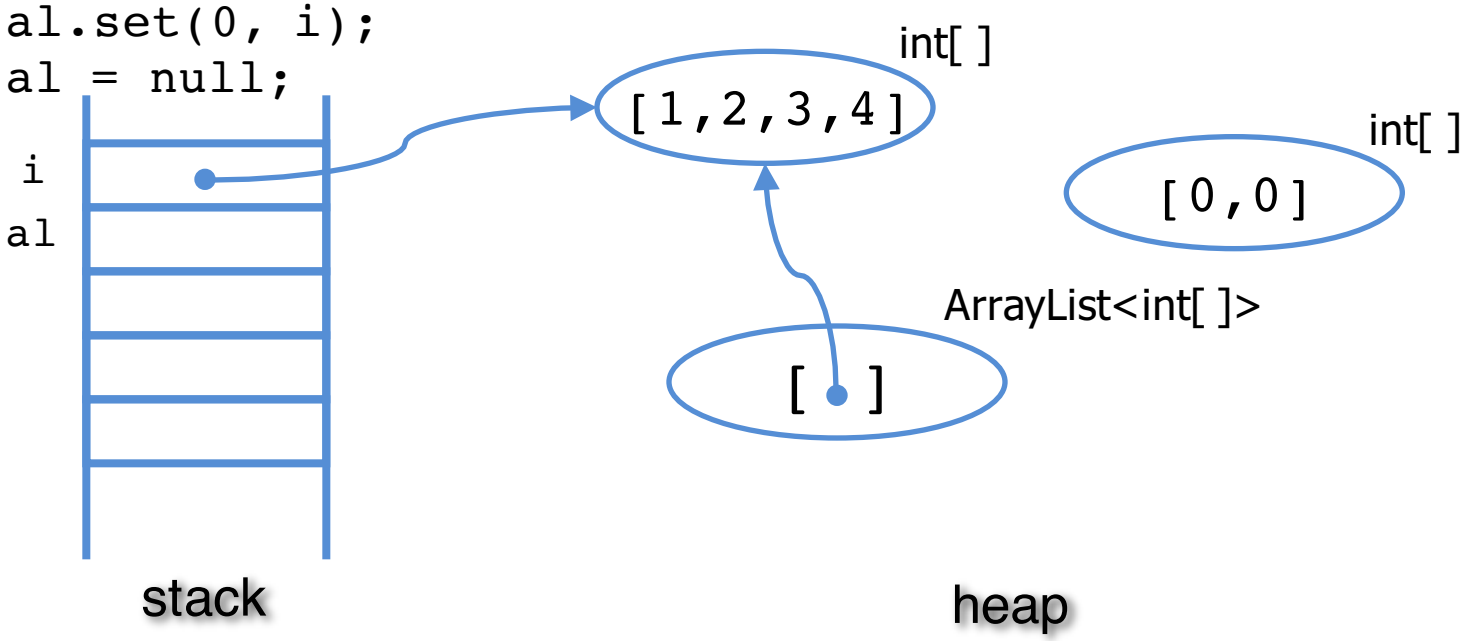
```
int[] i = {1,2,3,4};
List<int[]> al = new Arraylist<>();
al.add(new int[2]);
al.set(0, i);
al = null;
```



ArrayList

- Tiens, et si on exécute `al=null` maintenant et qu'ensuite le garbage collector se déclenche, qu'emportera-t-il ?

```
int[] i = {1,2,3,4};
List<int[]> al = new Arraylist<>();
al.add(new int[2]);
al.set(0, i);
al = null;
```



ArrayList

- Tiens, et si on exécute `al=null` maintenant et qu'ensuite le garbage collector se déclenche, qu'emportera-t-il ?

```
int[] i = {1,2,3,4};
List<int[]> al = new Arraylist<>();
al.set(0, i);
al = null;
```