Séquence 5

La mémoire et l'exécution de programme Java

1 Introduction

Un ordinateur dispose d'une mémoire importante. Cette mémoire est gérée par le système d'exploitation qui alloue une portion de la mémoire à chaque programme en cours d'exécution.

À un moment donné, il y a plusieurs programmes dont l'exécution est commencée et pas encore terminée. Par exemple, si vous êtes en train de travailler pour NFA032, vous pouvez avoir en cours d'exécution le programme Eclipse (environnement de programmation) et un navigateur qui vous permet de consulter le cours. De plus, vous avez tous les programmes du système d'exlpoitation qui assurent des tâches de service (par exemple, un programme système qui gère le réseau, un autre qui gère les fenêtres de l'écran, etc). Chaque programme utilise un morceau de la mémoire qu'il est le seul à utiliser. Lui seul a le droit de lire et d'écrire dans cette mémoire. On appellera cette mémoire la *mémoire privée du programme*.

Lorsqu'on demande à exécuter un programme Java, une portion de la mémoire de l'ordinateur va être allouée comme mémoire privée pour l'interpréteur Java, pour qu'il puisse exécuter le programme. Cette mémoire va contenir à la fois les instructions et les données du programme.

Nous allons présenter une version simplifiée de ce qui se passe dans la mémoire. Cela permet de comprendre les principes de fonctionnement et les conséquences sur les résultats produits.

2 Structure de la mémoire

La mémoire privée du programme est divisée en trois parties distinctes qui ont chacune un rôle et un mode de fonctionnement propre.

— La mémoire des classes qui contient notamment le code des méthodes, les instructions du programme.

— La mémoire des méthodes : nous l'appellerons la *pile* (*stack* en anglais). Elle contient les paramètres et variables des méthodes en cours d'exécution.

— La mémoire des objets : nous l'appellerons le *tas* (*heap* en anglais). Elle contient les objets et les tableaux.

À la fin de l'exécution du programme, toute la mémoire privée est rendue au système d'exploitation. Si on redemande à exécuter le programme, rien ne garantit que la mémoire privée allouée par le système sera la même que la première fois et de toute façon, aucune information ne sera gardée de

l'ancienne exécution. Tout se passe comme si la mémoire était effacée à chaque fois qu'elle est rendue au système d'exploitation.

3 Mémoire des classes

La mémoire des classes contient le code des méthodes et des constructeurs de chaque classe utilisée par le programme. Elle ne contient pas les variables d'instances de la classe : celles-ci sont dans la mémoire des objets (le tas).

L'accès à cette mémoire se fait au moyen du nom de la classe. Puis, à l'intérieur de la classe, en utilisant le nom de la méthode à exécuter.

Au début de l'exécution du programme, la mémoire contient un certain nombre de classes utilisées par l'interpréteur plus la classe contenant le `main` qu'il s'agit d'exécuter. Ces classes sont lues dans les fichiers .class et copiées en mémoire par un composant de l'interpréteur Java qui s'appelle le *chargeur de classe* ou *class loader*.

Ensuite, les instructions du programme sont exécutées. Si une instruction utilise une classe qui n'est pas encore dans la mémoire, celle-ci est chargée par le chargeur de classe. Cela se produit donc au cours de l'exécution du programme.

Une fois chargée en mémoire, une classe y reste jusqu'à la fin de l'exécution du programme.

4 Exemple de chargement des classes

Nous allons voir comment les classes sont chargées au fil de l'exécution du programme en prenant un exemple.

```
import java.util.Date;
class Un{
    public static void m(){}
}
public class ExempleLoad{
    public static void main(String[] args){
        System.out.println("Début␣main");
        Un.m();
        System.out.println("main␣avant␣déclaration");
        Date unedate;
        System.out.println("main␣avant␣instantiation");
        unedate = new Date();
        System.out.println("Fin␣main");
    }
}
```

Dans cet exemple, on intercale des affichage avec des instructions dont certaines utilisent des classes. Par exemple `Un.m()` demande l'exécution de la méthode statique `m` de la classe `Un`.

Pour voir quand une classe est chargée, nous allons demander l'exécution du programme dans une fenêtre terminal avec une option qui permet d'afficher les classes chargées. Cela se fait avec la commande suivante :

```
java -verbose:class ExempleLoad
```

Le résultat de l'exécution est un affichage de 421 lignes dont 416 concernent le chargement d'une classe. Il y a donc 416 classes dans la mémoire des classes à la fin de l'exécution du programme.

Nous allons nous concentrer sur les dernières lignes de l'affichage.

```
[Loaded ExempleLoad from file:/home/barthe/]
[Loaded sun.launcher.LauncherHelper$FXHelper from /usr/lib64/jvm/jre/lib/rt.jar
[Loaded java.lang.Class$MethodArray from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded java.lang.Void from /usr/lib64/jvm/jre/lib/rt.jar]
Début main
[Loaded Un from file:/home/barthe/]
main avant déclaration
main avant instantiation
[Loaded java.util.Date from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded sun.util.calendar.CalendarSystem from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded sun.util.calendar.AbstractCalendar from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded sun.util.calendar.BaseCalendar from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded sun.util.calendar.Gregorian from /usr/lib64/jvm/jre/lib/rt.jar]
Fin main
[Loaded java.lang.Shutdown from /usr/lib64/jvm/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from /usr/lib64/jvm/jre/lib/rt.jar]
```

Nous avons commencé cet extrait au moment où la classe `ExempleLoad`, la classe contenant le `main` du programme, est chargée. Il y a ensuite encore trois classes auxiliaires qui sont chargées (sun.launcher.LauncherHelper$FXHelper, java.lang.Class$MethodArray et java.lang.Void), puis la première instruction du main est exécutée. C'est le premier affichage.

Ensuite, pour la deuxième instruction, on voit que la classe Un est chargée. Cette classe est donc chargée en mémoire au moment d'exécuter `Un.m()`. Le code de la méthode `m` va être lu dans la mémoire des classes et exécuté.

Ensuite, dans l'affichage, on voit l'effet du deuxième `System.out.println`. La déclaration de la variable `unedate` ne provoque pas le chargement de la classe, car il n'y a pas de code à exécuter. En revanche l'instanciation avec `new` demande l'exécution d'un constructeur de la classe `Date`, donc là, il faut d'abord charger la classe, puis aller lire dans la mémoire des classes les instructions du constructeur qu'il faut exécuter.

On voit que la classe Date et/ou le constructeur utilise quatre autres classes qui sont chargée au même moment : `CalendarSystem, AbstractCalendar, BaseCalendar` et `Gregorian`.

Le dernier affichage est exécuté et deux dernières classes sont chargée pour finir l'exécution du programme : `Shutdown` et `Shutdown$Lock`.

Notons que beaucoup de classes couramment utilisées font partie des classes chargées par l'interpréteur et font partie des lignes d'affichage que nous n'avons pas détaillées : c'est le cas de `System`, `String`, `ArrayList`, `Math`, etc.

5 Mémoire des méthodes ou pile

Les paramètres et variables d'une méthode doivent être stockés dans la mémoire. L'exécution d'une méthode commence toujours par le calcul de la valeur des paramètres et l'allocation d'une portion de la pile qui sera réservée à cette exécution de la méthode. On dira qu'il s'agit de la mémoire privée de cette exécution de la méthode.

À un moment donné du programme, plusieurs méthodes sont en cours d'exéuction : par exemple, à la première ligne du main du programme `ExempleLoad`, il y a appel à la méthode `System.out.println`.

Pendant que cette méthode s'exécute, l'exécution du `main` est commencée et pas encore terminée. Elle est en cours. Il y a donc deux méthodes en cours d'exécution : `main` et `System.out.println`. Il y a à ce moment-là, deux mémoires privées dans la pile.

À la fin de l'exécution d'une méthode, la valeur qu'elle a calculée est renvoyée et la mémoire privée de cette exécution est libérée, rendue à l'interpréteur Java qui pourra l'utiliser pour l'exécution d'une autre méthode (ou même une autre exécution de la même méthode).

L'accès à un élément de la pile se fait toujours dans la dernière mémoire privée créée. Au sein de cette mémoire privée, l'accès se fait en utilisant le nom du paramètre ou de la variable.

6 Mémoire des objets ou tas

Le tas contient les objets et les tableaux. Ils ont en commun d'être créés par une instruction `new`. L'exécution de cette instruction consiste à allouer une mémoire privée pour l'objet ou le tableau. Cette mémoire contient les variables d'instance et le type de l'objet et dans le cas d'un tableau, les cases de ce tableau.

L'accès à la mémoire privée d'un objet ou d'un tableau se fait au moyen d'une adresse : c'est ce qu'on appelle en Java une référence.

Les objets et tableaux apparaissent dans la mémoire lors de l'exécution d'instruction `new` figurant dans le programme ou dans les librairies appelées par le programme. On peut considérer qu'ils restent en mémoire jusqu'à la fin de l'exécution du programme. En réalité, l'interpréteur Java peut supprimer de la mémoire les objets et tableaux dont il a la certitude qu'ils ne serviront plus. Du point de vue du comportement du programme lors de l'exécution, il n'y a aucune différence entre le cas où l'objet reste en mémoire et celui où il est détruit. Le programmeur peut donc ne pas tenir compte de cela.

L'exécution d'une instruction new comporte plusieurs étapes. Dans le cas d'un objet ces étapes sont :

- Calcul de la valeur des paramètres du `new` (s'il y en a).
- Allocation d'une mémoire privée dans le tas d'une taille suffisante pour y stocker les variables d'instances et le type (classe d'instance) de l'objet.
- Exécution du constructeur. Cela initialise les variables d'instance dans la mémoire privée de l'objet.
- L'instruction `new` renvoie l'adresse de la mémoire privée de l'objet dans le tas. Cette adresse s'appelle une référence.

Si le new apparaît dans une affectation :

```
var = new ArrayList<String>();
```

Une mémoire privée est crée pour le nouvel objet ArrayList avec dedans les variables d'instance, le type ArrayList<String> et l'adresse de cette mémoire est renvoyée. C'est donc l'adresse de l'objet qui est stockée dans la variable `var`.

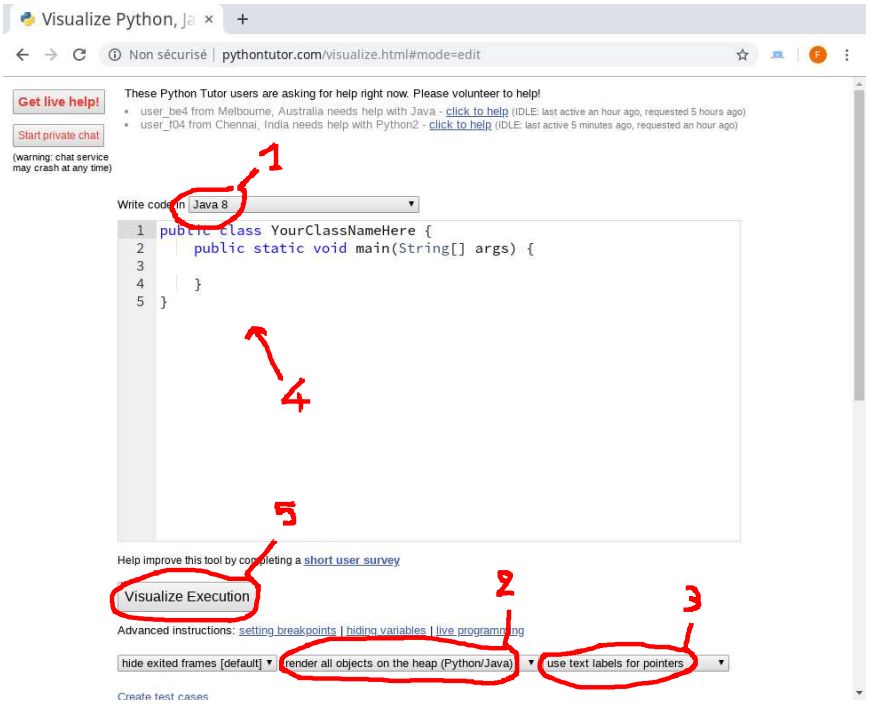
Pour un tableau, les étapes sont les suivantes :

- Allocation d'une mémoire privée d'une taille suffisante pour y stocker le type du tableau et les cases du tableau.
- Initialisation des cases avec la valeur par défaut du type (par exemple 0 pour un tableau d'entier, `null` pour un tableau contenant des objets).
- L'instruction `new` renvoie l'adresse de la mémoire privée de du tableau dans le tas (référence).

7 Pythontutor : voir la mémoire

On ne peut pas voir facilement le contenu de la mémoire au cours de l'exécution d'un programme. Il existe néanmoins un outil qui permet de visualiser le tas et la pile lors de l'exécution d'un petit programme : il s'agit de Pythontutor. C'est un outil pédagogique développé à l'université de Waterloo (Canada).

On peut l'utiliser via le site à l'adresse : `http://pythontutor.com/`.

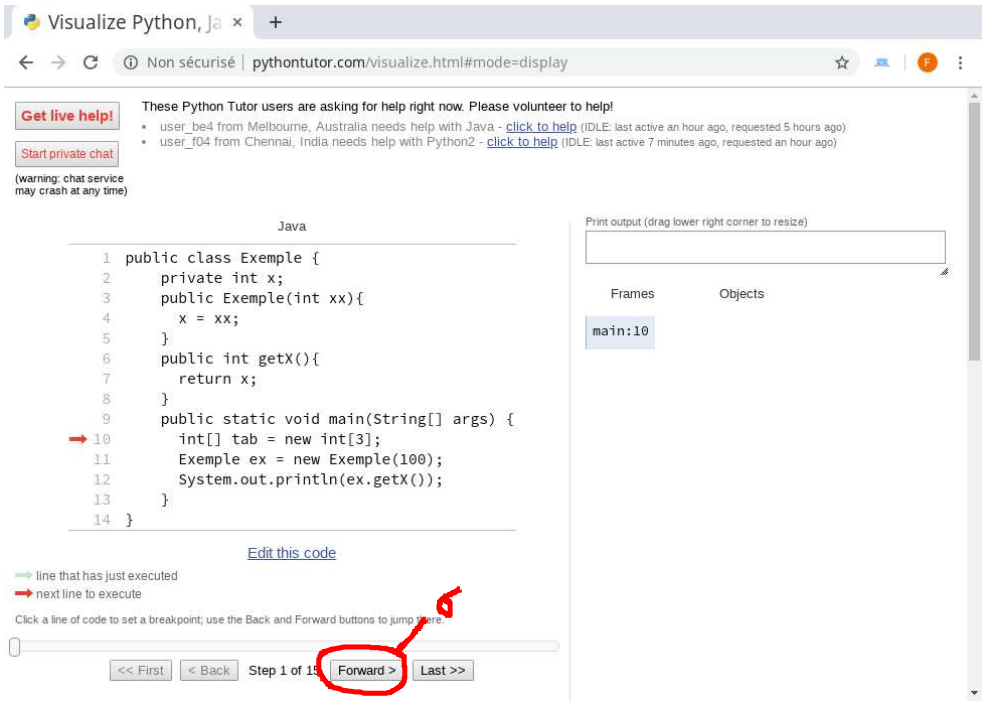


Il faut aller à la page de visualisation, choisir le langage java 8 (repère 1). Pour commencer, nous allons sélectionner les options : *render all objects on the heap* (repère 2) et *use text labels for pointers* (repère 3). Puis taper ou copier/coller le programme dans la zone de texte (repère 4 sur la figure).

Nous allons tester l'outil sur une exemple simple :

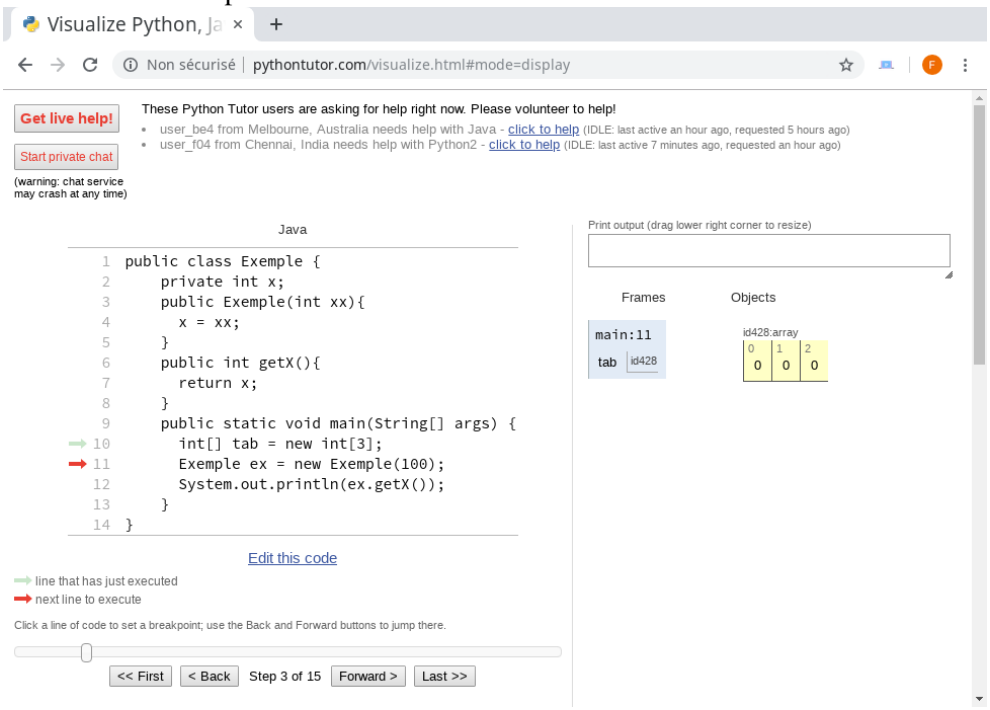
```
public class Exemple {
    private int x;
    public Exemple(int xx){
      x = xx;
    }
    public int getX(){
      return x;
    }
    public static void main(String[] args) {
      int[] tab = new int[3];
      Exemple ex = new Exemple(100);
      System.out.println(ex.getX());
    }
}
```

Une fois le programme entré, il faut cliquer sur le bouton `Visualize Execution` (repère 5). La visualisation commence avec l'état initial suivant.



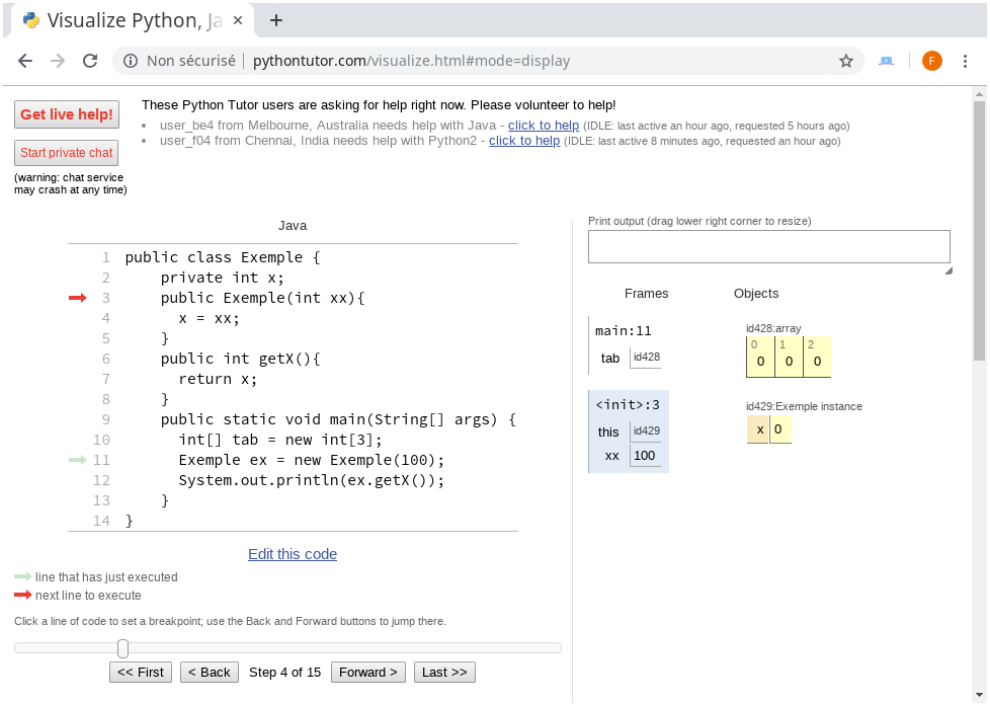
La pile est appelée ici *frames* et le tas est appelé *objects*. Dans cette représentation, le tas est initialement vide et la pile contient une mémoire privée pour l'exécution de la méthode `main`. Cette mémoire privée est vide.

On peut exécuter la première instruction du main en cliquant sur le bouton *forward* (repère 6) sous le programme. On va même le faire deux fois pour faire apparaître le tableau dans le tas et la référence du tableau dans la pile, associée au nom tab. La référence affichée ici est : `id428`. On voit cette référence dans la pile comme valeur de `tab` et dans le tas comme adresse du tableau.

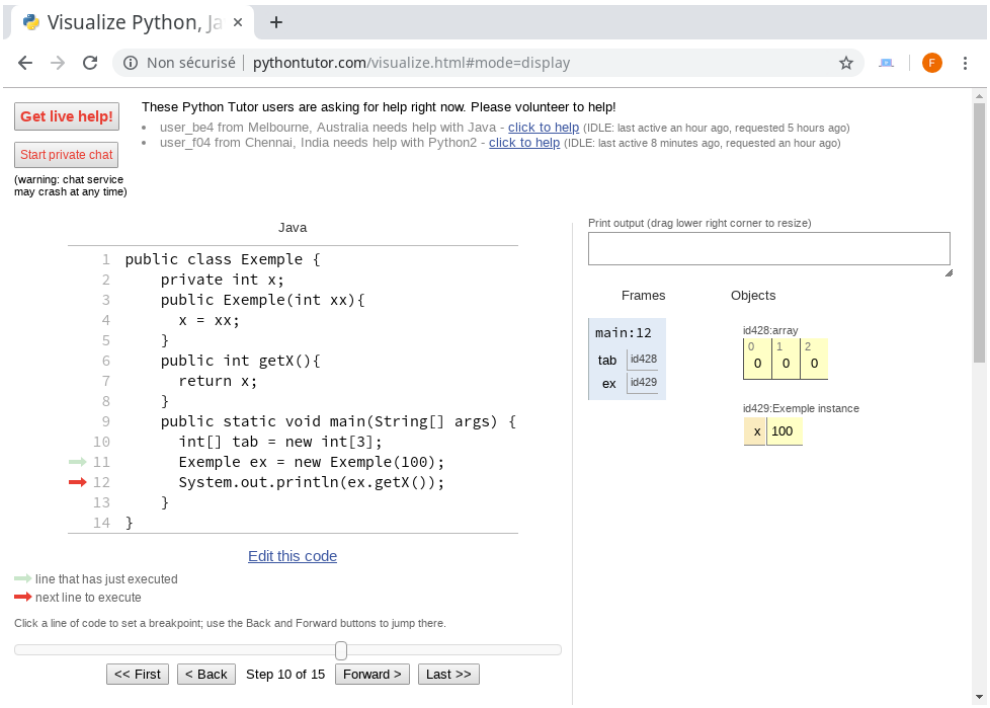


Encore un appui sur forward et on voit apparaître un objet instance de `Exemple` créé dans le tas

(effet du `new`) et la création dans la pile d'une mémoire privée pour le constructeur. Cette mémoire notée `<init>` contient la valeur du paramètre `xx`.



Avançons encore dans l'exécution du programme avec plusieurs forward successifs et on atteint l'état final du programme avec deux variables dans la mémoire privée de `main`, un tableau et un objet dans le tas.

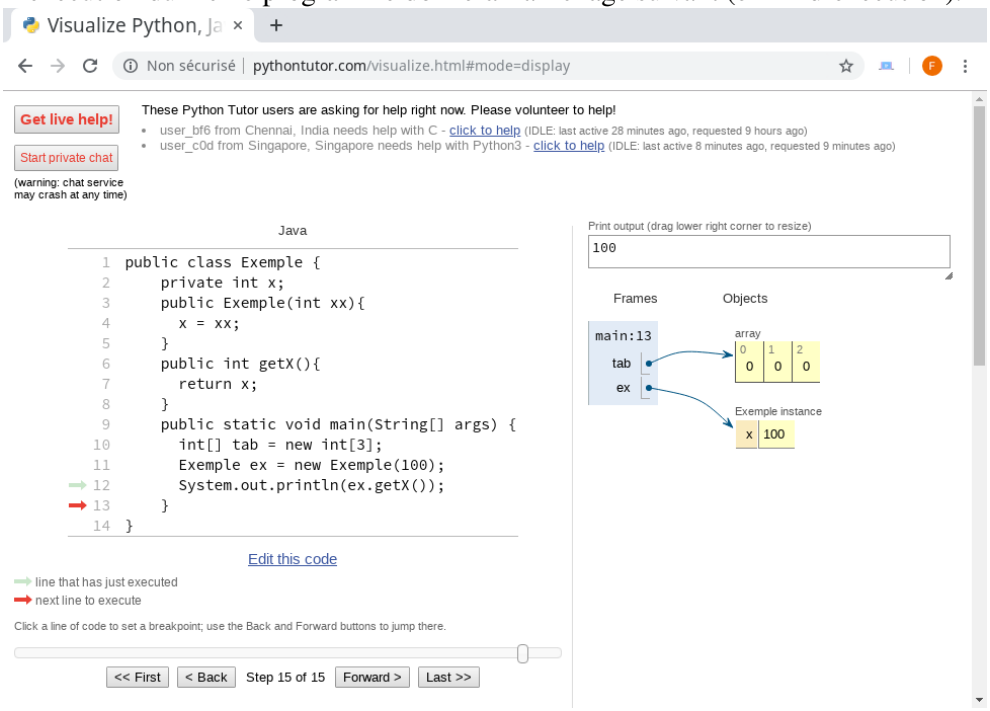


8 Autre type de visualisation

Avec un autre paramètre, on peut réprésenter une référence non pas avec un identificateur qu'on retrouve dans la pile et dans le tas mais sous la forme d'une flèche qui part de la pile et arrive dans le tas.

Il faut remplace l'option *use text labels for pointers* (repère 3 de la figure) par *draw pointers as arrows*.

L'exécution du même programme donnera l'affichage suivant (en fin d'exécution).



Cette représentation par flèches est assez agréable, sauf lorsque sur les exemples complexes, les flèches se croisent et qu'on ne distingue plus forcément bien où elles commencent et où elles se terminent.