je suis en train de taper le TP... la première partie est finie.

De l'autre côté du miroir

Comprendre les mécanismes qui régissent java, c'est faire un grand pas (en avant) dans la maîtrise de la programmation.

> Mieux vaut comprendre qu'apprendre! (Gustave Le Bon)

Il est vraiment important d'avoir une bonne représentation de ce qui se passe “sous le capot”, autrement dit savoir ce que java (ou tout autre outil d'ailleurs) fait réellement, au plus bas niveau.

Certains langage, comme le C par exemple (ou encore à un plus bas niveau: l'assembleur), ne masquent pas la complexité de la gestion de la mémoire. C'est à la fois plus difficile à prendre en main, car il faut maîtriser la gestion de la mémoire, mais aussi plus proche de la réalité et donc de la compréhension de ce qu'est un programme.

Java, comme de nombreux langages dits “de haut niveau”, masquent cette complexité au programmeur. Pourtant, les mêmes opérations sont réalisées.

Je vais donc vous montrer ce que java fait pour vous, et comment il fonctionne, afin que vous puissiez en apprécier les avantages et surtout déjouer certains pièges dans lesquels une mauvaise connaissance des mécanismes de bas niveaux peuvent vous précipiter sans que vous ne vous en rendiez compte.

La mémoire, les types

Tout programme utilise de la mémoire:

1. pour stocker des instructions à exécuter
2. pour sauvegarder des données et des informations

Une variable est une donnée. Pour être utilisée dans un programme, elle doit disposer d'un espace de stockage en mémoire, et d'une adresse pour cet emplacement mémoire qui permettra au programme d'y accéder.

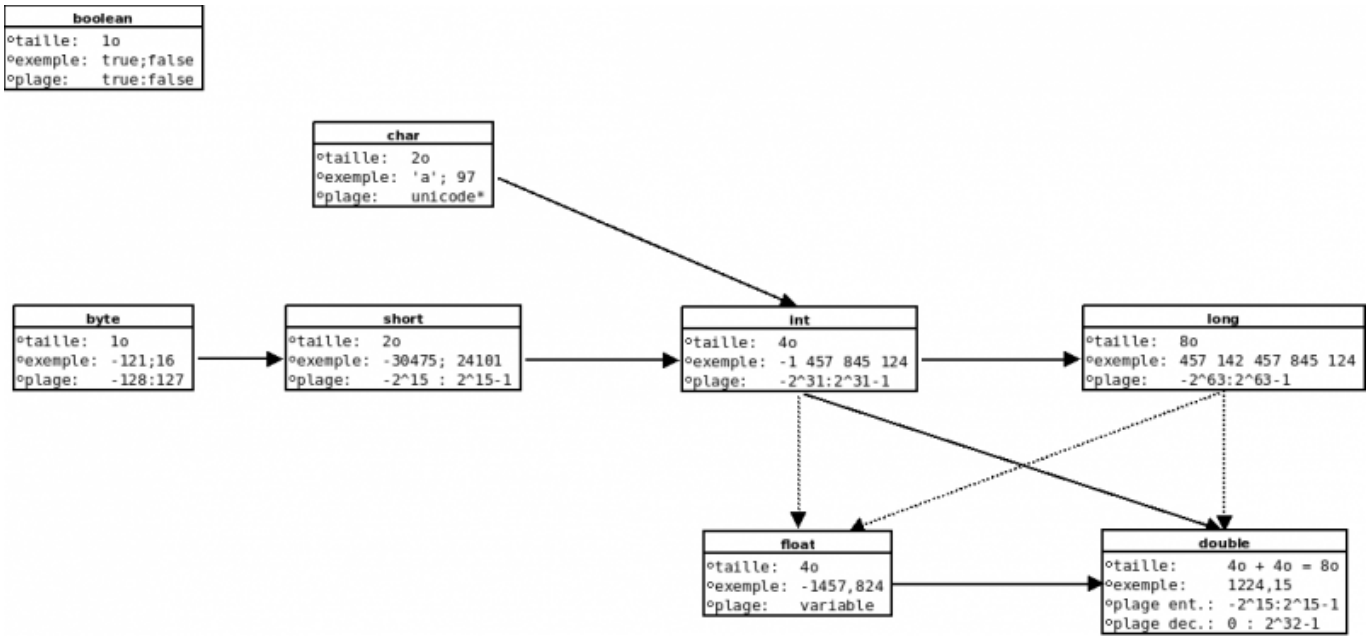
Il existe deux principaux types de variable, dont vous avez déjà entendu parler:

1. les variables de type primitif
2. les variables de type pointeur (dont font partie les types construits et les tableaux en java).

Les types primitifs

Les types primitifs ont une **taille fixe** en mémoire. Leur adresse est l'endroit de la mémoire où est stocké le premier octet des données de la variable.

Avant toute chose, rappelons quels sont ces types en java:



Chaque ligne de la figure correspond à un type de données: boolean, caractère, entier et réel. Chaque colonne à une taille en octet (1, 2, 4 et 8 octets). Les lignes pleines indiquent une conversion possible sans perte d'information, les lignes en pointillées des conversions avec perte d'information possible.

Toute autre conversion sera nécessairement avec perte: il faudra alors indiquer à java explicitement qu'on souhaite réaliser la conversion (c'est le fameux “cast” ou “transtypage”). Un double est en réalité la concaténation de 2 int: un pour la partie entière et le signe, l'autre pour la partie décimale. Ceci explique qu'il est facile de convertir un int en double.

Observons le code suivant:

```
    public static void types(){
        System.out.println("************ Conversions ************");
        byte b = -2;
        short s = 123;
        int i = 21456;
        long l = 3456123124L;//le L à la fin indique que c'est un long

        double d = 1452.4578;
        float f = 666.666666F;//le F à la fin indique que c'est un float


        char c = 'B';

        boolean bool = true;

System.out.println("données:"+b+";"+s+";"+i+";"+l+";"+d+";"+f+";"+c+";"+bool
);
        System.out.println(" 1) conversions sans pertes");

        s = b;
        i = s;
```

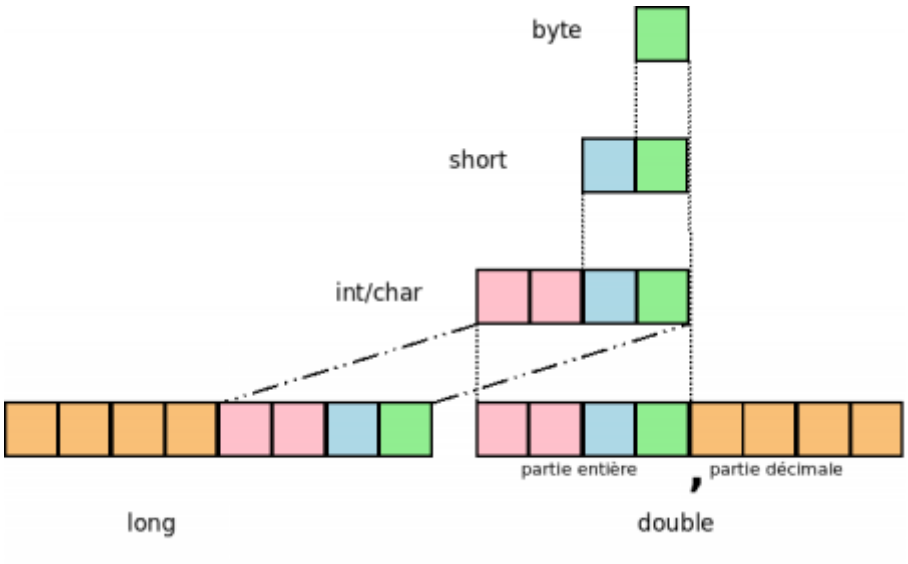
```
        l = i;
        d = i;
System.out.println("données:"+b+";"+s+";"+i+";"+l+";"+d+";"+f+";"+c+";"+bool
);
        l=c;
System.out.println("données:"+b+";"+s+";"+i+";"+l+";"+d+";"+f+";"+c+";"+bool
);
        /* error : incompatible type
        l=bool;
        */

        System.out.println(" 2) conversions automatiques avec pertes
possibles");
        f = i;//ça a l'air de fonctionner
        System.out.println(f+";"+i);
        i = 1456123124;
        f = i;//oups! ça ne fonctionne plus!!! Pourquoi?
        System.out.println(f+";"+i);

        System.out.println(" 3) conversions non autorisées par le
compilateur");
        //erreurs de type "possible loss of precision
        //l = d;
        //i = f;
        //erreurs de type "integer number too large"
        //i = 3456123124;
        System.out.println(" 4) conversions avec transtypage (cast)");
        l = 3456123124L;
        i = (int) l; //transtypage: plus d'erreur...
        System.out.println(i+";"+l);
        //... mais un résultat inexact!
        //... là ça marche
        l = -2147483648L;
        i = (int) l;
        System.out.println(i+";"+l);
        //et là ça marche plus, on est à la limite: pourquoi?
        l = -2147483649L;
        i = (int) l;
        System.out.println(i+";"+l);
    }
```

et le résultat de son exécution en console:

```
********** Conversions **********
données:-2;123;21456;3456123124;1452.4578;666.6667;B;true
 1) conversions sans pertes
données:-2;-2;-2;-2;-2.0;666.6667;B;true
données:-2;-2;-2;66;-2.0;666.6667;B;true
 2) conversions automatiques avec pertes possibles
-2.0;-2
1.45612314E9;1456123124
```

```
 3) conversions non autorisées par le compilateur
 4) conversions avec transtypage (cast)
-838844172;3456123124
-2147483648;-2147483648
2147483647;-2147483649
```

Si on résonne en taille dans la mémoire, les problèmes de conversion sont faciles à comprendre: tant que la représentation en mémoire est compatible (int et double, mais pas int et float par exemple) et que la taille du contenu ne dépasse pas celle du contenant (int et long, mais pas l'inverse), il n'y aura pas de problème:



Pour finir, si vous vous rappelez bien que les entiers relatifs sont stockés avec la méthode du complément à 2, vous devriez vous dire qu'il y a un peu plus qu'une simple “recopie” quand il s'agit de conversion d'entiers négatifs.

Prenons le cas du byte -25.

25 en binaire s'écrit 0001 1001.

-25 en complément à deux s'écrira donc 1110 0111.

Si je le recopiais directement dans un short, j'aurais donc:

0000 0000 1110 0111.

Mais ça, c'est l'entier 231 (128+64+32+4+2+1), pas -25!

Pour éviter ce problème, lors d'une conversion, le bit de signe est recopié à gauche:

1110 0111 → **1111 1111** 1110 0111

0011 0101 → **0000 0000** 0011 0101

Mémoire des types primitifs

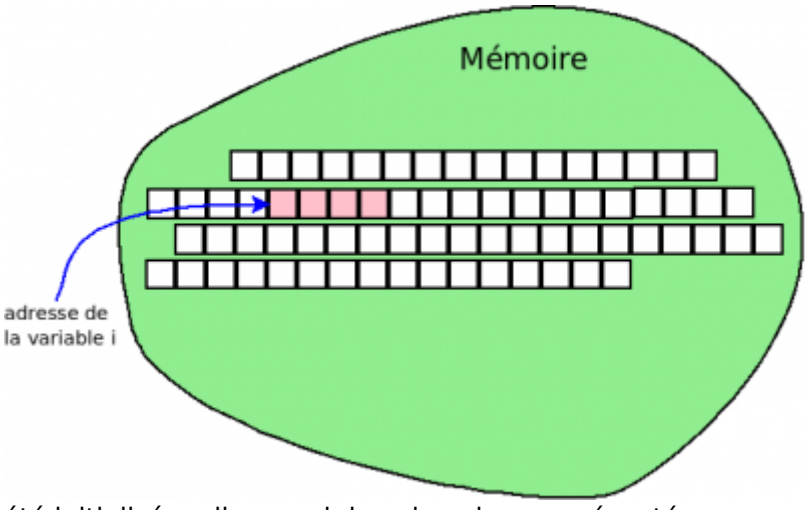
Comme **on connaît** la taille d'un type primitif, il suffit de connaître l'adresse du premier octet où il est stocké en mémoire pour accéder à son contenu.

Le programme lira autant d'octets que le type en possède.

Prenons le programme suivant, pas à pas:

```
int i;
```

Le programme réserve en mémoire un emplacement de 4 octets, et mémorise l'adresse du premier octet pour la variable i:



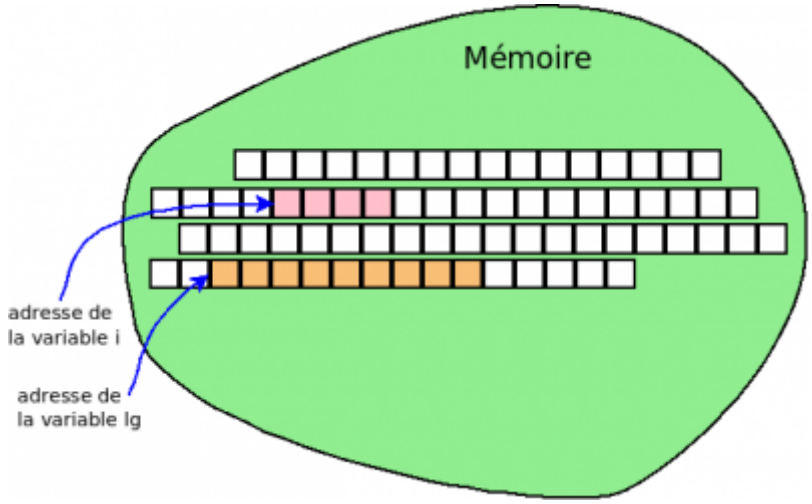
Notons que i n'a pas été initialisée: elle prend donc la valeur représentée par ce qu'il y avait dans les octets utilisés. **C'est pour cela que nous vous incitons fortement à initialiser toutes vos variables!**

Dans la pratique, il est fréquent que les langages de programmation affecte la valeur 0 par défaut en effaçant les données des octets.

```
long lg = -2147483649L;
```

Le programme réserve en mémoire un emplacement de 8 octets, et mémorise l'adresse du premier octet pour la variable lg. Notons au passage que -2147483649 s'écrit en binaire, avec le complément à 2 et sur 64 bits (8 octets):

1111 1111 1111 1111 1111 1111 1111 111**1 0**111 1111 1111 1111 1111 1111 1111 1111




```
i = (int)lg;
```

Cela revient à **copier** les quatre derniers octets de lg dans la zone mémoire de i.

i reçoit donc 111 1111 1111 1111 1111 1111 1111 1111.

i vaut donc 2147483647 (2^31 - 1).

Avec des types primitifs, une affectation est faite par *valeur* (on recopie la valeur, on ne change pas les adresses des variables en mémoire).

Les types pointeurs (types construits, tableaux)

Les types "pointeurs"

Un pointeur est **une adresse** dans la mémoire.

Si vous comprenez cela, vous avez tout compris!

Pour être plus facile à lire, cette adresse est souvent exprimée en hexadécimal. Regardons par exemple ce qui se passe quand fabrique des types construits:

```
public class TypeConstruit{
    /**
        Méthode qui affiche l'adresse mémoire de différents Types construits
en java
    */
    public static void adresseMemoire(){
        //création d'une instance d'Object
        Object o = new Object();
        System.out.println(o);

        //tableau d'entiers
        int[] tabInt = new int[10];
        System.out.println(tabInt);
```

```
        //tableau de double
        double[] tabDouble = new double[10];
        System.out.println(tabDouble);

        //tableau d'Object
        Object[] tabObject = new Object[10];
        System.out.println(tabObject);

        Object[] tabObjectCopie = tabObject;
        System.out.println(tabObjectCopie);
    }


    public static void main(String args[]){
        TypeConstruit.adresseMemoire();
    }
}
```

Voici l'affichage en console obtenu sur ma machine:

```
java.lang.Object@306f7492
[I@654e3615
[D@71c0d0a8
[Ljava.lang.Object;@29c2fff0
[Ljava.lang.Object;@29c2fff0
```

Cet affichage correspond à ce que renvoie la méthode *toString()* appelée sur les types construits. Nous reverrons dans un autre TP guidés d'où sort cette méthode. Retenez simplement qu'elle renvoie une chaîne de caractère qui donne une représentation textuelle d'un objet (type construit).

Notez donc au passage que pour java, **un tableau est un objet**.

Décortiquons un peu les chaînes de caractères qui sont affichées:

- le '@' marque la séparation entre deux informations importantes: le **type** et l'**adresse** qui a été utilisée lors de la création de l'objet.
- dans notre exemple, l'instance *o* de la classe *Object* (plus exactement de la classe *java.lang.Object*) a été créée à l'adresse 30 6F 74 92
- l'instance tabInt est un tableau d'entiers (*[I*) créé à l'adresse 65 4e 36 15
- l'instance tabObject est un tableau d'Object ([Ljava.lang.Object) créé à l'adresse 29 c2 ff f0

On s'aperçoit donc que ma machine virtuelle java utilise 4 octets pour représenter une adresse en mémoire. La machine virtuelle java garde quelque part une association entre les variables et les adresses. Dans notre cas, la table des variables ressemblerait à quelque chose du genre:

variable	**type**	**adresse mémoire à la création**
o	Object	30 6F 74 92
tabInt	int[]	65 4e 36 15
tabDouble	double[]	71 c0 d0 a8
tabObject	Object[]	29 c2 ff f0
tabObjectCopie	Object[]	29 c2 ff f0

Créez d'autres variables et observez le résultat en console: les adresses mémoire vont changer.

Notez toutefois que l'adresse affichée n'est pas l'adresse réelle mais l'adresse à la création: java gère lui-même son espace mémoire, et souvent déplace les variables, change les adresses. Mais à ce niveau, **il n'est plus possible de savoir ce qui se passe exactement**.

Notons pour finir ce qui se passe quand on affecte une variable de type construit à une autre variable:

```
        Object[] tabObjectCopie = tabObject;
        System.out.println(tabObjectCopie);
```

Java se contente de mettre sa table de correspondance à jour, et ne crée par un nouvel espace mémoire: effectivement, le mot-clef "**new**" n'a pas été utilisé.

Ce type d'affectation s'appelle un affectation par *référence*.

Il diffère de l'affection **par copie** que nous avons vu avec les types primitifs.

Retenez ceci:

> Seul l'utilisation du mot-clef **new** permet de construire une instance en mémoire

Autrement dit, lorsqu'on déclare une variable de type construit, il n'y a pas de création d'instance.

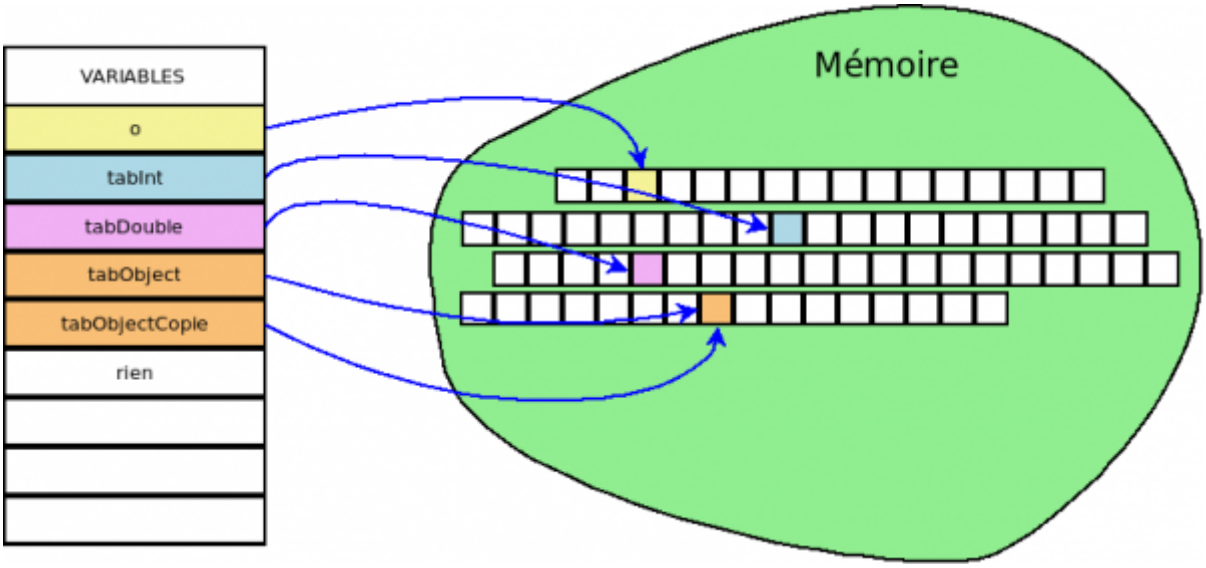
L'adresse de la variable est alors représentée par **NULL** (qui signifie: rien).

Examinons le code suivant:

```
        Object rien = null;//je n'utilise pas "new" ni une variable
construite
        System.out.println(rien);
```

L'affichage produira "null" simplement. Si j'essaie d'appeler des méthodes sur la variable *rien*, java générera une erreur de type *NullPointerException*, autrement dit: le pointeur est vide: il n'a pas d'adresse en mémoire.

La figure suivante représente cette affectation dans la mémoire.



Seul le premier octet de l'adresse est colorié. En effet, les types construits n'ont pas de taille fixe, nous allons maintenant voir comment java leur alloue la mémoire dont ils ont besoin.

La mémoire et les tableaux

Vous avez appris qu'un tableau en java ne peut pas changer de taille: il faut lui indiquer combien d'éléments il va contenir à sa création.

```
        //tableau de 3 entiers
        int[] tabInt = new int[3];

        //tableau de 4 Object
        Object[] tabObject = new Object[4];
```

Pourquoi donc?

Hé bien c'est facile à comprendre. Java va mémoriser la taille du tableau, et pour chaque case va réserver une zone mémoire qui contiendra une adresse vers la donnée.

Les adresses sont en général codées avec 4 octets. Un tableau de 20 éléments aura donc une taille de 20*4 = 80 octets, **quel que soit le type de données qu'il contient**. Si vous essayez de lire une donnée en dehors de la plage déclarée (par exemple *tabInt[120]*), java vous interdit cette opération car vous tomberiez dans une zone mémoire qui contient d'autres données que celles du tableau (120*4=480, soit le 480ème octet après le premier octet du tableau).

Java génère dans ce cas une erreur de type *ArrayIndexOutOfBoundsException*: l'index demandé est en dehors des limites du tableau.

Voici quelques exemples illustrés/

Tableau de types primitifs

En mémoire, chaque case du tableau “pointe” (c'est à dire contient l'adresse) d'une zone mémoire qui contient la donnée recherchée.

L'**adresse** de la case du tableau fait 4 octet.

La **zone pointée** fait la taille du type primitif (2 octets pour un short, 4 pour un int, etc.).

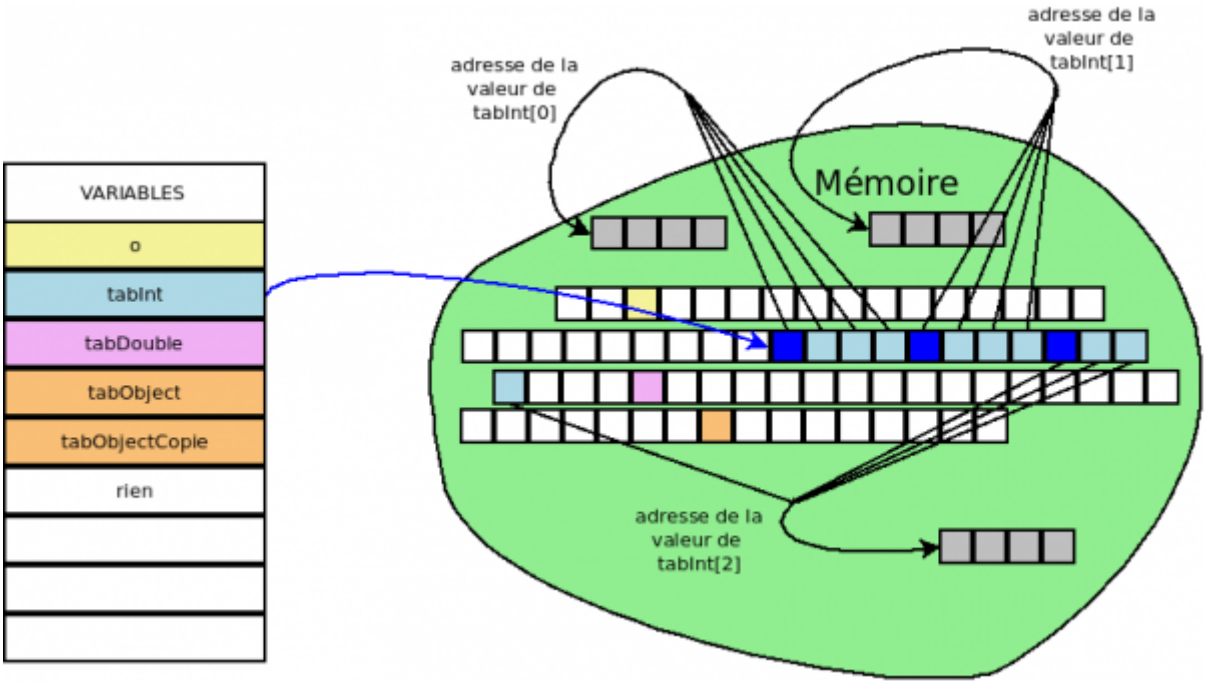


Tableau de dimension 2, 3, n...

Soit le code suivant:

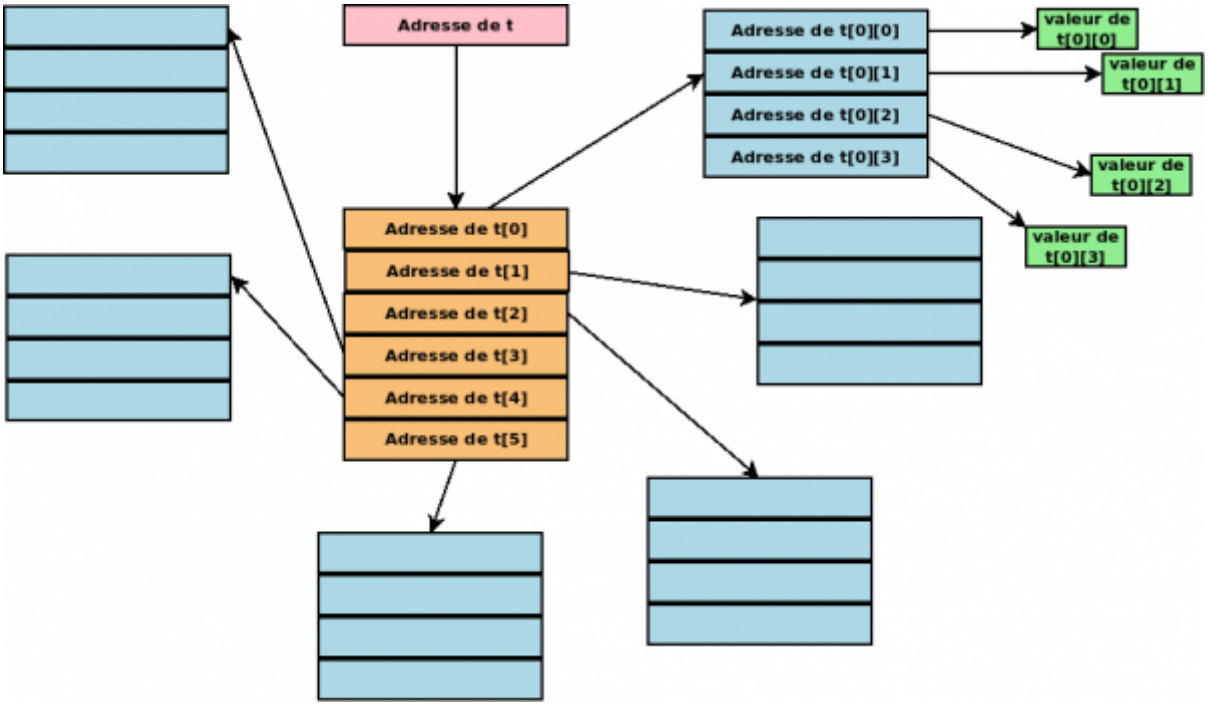
```
short [][] tab = new tab[6][4];
```

tab est un tableau de dimension 6 qui contient des tableaux de dimension 4.

Nous avons vu qu'un tableau est identifié par son adresse en mémoire.

Donc, chacune des 6 cases du tableau *tab* va contenir l'adresse d'un tableau de dimension 4. Chaque tableau de dimension 4 contiendra l'adresse où se situe la donnée de type *short*

On devrait donc parler d'un tableau de 6 adresses de tableaux de 4 adresses de short.



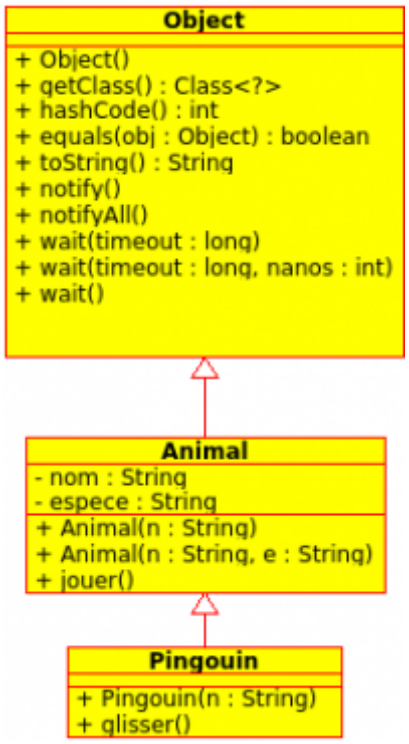
Construction d'objets

Nous avons vu que pour qu'il y ait création, il faut nécessairement que **new** soit utilisé.

Mais que fait cet opérateur **new**?

new attend en paramètre un **constructeur** (pas un nom de classe!!!).

Prenons par exemple la hiérarchie de classes suivante:



Voici le code des classes Animal et Pingouin:

Animal.java

```
/**
 Une toute petite classe représentant un Animal
 @author Bruno Mascret
*/
    public class Animal{
    /**le nom de l'animal*/
    private String nom ="";
    /** l'espèce de l'animal*/
    private String espece="";

    /**
        Constructeur d'Animal
        @param n valeur pour nom
        @param e valeur pour espece
    */
    public Animal(String n, String e){
        nom = n;
        espece = e;
    }

    /**
        Constructeur d'Animal
        L'animal est d'espèce inconnue
        @param n valeur pour nom
    */
    public Animal(String n){
        nom = n;
        espece = "inconnue";
    }

    /**
        Pour faire jouer l'Animal.
        Affiche un petit message en console.
    */
    public void jouer(){
        System.out.println("Je joue et c'est vraiment l'éclate");
    }

}
```

et Pingouin:

Pingouin.java

```
/** Une toute petite classe pour représenter un Pingouin*/
    public class Pingouin extends Animal{

    /**
```

```
        Constructeur de Pingouin, qui affecte automatiquement la chaîne
"pingouin" à l'attribut hérité "espece"
        Utilise le constructeur d'Animal Animal(String, String)
        @param n le nom du pingouin
    */
    public Pingouin(String n){
        super(n,"pingouin");
    }

    /**
        Ce qui est bien quand on est Pingouin, c'est qu'on peut glisser
sur le ventre
    */
    public void glisser(){
        System.out.println("Waou, je glisse sur le ventre");
    }

}
```

De quels constructeurs disposons-nous?

Classe Object: *Object()*

Classe Animal: *Animal(String n, String e)* et *Animal(String n)*

Classe Pingouin: *Pingouin(String n)*

Pour construire (créer) un *Object* (une instance d'*Object*), je peux utiliser n'importe lequel d'entre eux (un Pingouin **est un** Animal qui **est un** Object).

```
Object o1 = new Object();
Object o2 = new Animal("Serge");
Object o3 = new Animal("Tryphon", "Pingouin");
Object o4 = new Pingouin("Thérèse");
```

J'ai utilisé 4 fois l'opérateur **new**: j'ai donc 4 instances en mémoire.

Parmi ces instances:

- *o1* pointe vers une instance de type *Object*
- *o2* et *o3* pointent chacune vers une instance de type *Animal*
- *o4* pointe vers une instance de type *Pingouin*.

Mais est-ce que pour autant j'ai la même chose en mémoire?

NON!

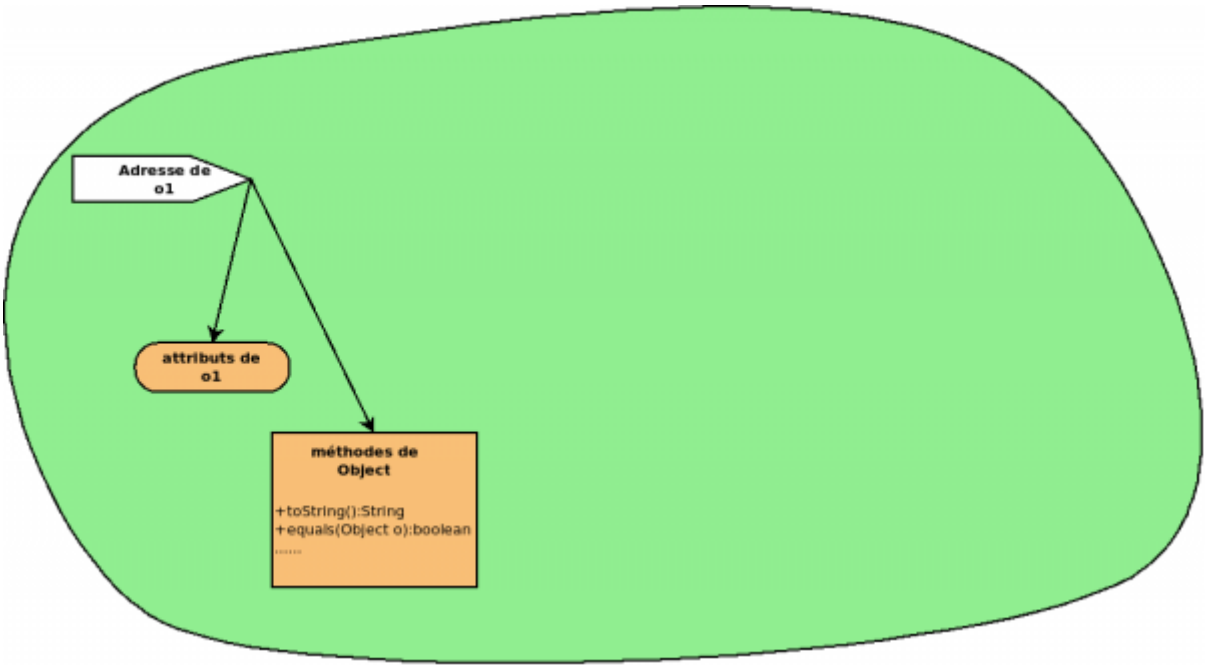
Regardons ce qui se passe en détail.

```
Object o1 = new Object();
```

Java va créer une zone de mémoire pour les attributs de *o1*, une autre pour ses méthodes (qui sont celles de la classe Object).

Les attributs (caractéristiques) de chaque instance d'Object **diffèrent**, les méthodes (comportement) restent les mêmes.

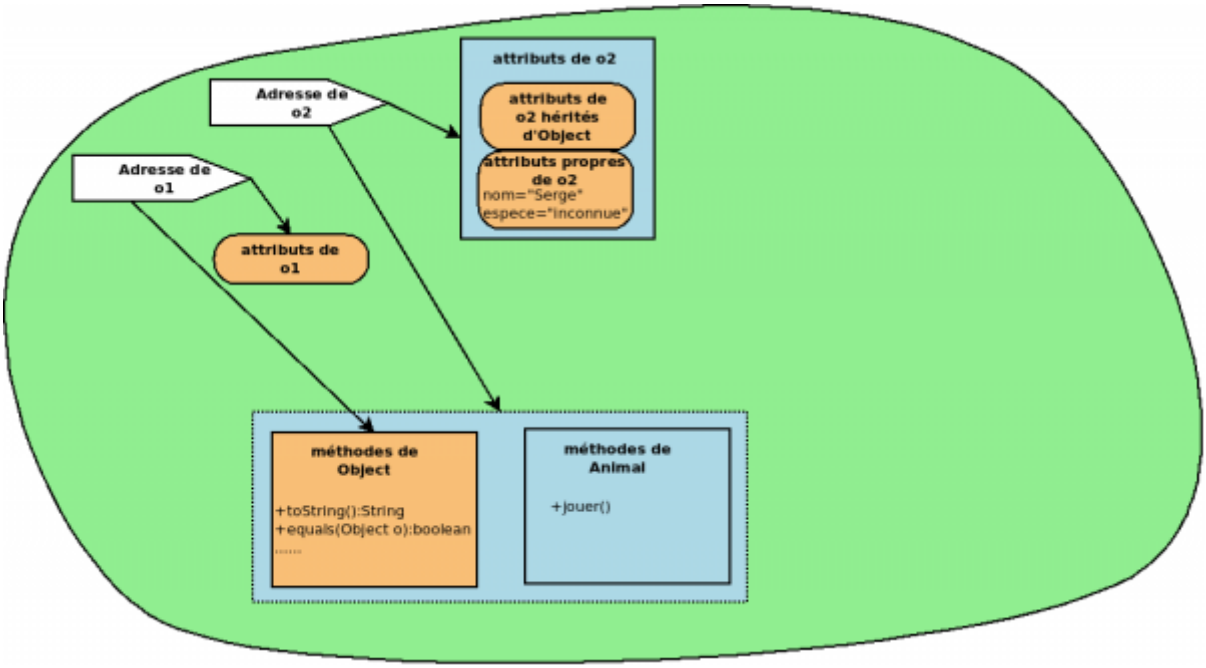
On peut schématiser cette opération de la manière suivante:




```
Object o1 = new Object();
Object o2 = new Animal("Serge");
```

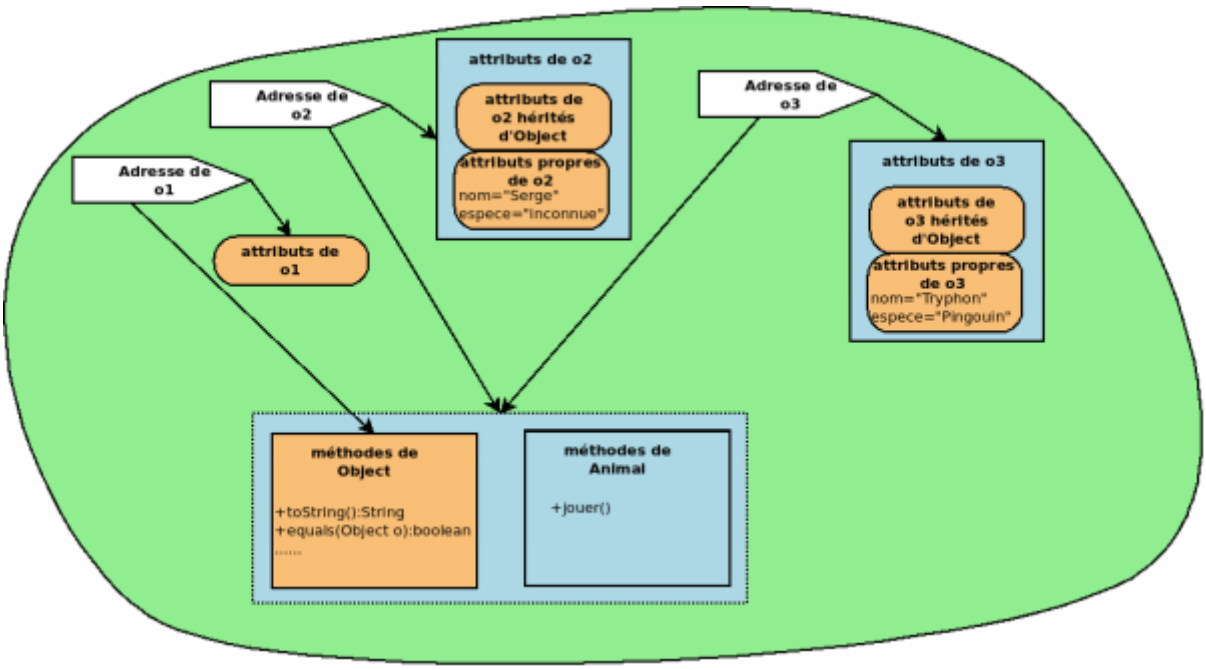
Cette fois-ci, on utilise le constructeur de Animal. Un Animal **est un** Object, donc java va commencer par construire un Object en appelant le constructeur *Object()* hérité par Animal.

Ensuite, il va ajouter les attributs de l'instance o2 venant de Animal, et les méthodes de Animal:




```
Object o1 = new Object();
Object o2 = new Animal("Serge");
Object o3 = new Animal("Tryphon", "Pingouin");
```

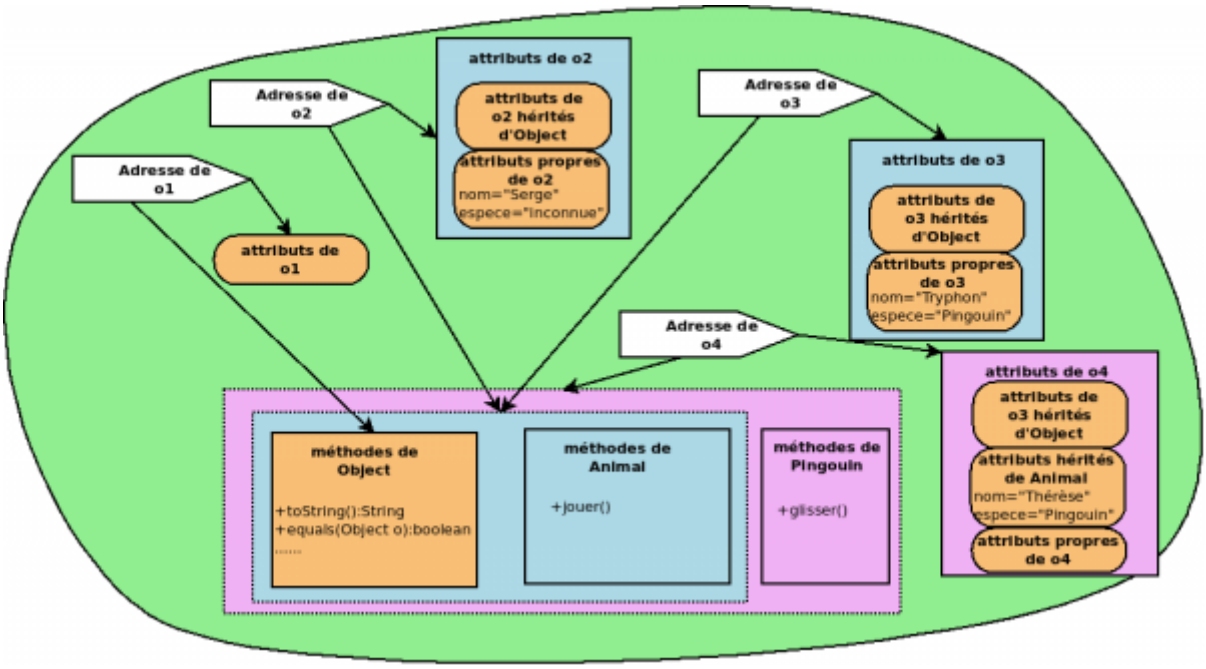
On utilise maintenant le constructeur *Animal(String n, String e)*. Il n'y a pas de différence avec le cas précédent. Notez cependant que la zone mémoire contenant les méthodes de Animal n'est pas dupliquée (les méthodes/comportement sont les mêmes pour toutes les instances d'Animal).




```
Object o1 = new Object();
Object o2 = new Animal("Serge");
Object o3 = new Animal("Tryphon", "Pingouin");
Object o4 = new Pingouin("Thérèse");
```

Cette fois-ci, le constructeur *Pingouin(String n)* va appeler **explicitement** le constructeur *Animal(String n, String e)* qui va lui-même appeler (implicitement) le constructeur *Object()*.

Il suffit alors de “dépiler” les appels et voici ce qu'on obtient au final en mémoire:



Vous vous rappelez du “Garbage Collector” (le ramasse-miettes en français).

C'est le processus de java qui s'occupe de libérer la mémoire.

La dernière figure vous permettra de bien comprendre comment il fonctionne: tant qu'une zone de la mémoire reçoit une flèche, ça veut dire qu'elle est “pointée”, donc encore utilisée.

Une zone de mémoire qui ne reçoit plus de flèche est une zone dont les données ne pourront jamais être récupérées: le garbage collector va donc pouvoir effacer cette zone, et indiquer qu'elle est libre pour une nouvelle utilisation. C'est simple avec un dessin, non?

Et les tableaux d'objets?

Que se passe-t-il dans le cas de tableaux d'objets?

Réfléchissez, vous avez déjà la réponse!

Java l'amnésique

Et pour finir, parlons des pertes de mémoires de java! Je veux parler du problème suivant:

```
Object o1 = new Object();
Object o2 = new Animal("Serge");
Object o3 = new Animal("Tryphon", "Pingouin");
Object o4 = new Pingouin("Thérèse");

o4.jouer();
```

```
o4.glisser();
```

Le compilateur java nous indique deux erreurs:

```
TypeConstruit.java:62: error: cannot find symbol
        o4.jouer();
          ^
  symbol:   method jouer()
  location: variable o4 of type Object
TypeConstruit.java:64: error: cannot find symbol
        o4.glisser();
          ^
  symbol:   method glisser()
  location: variable o4 of type Object
2 errors
```

Pourtant, *o4* a été construit avec un constructeur de *Pingouin*!

Rappelez-vous que javac n'est qu'un programme, pas toujours très intelligent donc!

Ce que javac voit, c'est que *o4* est déclaré comme *Object*.

Il cherche donc les méthodes *jouer()* et *glisser()* dans la zone de mémoire d'Object... et bien sûr ne les trouve pas car elles n'y sont pas!

Pourtant, si on regarde le dernier schéma, on voit bien que ces méthodes sont chargées dans la mémoire!

Il va donc falloir explicitement dire à Java de regarder un peu plus en détail. C'est le fameux **cast** ou **transtypage** que nous avons déjà vu avec les types primitifs, sauf que là il ne peut y avoir perte de données: soit l'opération est possible, soit elle est impossible et génerera une erreur de type *ClassCastException*.

Nous allons donc définir une variable et lui dire de pointer dans la même zone mémoire que *o4*, en demandant à java d'étendre un peu son champ de vision, ou de *retrouver la mémoire* 😉

```
        Object o4 = new Pingouin("Thérèse");
        /* ça marche pas!
        o4.jouer();
        o4.glisser();*/

        Animal a = (Animal)o4;
        a.jouer();

        ((Pingouin)o4).glisser();
```

Je vous ai mis deux manières de faire:

- dans le premier cas, je déclare une variable *a*, et après j'appelle jouer();
- dans le deuxième cas, je fais directement le cast sans passer par une variable: java créera une variable temporaire et appellera ensuite la méthode glisser().

Et la question de la fin pour voir si vous avez bien compris:

“Combien d'instances ont été créées en tout dans le dernier code???”

Codes complet des exemples

Archive du code au format tar.gz : code des fichiers java utilisés

Vous pouvez exporter ce TP en pdf en utilisant le lien du menu de gestion de cette page (le dernier, petite icône en crayon)