

## ***La machine virtuelle Java***

## ***La JVM***

- Historique et rappels
- Organisation mémoire de la JVM
- Le garbage collector
- Le bytecode, la machine à pile.
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

***Plan***

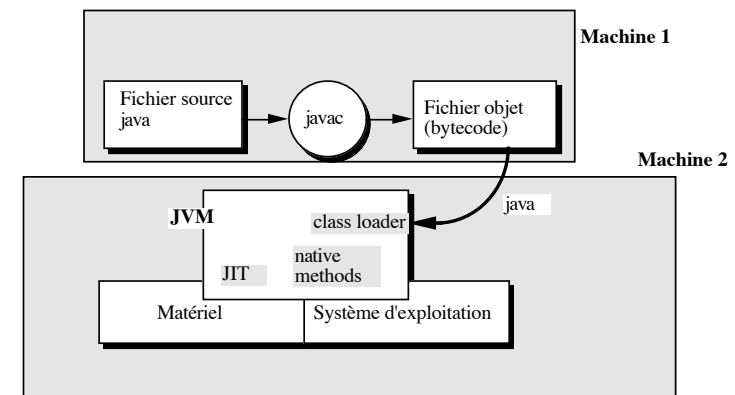
- **Historique et rappels**
- Organisation mémoire de la JVM
- Le garbage collector
- Le bytecode, la machine à pile.
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

## Rappels

- Les compilateurs java produisent un fichier contenant une représentation appelée *bytecode*.

Ce code n'est exécutable sur **aucun processeur**, il sera interprété par une machine virtuelle java ou JVM (*Java Virtual Machine*). C'est à dire que chaque instruction du bytecode sera lue et décodée puis traduite en langage machine local.

Un exécutable Java fonctionnera donc aussi bien sur un PC que sur une station Sun ou sur un Macintosh où qu'il ait été compilé, pourvu que l'on dispose d'une JVM sur la machine cible.



- Objectif :du bytecode : **Write once, run anywhere**

## ***VM de JVM : Historique***

- L'ancêtre des machine virtuelle : à l'université UCSD dans les années 70 : le p-System basé sur le langage Pascal.
- Sun, année 1992 : P. Naughton, M. Sheridan et J. Gosling mettent au point Oak et la machine virtuelle associée. Cet environnement est destiné aux systèmes embarqués, échec . Repris en 1995 et renommé en Java, Oak est proposé comme outil pour développer des applications Web. (Java, c'est le C++ sans les couteaux ni les revolvers dixit Gosling).

## ***VM de JVM : Spécificités***

- Les principaux modules d'une JVM :
  - Le chargeur de classes (class loader)
  - L'interprète de bytecode, *JITcompiler*
  - Le garbage collector
  - Et aussi : les E/S , le graphique, la sécurité, ...
  
- La JVM ne fait pas d'hypothèses sur le matériel qui l'accueille :
  - Pas d'hypothèse sur le nombre de registres -> la JVM est une machine à pile
  - Les seuls registres utilisés seront : un compteur ordinal, et trois registres pour gérer la pile
  
- Portabilité, **attention** aux liens avec le matériel et le système :
  - Gestion des threads (priorités), gestion du garbage collector
  - Gestion du graphique, gestion des E/S

## **Rappels : J de JVM**

- Différence entre classe et objet : un objet est l'instance d'une classe.  
Les tableaux et les *String* sont des objets.
- Différence entre variables et méthodes de classe, variables et méthodes d'instance :  
Une méthode d'instance est invoquée sur un objet et a accès aux variables associées à cette instance
- Pour créer un objet : appel à `new`, ou à certaines méthodes comme `getByName`, `valueOf`
- Attention :  
L'accès aux objets se fait par **référence** (un pointeur + des éléments de vérification). En effet, les paramètres sont passés par valeur, et pour un objet, on passe la valeur de cette référence.

***Plan***

- Historique et rappels
- **Organisation mémoire de la JVM**
- Le garbage collector
- Le bytecode, la machine à pile.
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

Organisation

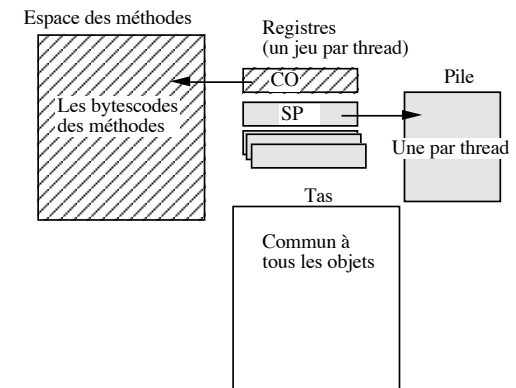
## **Organisation Mémoire de la JVM**

- L'espace des méthodes (des exécutables), c'est ici que se trouve le bytecode proprement dit, le CO pointe sur l'instruction à exécuter. Il contient aussi les tables de symboles. La zone des constantes, les tables associées aux méthodes, les définitions des classes sont rangées dans un tas non géré par le GC
- La pile où sont rangées les frames (cadres) d'invocation des méthodes. Elle est gérée par les trois pointeurs `frame`, `vars` et `optop`. Il y a une pile par thread.
- Les registres sont le CO et trois pointeurs pour gérer la pile : il y a un jeu de registres par thread.
  
- Le tas (heap) où sont rangés les objets (instances des classes et les tableaux). Il est parcouru régulièrement par le ramasse-miettes. Un seul tas.
- Si la taille de l'une de ces zones ne permet pas l'exécution d'un thread, l'exception `OutOfMemoryError` est envoyée par la JVM.

Remarque : les variables locales sont implantées sur la pile, les objets (créés par `new`) sont implantés sur le tas.

## Organisation Mémoire de la JVM

- Organisation logique de la JVM :



## ***L'espace des méthodes***

- Contient le bytecode des méthodes, des constructeurs et des informations telles que la table des symboles
- Contient aussi les informations de typage sur les objets (nom, *modifier*, nom de la superclasse, nombre et noms des interfaces implémentées, nombre de méthodes, et pour chaque méthode, un descripteur (nom, nombre de paramètres, table des exceptions, ...))
- cf `instanceOf` qui vérifie la classe d'un objet
- constant pool

## ***La pile et les registres associés***

- Chaque thread dispose de sa propre pile
- La pile contient : les paramètres passés aux méthodes exécutées, les valeurs retournées par ces méthodes, leur état, les variables locales.
- La pile est un ensemble de `frames`, elle est gérée par les pointeurs `frame` (qui donne le `frame` de la méthode courante), `vars` et `optop`

Une `frame` se divise en trois parties :

- la pile où sont rangés les opérandes des instructions du bytecode (partie repérée par le pointeur `optop`)
- l'environnement d'exécution, pour retrouver l'état de la méthode exécutée (partie repérée par le pointeur `frame`)
- variables locales de la méthode (partie repérée par le pointeur `vars`)

- Remarque :

- durée de vie d'un objet dans le tas : de `new` à sa suppression par le GC
- durée de vie d'une variable locale sur la pile : de sa création au retour de la méthode qui l'utilise,

## ***Le tas***

- Le tas héberge tous les objets d'un programme. Chaque appel à `new` entraîne une allocation de mémoire sur le tas.
- Le tas est partagé par tous les threads
- La gestion du tas est prise en charge par la JVM
- Attention : La spécification de la JVM ne dit pas comment doit fonctionner le garbage collector. (L'algorithme le plus employé est de type mark and sweep).

***Plan***

- Historique et rappels
- Organisation mémoire de la JVM
- **Le garbage collector**
- Le bytecode, la machine à pile.
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

## ***Le Garbage collector (ramasse miettes)***

Sur le tas sont rangés les objets créés lors de l'exécution d'un programme (appel à `new`). Le GC se charge en particulier de libérer l'espace occupé par les objets qui ne sont plus référencés.

Le GC **recycle** l'espace mémoire :

- récupération des blocs libres,
- exécution des « finalizers »
- gestion de la fragmentation (pénalisante même dans le cas d'une mémoire paginée : on augmente le taux de pagination) par compactage et garbage collection,

• Fonctionnement du GC :

- plusieurs algorithmes sont utilisés : mark and sweep est le plus courant
- le GC automatique libère le programmeur des pb de gestion du tas, il garantit que la JVM ne s'arrête pas à cause d'un pb mémoire
- inconvénients : overhead, non déterminisme (par exemple, les finalizers sont exécutés par le GC, on ne peut donc pas savoir précisément quand), plus de temps consommé que s'il était fait à la demande
- fait par un thread dédié, appels explicites : `System.gc()` et `Runtime.gc()` (immédiat sur Sun et NT)

## ***Configuration et suivi du Garbage collector***

- L'option `-verbose:gc` de la commande `java` trace les appels au Garbage Collector
- Options de la commande `java` pour affiner le dimensionnement des zones utilisées par le GC :
  - `Xmx` : modification de la taille du tas (défaut de 64Mo, souvent insuffisant)
  - `Xms` : la taille minimale du tas (faire `Xms = Xmx`)
  - `-XX:NewRatio=n` : `n` est le ratio entre l'ancienne et la jeune génération (celle-ci doit utiliser plus de 50 % du tas)
  - `noasync` : pas de GC asynchrone (le thread de GC n'est pas utilisé), seulement à la demande, ou si l'on n'a plus de mémoire
  - `noclassgc` : pas de GC pour les classes inutilisées

## GC : Exemple(1)

```
class Test_Threads {
    public static void main (String args[]) {
        for (int i=0; i < 20; i++){
            new MonThread("_Thread_" + i).start();
        }
        System.out.println("GC 1 : AVANT");
        System.gc();
        System.out.println("GC 1 : APRES");
        for (int i=0; i < 200; i++)
            new MonThread("__Thread_" + i).start();

        System.out.println("GC 2 : AVANT");

        Runtime r = Runtime.getRuntime() ;

        // Quantite de memoire libre
        long free = r.freeMemory();
        // Quantite de memoire diponible, allouee et non-allouee
        long total = r.totalMemory();
        // Quantite de memoire allouee
        long inuse = r.totalMemory() - r.freeMemory() ;

        System.out.println(" free =" + free + " total="+ total+ "
inuse="+ inuse);

        System.gc();
        System.out.println("GC 2 : APRES");
        while (true) {};
    }
}
class MonThread extends Thread {
    public MonThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
    }
}
```

## GC : Exemple (2)

- Exécution du programme sur JVM 1.3 :

```
java -verbosegc -noasyncgc Test_Threads
Warning: -noasyncgc option is no longer supported.
GC 1 : AVANT
[Full GC 1198K->316K(3520K), 0.1136632 secs]
GC 1 : APRES
GC 2 : AVANT
      free =3215304 total=3604480 inuse=389176
[Full GC 380K->347K(5248K), 0.1151222 secs]
GC 2 : APRES
```

- Exécution du programme sur JVM 1.1 :

```
$ java -verbosegc -noasyncgc Test_Threads
GC 1 : AVANT
<GC: heap 5120K, total before 1338K, after 1317K (6588/6260 objs)
  74.3% free, allocated 7083K (#11941), marked 94K, swept 21K (#328)
  0 objs (0K) awaiting finalization>
GC 1 : APRES
<GC: heap 5120K, total before 4548K, after 4461K (8346/7267 objs)
  12.9% free, allocated 3235K (#2090), marked 299K, swept 86K (#1079)
  0 objs (0K) awaiting finalization>
<GC: heap 7168K, total before 6379K, after 6297K (8530/7914 objs)
  12.1% free, allocated 1917K (#1263), marked 439K, swept 81K (#616)
  0 objs (0K) awaiting finalization>
GC 2 : AVANT
      free =57671680 total=67108864 inuse=9437184
<GC: heap 9216K, total before 7946K, after 7871K (8995/8425 objs)
  14.6% free, allocated 1704K (#1151), marked 539K, swept 75K (#570)
  0 objs (0K) awaiting finalization>
GC 2 : APRES
```

## ***Plan***

- Historique et rappels
- Organisation mémoire de la JVM
- Le garbage collector
- **Le bytecode, la machine à pile.**
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

## ***Qu'est ce qu'une Machine à pile***

- Soit l'instruction écrite en langage de haut niveau :

$i = j + k$  ;

- Code généré sur une machine à pile (MP) :

```
push j      /* mettre j en haut de pile */
push k      /* mettre k en haut de pile */
add         /* additionner les deux valeurs et les dépiler
           empiler le résultat */
store i     /* prendre le sommet de pile et le ranger dans i */
```

- Comptons les accès à la mémoire : L pour load (mémoire vers pile), S pour store (pile vers mémoire) :

```
push j      /* L S */
push k      /* L S */
add         /* L L S */
store i     /* L S */
```

il y a plus d'accès mémoire que sur un processeur doté de registres, mais moins de temps est passé en décodage et recherche des opérandes

- Dans le cas de la compilation à la volée (just in time compiler, JIT), il faut gérer l'allocation des registres à partir du bytecode...

## ***Exécution D'un programme Java : généralités***

- Un programme Java est organisé en fichiers objets appelés classes, chaque classe étant compilée séparément.
  - La JVM commence l'exécution en chargeant (*loading*) la classe contenant la méthode `main`. Ce travail de chargement est assuré par le `ClassLoader` qui opère à la demande ou de façon anticipée en pré chargeant certaines classes.
  - La deuxième étape est le *linking* ou édition de liens qui se divise en trois phases :
    - vérification des contraintes de sécurité et de sémantique
    - préparation de l'espace d'adressage : allocation et initialisation des zones mémoires,
    - résolution des références non satisfaites. Cette phase peut entraîner de nouveaux chargements. Ceux-ci peuvent être faits immédiatement ou seulement à l'appel de la référence, suivant la technique du *lazy linking*.
  - Un thread est créé pour exécuter la méthode `main`.
- *La compilation à la volée (just in time ou JIT)* consiste à traduire le byte code en langage machine au fur et à mesure de son décodage, après la phase de vérification. Cette technique peut diminuer d'un facteur 10 le temps de traitement du byte code qui n'est plus interprété, mais compilé.

## Les instructions De la JVM

- Les fichiers contenant du bytecode peuvent être désassemblés grâce à l'utilitaire javap.

Rappel : le bytecode est rangé dans l'espace des méthodes

Par exemple pour le fichier suivant :

```
public class HelloWorld
{
    public static void main(String args[])
    {
        int i;
        for ( i=0 ; i<10 ; i++) ;
        System.out.println(" Hello world,  i = " + i);
    }
}
```

```
$ javac HelloWorld.java
```

```
$ java HelloWorld
```

```
Hello world,  i = 10
```

```
$ javap -c HelloWorld
```

```
...
 0 iconst_0           # mettre 0 sur la pile
 1 istore_1           # vider la pile dans la
                     # première variable
 2 goto 8
 5 iinc 1 1           # ajouter 1 à la première
variable
 8 iload_1
 9 bipush 10          # (on ne peut pas incrémenter
11 if_icmplt 5        # directement sur la pile)
                     # empiler la première variable
                     # mettre 10 sur la pile
                     # comparer les deux valeurs en
                     # sommet de pile
...
36 return
```

### **Bytecode : exemple (1)**

```
// Gestion de la machine a pile Java
// Pour desassembler et obtenir le bytecode, utiliser la commande :
//                               javap -c -verbose Demol > Demol.bc#

public class Demol {

    public static void main(String[] args) {
        System.out.println("Coucou !");
        // GestionFile();
    }

    public void GestionFile() {
        int Var1, Var2, Var3, Var4, Var5, Var6;

        // Pour voir les differents types de load
        Var1 = 2;
        Var2 = 7;
        Var3 = 200;
        Var4 = 1000000; // 1000000 est une GROSSE constante ...

        // Arithmetique sur la pile
        Var5 = Var1 + Var4;
        // Gestion des resultats intermediaires
        Var5 = 3*( Var1/Var2) - 2/(Var4 + Var5) );
        Var1++;
        Var2 <<= 3;
        // Pour voir ...
        Var6 = 2*(Var1 = 4);
        // Appel de methode
        Var3 = Ajouter(Var1, Var2, -1);
    }

    public int Ajouter(int x, int y, int z) {
        int Somme;
        Somme = x + y + z;
        return Somme;
    }
}
```

## ***Bytecode : exemple (2)***

- Le début du fichier et la fonction main en byte code :

```
Compiled from Demol.java
public class Demol extends java.lang.Object {
    public Demol();
    /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
    /* Stack=2, Locals=1, Args_size=1 */
    public void GestionFile();
    /* Stack=5, Locals=7, Args_size=1 */
    public int Ajouter(int, int, int);
    /* Stack=2, Locals=5, Args_size=4 */
}

Method Demol()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object.>
  4 return

Method void main(java.lang.String[])
  0 getstatic #2 <Field java.io.PrintStream out>
  3 ldc #3 <String "Coucou !">
  5 invokevirtual #4 <Method void println(java.lang.String)>
  8 return
```

## Bytecode : exemple (3)

- La fonction GestionPile en byte code :

```
Method void GestionPile()
 0 iconst_2          /* push constante 2 */
 1 istore_1          /* Rangement dans la variable num. 1 */
 2 bipush_7          /* byte integer push */
 4 istore_2          /* Rangement dans la variable num. 2 */
 5 sipush 200        /* short integer push */
 7 istore_3          /* Rangement dans la variable num. 3 */
 8 ldc #5 <Integer 1000000> /* CREATION et push de la cste */
10 istore 4

12 iload_1
13 iload 4
15 iadd
16 istore 5          /* 12 a 16 : Var5 = Var1 + Var4; */

18 iconst_3
19 iload_1
20 iload_2
21 idiv              /* (Var1/Var2) */

22 iconst_2
23 iload 4
25 iload 5
27 iadd              /*(Var4 + Var5) */
28 idiv
29 isub
30 imul              /* 3*( (Var1/Var2) - 2/(Var4 + Var5) ) */

31 istore 5
33 iinc 1 1          /* INSTRUCTION SPECIALE Var1++; */
36 iload_2
37 iconst_3
38 ishl
39 istore_2
40 iconst_2
41 iconst_4
42 dup
43 istore_1
44 imul
45 istore 6
47 aload_0
48 iload_1
49 iload_2
50 iconst_m1         /* push constante -1 */
51 invokevirtual #6 <Method int Ajouter(int, int, int)>
54 istore_3
55 return
```

## ***Bytecode : exemple (4)***

- La fonction Ajouter en Java :

```
public int Ajouter(int x, int y, int z) {  
    int Somme;  
    Somme = x + y + z;  
    return Somme;  
}
```

- Le byte code correspondant :

```
Method int Ajouter(int, int, int)  
0 iload_1  
1 iload_2  
2 iadd  
3 iload_3  
4 iadd  
5 ireturn
```

- Rappel : l'appel à la méthode en Java et byte code:

```
// Appel de methode  
Var3 = Ajouter(Var1, Var2, -1);  
  
51 invokevirtual #6 <Method int Ajouter(int, int, int)>  
54 istore_3
```

## ***Bytecode : Optimisations***

- Interpréter le code est très pénalisant, par exemple dans le cas des boucles : on refait la même opération n fois. D'où l'intérêt de la compilation à la volée .

- Compilation à la volée (Just in time compilation)

Dans ce cas, la JVM compile le bytecode en code directement exécutable par le processeur au lieu de l'interpréter .

Cette option est utilisée par défaut sur les stations Sun, on peut la désactiver

***Plan***

- Historique et rappels
- Organisation mémoire de la JVM
- Le garbage collector
- Le bytecode, la machine à pile.
- **Les threads**
- Suivi, tracé, optimisation d'un programme Java
- JVM embarquées

## ***La gestion des threads***

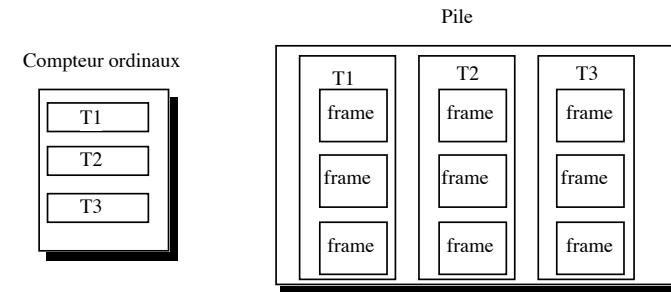
- Rien n'est dit sur l'ordonnancement des threads dans la spécification de la JVM.
- Changement de contexte si :
  - le courant sort de l'état runnable, se bloque ou se termine
  - yield passe la main au suivant de même priorité, ou au premier de priorité immédiatement inférieure
  - un plus prioritaire devient runnable

## ***La gestion Des priorités***

- Les priorités sont numérotées 1(min) à 10 (max), défaut : 5. Un thread hérite de la priorité de son créateur
- setpriority() modifie la priorité
  - paramètres :Thread.MIN\_PRIORITY, Thread.MAX\_PRIORITY , Thread.NORM\_PRIORITY
- Pour setPriority, on trouve :  
*Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.*

## **La gestion Mémoire Pour les threads**

- Gestion de la mémoire pour les threads : leurs piles sont généralement allouées sur le tas, à chaque thread est associé un compteur ordinal.



- Au démarrage d'un thread, la JVM lui associe une pile. A chaque invocation de méthode est associée une « frame ». Cette frame contient les paramètres d'appel (les objets sont toujours passés par référence), les résultats intermédiaires, des informations de gestion (exceptions,...)
- A la fin de la méthode, la frame est dépilée, la frame suivante devient la « current frame ».
- La structure de données qui implémente cette pile peut être contiguë ou non, un tas ou une pile, cela dépend de l'implémentation

**Synchronisation :  
Opérations  
Sur les long et double,  
volatile**

- Les opérations sur des variables du type `double` ou `long` **peuvent** être traitées comme si ces variables étaient implantées sur deux mots de 32 bits.
- Les opérations (même les affectations) sur les `long` et `double` peuvent donc ne pas être atomiques. Pour assurer cette atomicité il faut les déclarer :  

```
volatile double  
volatile long
```
- Bien sûr, si des variables partagées du type `long` ou `double` sont dans des zones **synchronized**, le problème ne se pose pas.
- D'où l'utilité de **toujours** gérer correctement (c'est-à-dire en utilisant des outils de synchronisation) l'accès aux ressources partagées.

## ***Synchronisation : Utilisation de Synchronized***

- A chaque objet est associé un verrou qui est pris lors de l'entrée dans une méthode ou une région de code qualifiée de `synchronized`.
- `synchronized` doit être utilisé pour gérer tout accès à des variables partagées par plusieurs thread, y compris les affectations.
- Attention à l'utilisation de `synchronized` sur une méthode `static` : on utilise le verrou qui gère l'accès à la **classe** !

***Plan***

- Historique et rappels
- Organisation mémoire de la JVM
- Le garbage collector
- Le bytecode, la machine à pile.
- Les threads
- **Suivi, tracé, optimisation d'un programme Java**
- JVM embarquées

## ***Optimisations***

- Suivi et traces : `java -verbose:gc -Xprof -Xrunhprof`
- Vérifier la portée des objets (scope)
- Attention à l'utilisation des outils de synchronisation (granularité)
- Utiliser `StringBuffer` plutôt que `String` pour manipuler des chaînes de caractères
- Utiliser des méthodes statiques :

```
int i = new Integer(Valeur).intValue() ;  
int i = Integer.parseInt(Valeur);
```

- Allocation des variables :
  - rapide pour les variables locales (restent sur la pile)
  - lente sur le tas : variables statiques et variables d'instances
- Il vaut mieux créer un tableau statique de 2000 octets que d'en allouer à chaque appel d'une méthode fréquemment appelée
- `Vector` et `WrappedArrayList` sont synchronisées, pas `ArrayList`

## **Optimisations**

### **Exemple : le type *String***

- La JVM réutilise et partage les `String` parce qu'elles sont des constantes et donc thread-safe.

La classe `String` gère un pool de `String`'s: si la chaîne existe déjà, la JVM renvoie une référence sur cette chaîne, sinon elle en crée une nouvelle.

- Mais la création par `new` ignore ce mécanisme... utiliser `new` pour une `String` que si nécessaire. Dans ce cas, la chaîne est allouée sur le tas (pas sur une JVM embarquée).

- Exemple :

```
String chaine1 = "Bonjour";  
String chaine2 = "Bonjour" ;  
String chaine3 = new String("Bonjour") ;
```

`chaine2 == chaine1` ; renvoie vrai (**pointeurs égaux**)

`chaine3 == chaine1` ; renvoie faux.

- Rappel :

`==` vérifie les références,

`equals()` vérifie les contenus!

## ***String et StringBuffer***

- Les `String` sont des constantes chaînes, `StringBuffer` des variables.

La concaténation est plus efficace avec `StringBuffer` qu'avec `String`

***Plan***

- Historique et rappels
- Organisation mémoire de la JVM
- Le garbage collector
- Le bytecode, la machine à pile.
- Les threads
- Suivi, tracé, optimisation d'un programme Java
- **JVM embarquées**

## ***Les JVM embarquées***

- Contraintes sur le matériel :
  - taille de la mémoire, vitesse du processeur, consommation
  - capacités graphiques réduites
  
- Contraintes logicielles :
  - ordonnancement, synchronisation
  - fonctionnement du GC
  - gestion des E/S, des interruptions
  - chargement dynamique des classes

