

# Optimisation

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)

## TABLE DES MATIERES

1.	Optimiser la vitesse.....	3
2.	Optimiser la mémoire.....	12
3.	Optimiser le chargement.....	16
3.1	Le chargement d'une applet .....	16
3.2	Améliorer le chargement .....	16
3.3	Anticiper l'utilisation.....	17
3.4	Architecture globale .....	17
3.5	Les autres techniques .....	18
4.	Conclusion .....	20

## Avant-propos

*Ce document décrit différentes techniques pour optimiser un programme Java. Il y a trois objectifs à obtenir : une meilleure performance, une limitation des ressources nécessaires et un chargement rapide de l'applet.*

Donal Knuth, un des pionniers de la science des ordinateurs, disait : "Une optimisation prématurée est le chemin du diable".

Dans un monde idéal, il ne serait pas nécessaire d'optimiser les programmes. Les machines fonctionneraient à une vitesse tel, que n'importe quel algorithme puisse être utilisé sans que cela ait un impact pour l'utilisateur. Les ressources seraient également infinies.

Malgré les augmentations régulières de la puissance des machines, il est parfois nécessaire d'optimiser les programmes. Cela demande un effort particulier qui ne se justifie que rarement.

Il y a de nombreuses raisons pour ne pas optimiser votre code :

- S'il fonctionne, l'optimiser entraîne nécessairement de nouvelles erreurs subtiles.
- Un code optimisé est plus difficile à maintenir et à comprendre.
- Optimiser un code pour une plate-forme peut pénaliser le programme sur une autre.
- Il faut consommer beaucoup de temps pour optimiser un code, pour un gain négligeable.
- Un code est plus souvent lu que modifié.

Avant d'optimiser un programme, il faut bien mesurer les avantages et les inconvénients. Il est préférable de modifier profondément un algorithme que d'apporter des optimisations très localisées. Le code le plus rapide est celui que l'on n'exécute jamais ! Une optimisation de haut niveau est toujours préférable à une optimisation de bas niveau.

Pour rédiger un algorithme optimisé, donc complexe, il est préférable de rédiger précédemment un algorithme simple, non optimisé. Si les performances sont dramatiques, l'algorithme simple permettra de vérifier l'algorithme complexe par des tests en post-condition. Le même traitement est effectué par l'algorithme simple et par l'algorithme complexe et leurs résultats sont comparés.

Pourquoi optimiser le code ? Dans le monde réel, il y a deux problèmes que les développeurs doivent résoudre. La vitesse des machines et les ressources ne sont pas illimitées. Les programmes passent généralement 90% de leurs temps dans 10% du code. Il est nécessaire d'identifier les 10% stratégiques avant d'entamer une optimisation. Java impose une troisième voix : la réduction du temps de chargement d'une applet.

Dans l'idéal, un programme java doit être petit, rapide et consommer peu de ressource. Nous allons regarder comment répondre à ces différentes stratégies. Certaines techniques sont contradictoires. Par exemple, elles améliorent la vitesse du programme, mais en augmentent sa taille. Il faut user de mesure pour équilibrer les trois objectifs.

Les compilateurs sont capables d'effectuer certaines optimisations (`javac -O`). Ils calculent automatiquement les constantes comme `i = (10*10)` compilé en `i = 100`, supprime les sauts en cascade (un `goto` qui arrive sur un autre `goto`) et élimine les code mort (`if (false) ...`)

La deuxième génération des machines virtuelles propose un compilateur à la volée (JIT). Le byte-code est converti en assembleur. Les compilateurs JIT doivent privilégier la vitesse de compilation au détriment de la qualité du code généré.

La troisième génération des machines virtuelle, inauguré par Sun avec HotSpot™ permettent des optimisations impossibles avec un compilateur classique. Java est un langage polymorphique. Cela veut dire que chaque appel de méthode non `final` peut arriver sur un traitement différent. HotSpot™ détecte à la volée les différentes situations arrivant effectivement lors de l'exécution et importe le corps de la méthode dans l'appelant avant d'effectuer une optimisation agressive. Par exemple, la méthode `toString()` est surchargée un nombre incalculable de fois. Dans un contexte particulier, seule la méthode `Date.toString()` peut être appelé. HotSpot™ détecte cela et remplace l'appel à la méthode par son contenu. HotSpot™ détecte les 10% du code le plus utilisé et génère alors des méthodes possédant en ligne, pratiquement toutes les méthodes appelées. Ces traitements utilisent une optimisation agressive, impossible à obtenir avec un compilateur en ligne de commande.

## 1. OPTIMISER LA VITESSE

Il existe de nombreuses techniques pour améliorer la vitesse. Les contextes d'exécutions d'un programme Java étant variés, il est difficile de choisir la meilleure approche répondant à toutes les situations. Certaines rédactions sont efficaces avec une machine virtuelle mais pas avec d'autres. On peut quand même décrire quelques règles s'appliquant généralement à toutes les machines virtuelles.

### Compiliez optimisé

Utiliser le paramètre `-O` lors de la compilation. Le compilateur est alors capable de placer en ligne les méthodes `private` et `final`.

### Déclaré des méthodes utilitaires

Une méthode qui ne manipule pas d'attribut ni de variables statiques est un utilitaire. Vous pouvez la déclarer `static`. Cela améliore la vitesse d'exécution. Par exemple, la méthode `copyValueOf(char[])` de la classe `String` est déclaré `static` car elle ne manipule pas d'instance.

### Utiliser les API

Certaines méthodes de l'API Java sont rédigées en C et permettent une nette amélioration des performances. Par exemple, la méthode `arrayCopy()` permet de copier un tableau très rapidement.

## Remplacez les API

Parfois, les API de Java font plus que ce que vous désirez. Vous pouvez améliorer les performances en spécialisant le code pour votre besoin spécifique. Par exemple, la classe `Vector` permet d'avoir une liste chaînée de tous types d'objet. Il est plus efficace, si les instances à manipuler le permettent, d'ajouter une référence `next_` et `previous_` directement dans l'objet. Cela réduit le nombre d'instance en mémoire et évite une conversion.

La classe `Date` de java est riche. Elle peut judicieusement être remplacé par un `long`, initialisé par exemple avec un appel à `System.currentTimeMillis()`. Lors de la manipulation de la date, construisez une instance `Date` ou utilisez la classe `Calendar`.

La classe `Dimension` permet de manipuler une hauteur et une largeur. Cela entraîne la création d'une instance, et sa duplication si nécessaire. Il est parfois préférable de proposer directement deux accesseurs pour obtenir ces informations.

```
public class MaClass
{
    private int width_;
    private int height_;

    public Dimension getDimension()
    { return new Dimension(width_,height_);
    }
    public int getHeigth()
    { return height_;
    }
    public int getWidth()
    { return width_;
    }
}
```

## Réutilisez les instances

Créer des objets est plus coûteux que les réutiliser. Déclarez des instances statiques et réinitialisez les.

## Recyclez les instances

La création d'une instance est longue et demande du travail au ramasse-miettes. Pour réduire cela, il est préférable de les recycler. Proposez une méthode `copy()` pour chaque constructeur.

```
class MaClass
{ public MaClass(int x)
  { ...
  }
  public MaClass(String x)
  { ...
  }
  public void copy(int x)
  { ...
  }
  public void copy(String x)
  { ...
  }
}
```

Vous pouvez créer un groupe d'instance permettant de les recycler. Par exemple, pour obtenir une instance libre de type `Rectangle`, il faut appeler la méthode statique `PoolRectangle.getRectangle()` et l'initialiser. Lorsque l'instance n'est plus nécessaire, il faut la replacer dans le groupe par la méthode `releaseRectangle(rect)`. Pour libérer les instances `Rectangle` qui ne sont plus nécessaires, vous pouvez utiliser le package `java.lang.ref` du JDK 1.2. Le ramasse-miettes supprimera automatiquement les instances inutiles lorsque ce sera nécessaire.

## Recyclé les tâches

La création d'une tâche est un processus long. Il est préférable de les recycler pour économiser le temps de création. Attention, chaque tâche exige 32KB pour la pile d'appel.

## Modification d'une instruction par une autre.

Certains traitements sont plus rapides que d'autre.

- `x >> 2` peut être utilisé à la place de `x / 4`
- `x << 1` remplace `x * 2`
- `a[i] += x` remplace `a[i] = a[i] + x`
- `MaClass.class` remplace `Class.forName("MaClass")`.
- `(x<y) ? x : y` remplace `Math.min(x,y)`
- `(x>y) ? x : y` remplace `Math.max(x,y)`

- `"value".equals(ref)` remplace `(ref!=null) && (ref.equals("value"))`

## Élimination des sous-expressions commune.

Cela consiste à déplacer les sous-expressions pour ne les calculer qu'une seule fois.

```
double prixTTC1 = prix1 * (1+tva);
double prixTTC2 = prix2 * (1+tva);
```

Ce code est remplacé par :

```
double sousexpr = 1+tva;
double prixTTC1 = prix1 * sousexpr;
double prixTTC2 = prix2 * sousexpr;
```

Les compilateurs savent parfois identifier ce type de calcul, mais ne peuvent pas simplifier les expressions ayant potentiellement un effet de bord. Par exemple, ils ne peuvent pas factoriser les appels de méthode.

```
double prixTTC1 = prix1 * getTVA();
double prixTTC2 = prix2 * getTVA();
```

Dans ce cas, il faut optimiser les expressions à la main.

```
double sousexpr = getTVA();
double prixTTC1 = prix1 * sousexpr;
double prixTTC2 = prix2 * sousexpr;
```

## Déplacement de code

Cela consiste à détecter les codes dont le résultat ne varie pas et à le déplacer en dehors des boucles.

```
for (int i=0 ; i < x.length ; ++i)
    x[i] *= Math.PI * Math.cos(y);
```

devient

```
double sousexpr = Math.PI * Math.cos(y);
for (int i=0 ; i < x.length ; ++i)
    x[i] *= sousexpr;
```

Vous pouvez également sortir l'appel à `length` de la boucle.

```
double sousexpr = Math.PI * Math.cos(y);
int len=x.length;
for (int i=0 ; i < len ; ++i)
    x[i] *= sousexpr;
```

Le gain est encore plus important lors de l'utilisation d'un conteneur. La méthode `length()` doit être sorti de la boucle.

```
Vector v=...
int len=v.length();
for (int i=0 ; i < len ; ++i)
    x.elementAt(i) *= sousexpr;
```

## Organiser une expression

Java impose un ordre strict pour l'évaluation d'une expression. Cela limite les optimisations effectuées généralement par les compilateurs.

```
1 + i + 2
0.5 * j * 2
```

Java interprète ces expressions comme ceci :

```
(1 + i) + 2
(0.5 * j) * 2.0
```

Les interprétations par un compilateur C/C++ peuvent être

```
(1 + i) + 2 ; 3 + i ; i + 3
(0.5 * j) * 2.0 ; 1.0 * x ; x
```

Il faut éventuellement réorganiser une expression pour simplifier les calculs.

L'approche de Java est parfois pénalisante, mais dans d'autre situation, elle permet d'éviter les surprises.

```
int i=1;
f(i = 3, i, ++i)
```

est compris par Java en

```
f(3,3,4)
```

Un compilateur C/C++ peut comprendre

```
f(3,3,4) ; f(3,1,2) ; f(3,1,4)
```

Quel que soit le compilateur Java, le code est toujours exécuté dans le même ordre.

Parfois, il est préférable d'organiser une expression par rapport à la probabilité d'évaluation.

```
if ((age <= 6) || (age > 17))
```

Cette expression doit être réorganisé, car la population est de plus en plus vieille.

```
if ((age > 17) || (age <=6))
```

Ainsi, le deuxième test sera exécuté moins souvent. Il faut que le test le plus probable soit exécuté en premier.

## Réduction d'une boucle

Cela consiste à effectuer plusieurs traitements dans une seule itération d'une boucle. Par exemple, si vous savez que `x.length` est un multiple de deux, vous pouvez effectuer deux calculs à chaque itération.

```
for (int i=0 ; i < x.length ; i += 2)
{ x[i] *= 2;
  x[i+1] *= 2;
}
```

## Inversez l'ordre d'exécution d'une boucle

Si l'ordre de traitement d'une boucle n'a pas d'importance, il est préférable de l'inverser pour pouvoir comparer la condition de sortie par rapport à la constante zéro.

```
for (int i=0;i<array.length;++i)
{ ...
}
```

devient

```
for (int i=array.length-1;--i>=0;)
{ ...
}
```

La machine virtuelle de Java, comme tous les microprocesseurs, possède des instructions spécifiques pour comparer un entier avec la constante zéro. Le code est ainsi plus court et plus rapide.

## Favoriser les méthodes inline

Une méthode inline est une méthode qui peut être inséré à la place de chaque appel. Cela permet de supprimer l'appel de la méthode et permet également au compilateur d'optimiser l'appelant.

Lorsque vous rédigez votre programme, essayez de prévoir l'arbre syntaxique compris par le compilateur. Certains experts rédigent leurs programmes en fonction des possibilités d'optimisations. Par exemple, un code comme celui-ci :

```
class Jeu
{
  static final Direction[] direction_=
  {
    Direction.Nord,
    Direction.Ouest,
    Direction.Sud,
    Direction.Est
  };

  private void recherche(Direction direction)
  { //...
  }

  void rechercheAll()
  { for (int i=0;i<4;++i)
    { recherche(direction_[i]);
    }
  }
}
```

utilise un tableau de constantes pour manipuler l'ensemble des quatre points cardinaux. La boucle `i` appelle la fonction `recherche` pour chacune des constantes. Une seule génération de cette fonction est présente dans le corps de la boucle. Cette traduction utilise une variable pour le paramètre de `recherche`. Le compilateur ne peut pas optimiser la fonction `recherche` lors de sa génération en ligne. Si le programme est rédigé différemment :

```
class Jeu
{
  ...
  void rechercheAll()
  { recherche(Nord);
    recherche(Ouest);
    recherche(Sud);
  }
}
```

```

    recherche(Est);
}
}

```

le compilateur va pouvoir spécialiser la fonction `recherche` pour chacune des constantes d'orientation. Le choix de la rédaction de `rechercheAll()` est ici dicté par les possibilités d'optimisation du compilateur. Le corps de la fonction sera beaucoup plus gros, car quatre générations de `recherche()` seront présentes. Par contre, l'exécutable sera plus rapide.

Si la méthode `recherche` possède des instructions du type `if (direction==Nord)` le compilateur pourra détecter du code mort et supprimer le test.

Vous pouvez également modifier les signatures de vos méthodes pour augmenter les chances d'utilisation de constantes. Par exemple, un programme de jeu voulant parcourir un damier suivant les directions horizontales, verticales et diagonales peut utiliser une méthode recevant dans les paramètres le coefficient à ajouter à la coordonnée `X`, et le coefficient pour la coordonnée `Y` au lieu de recevoir la direction voulue.

```

class Jeu
{
    static final int MaxX=10;
    static final int MaxY=10;
    class Offset
    { int offsetx;
      int offsety;
    }
    Offset offset_[]=new Offset[]{{1,0},{0,1},{1,1}};

    static final int Horizontale=0;
    static final int Verticale=1;
    static final int Diagonale=2;

    private void calcul(int dir)
    { int offx=Offset[dir].offsetx;
      int offy=Offset[dir].offsety;
      for (int x=0;x<MaxX;x+=offx)
        { for (int y=0;y<MaxY;y+=offy)
          { //...
            }
          }
        }
    }
    void calculAll()
    { calcul(Horizontale);
      calcul(Verticale);
      calcul(Diagonale);
    }
}

```

peut-être modifié comme ceci :

```

class Jeu
{ ...
  private void calcul(int offx,int offy)
  { for (int x=0;x<MaxX;x+=offx)
    { for (int y=0;y<MaxY;y+=offy)
      { //...
        }
      }
    }
  void calculAll()
  { calcul(1,0);
    calcul(0,1);
    calcul(1,1);
  }
}

```

Ce code est moins élégant que le précédent, mais le compilateur pourra l'optimiser au mieux. Il va générer un code semblable à

```

void calculAll()
{
    // calcul(1,0)
    for (int x=0;x<MaxX;++x)
    { //...
    }

    // calcul(0,1)
    for (int y=0;y<MaxY;++y)
    { //...
    }

    // calcul(1,1)
    for (int x=0;x<MaxX;++x)

```

```

    { for (int y=0;y<MaxY;++y)
      { //...
      }
    }
}

```

Les trois générations de la méthode `calcul()` sont très différentes.

Appeler une méthode `private` ou `final` dont un paramètre est une constante, permet d'optimiser grandement le code. Cela est utilisé par la machine virtuelle HotSpot™ de Sun.

Pour qu'une méthode soit générées en ligne par le compilateur, il faut qu'elle respecte certaines conditions. Elle doit être `private`, `static` ou `final`. Généralement, les compilateurs ne savent pas placer en ligne les méthodes `synchronized`. De même, le compilateur de Sun ne sait pas placer une méthode en ligne si elle possède des variables locales.

## Cast

Contrairement au C++, les conversions avec Java ne sont pas effectuées lors de la compilation. Cela à un coût lors de l'exécution. Evitez de convertir une variable plusieurs fois.

## Synchronize

Les méthodes `synchronized` sont plus lentes que les méthodes classiques. Les dernières générations de machines virtuelles améliore considérablement cela.

Pour éviter les méthodes `synchronized`, vous pouvez rédiger une classe immuable. Vous êtes alors certain qu'il n'y aura pas de conflit d'accès.

## Utiliser correctement la concaténation des String

La description de la méthode `toString()` explique comment utiliser correctement l'addition de deux chaînes de caractères. Cela permet de réduire le nombre de copies de tampon. Chaque fois que vous utilisez l'opérateur `+=` avec une chaîne, vous dupliquer le tampon.

En choisissant correctement la taille du `StringBuffer`, vous améliorez également les performances. En effet, lorsqu'un caractère ne peut plus être ajouté, l'instance doit créer un nouveau tableau plus grand, et recopier tous les éléments. Cela peut s'effectuer plusieurs fois lors d'un traitement. En initialisant la taille du tampon, on évite ces recopies.

Parfois, vous pouvez éviter les concaténation de String.

```

for (int i=0;i<3;++i)
{ img[i]=getImage(getCodeBase(),"G"+i+".gif");
}

```

Peux être optimisé ainsi :

```

img[0]=getImage(getCodeBase(),"G0.gif");
img[1]=getImage(getCodeBase(),"G1.gif");
img[2]=getImage(getCodeBase(),"G2.gif");

```

## Utilisez le dictionnaire de chaîne.

La classe `String` propose une méthode obscure : `intern()`. Cette méthode permet de mémoriser une chaîne de caractère dans un dictionnaire global. Elle retourne une référence sur l'instance présente dans le dictionnaire. Si une chaîne est déjà présente dans le dictionnaire lors de l'invocation de la méthode `intern()`, c'est celle-ci qui est retournée. A quoi cela sert-il ? A optimiser la mémoire et certaines comparaisons.

Pour comparer deux chaînes de caractères, il faut normalement utiliser la méthode `equals()`.

```

void f(String a,String b)
{
    if (a.equals(b))
    { ...
    }
}

```

Dans certaines situations particulière, on sait que les deux références, si les chaînes sont identiques, pointent sur la même instance. Il est alors possible de comparer les pointeurs.

```

void f(String a,String b)
{
    if (a==b)
    { ...
    }
}

```

Pour garantir cette situation, il faut demander à une chaîne de caractère d'être installé dans le dictionnaire. Les chaînes ayant une durée de vie équivalente à la durée de vie de l'application peuvent judicieusement être placés dans le dictionnaire.

```

public class MaClass
{
    private static String className_ ;
}

```

```

...
public static void calcul(String basename)
{
    className_=( "monpackage."+basename).intern();
}
}

```

La chaîne de caractère `className_` est le résultat d'un calcul simple, l'ajout d'un préfixe. La chaîne résultante possède une durée de vie importante. Pour économiser les instances, et optimiser la comparaison, la chaîne est installée dans le dictionnaire de la classe `String`.

Ce dictionnaire est utilisé par java pour identifier les noms de classes, de méthodes, etc. Il y a de grande chance que la chaîne `className_` soit déjà présente.

## Utilisez `static final` pour les constantes

Que les variables soient dans une interface ou dans une classe, le compilateur place la valeur des variables primitive `static final` en ligne.

```

interface Const
{
    static final int ID=123;
}
class Test
{
    public static void main(String[] args)
    {
        int i=Const.ID;
    }
}

```

La méthode `main()` est compilé en

```

public static void main(String[] args)
{
    int i=123;
}

```

Cela veut dire qu'il faut recompiler toutes les classes si on modifie la valeur d'une variable primitive `static final`. Ce n'est pas le cas si la constante est une référence ou si elle est initialisée dans les constructeurs.

## Ordonnez les variables

Contrairement aux langages compilés, l'accès aux variables dépend de leurs localisations. Une variable locale est plus rapide qu'une variable statique qui est plus rapide qu'une variable d'instance.

Les quatre premières variables locales utilisent un code optimisé. Il faut organiser les variables pour déclarer en premier les plus utilisées. Les bons compilateurs sont capables de détecter cela.

De même, il ne faut pas hésiter à dupliquer une variable d'instance dans une variable locale si elle est beaucoup utilisée dans un algorithme. Le résultat pourra retourner dans la variable d'instance lorsque tout sera terminé.

Vous pouvez également éviter des accès à des variables. Par exemple :

```
for (int i=0;i<limit;++i)
```

Peu devenir

```
for (int i=limit-1;--i>=0;)
```

La variable `limit` n'est consulté qu'une seule fois.

## Choisissez le bon type

Java propose plusieurs types primitifs plus ou moins précis. Il faut savoir que toutes les opérations effectuées sur les types `char` ou `byte`, commencent à convertir les variables en entier. Il est alors préférable d'utiliser le type `int` afin d'éviter les conversions successives.

Pour calculer un pourcentage, il est préférable de réorganiser l'expression pour éviter l'utilisation de flottant.

```
(int)(x*((float)cur/max))
```

devient

```
(x*cur)/max
```

Bien sûr, il faut que `x*cur` soit dans les limites d'un entier ( $2^{32}$ ).

## Évitez les récursivités

Un appel récursif n'est justifié que s'il existe au moins deux appels dans la même méthode. Le contre-exemple classique consiste à rédiger la méthode `fact()` avec une approche récursive.

```
static double fact(double i)
{ return i==1 ? 1 : i*fact(i-1);
}
```

Il n'y a qu'un seul appel récursif dans le corps de la méthode. Elle peut être rédigé plus efficacement à l'aide d'une boucle.

```
static double fact(double i)
{ double rc=1;
  for (;i>1;--i)
  { rc = rc * i;
  }
  return rc;
}
```

Une récursivité terminal est très facilement supprimable.

## Aidez le ramasse-miettes

Indiquez explicitement les références sur les objets à `null` lorsqu'ils ne sont plus nécessaires car cela évite au ramasse-miettes de faire une étude en profondeur pour identifier les instances orphelines. De même, il est préférable de casser les références croisées avant de perdre une instance. Cela permet au ramasse miette de libérer rapidement les instances.

Une liste composée de beaucoup d'objet (>1000) entraîne un travail excessif pour le ramasse-miettes lorsqu'il cherche à identifier les instances actives.

## N'appellez pas `System.gc()`

La méthode `System.gc()` permet de demander au ramasse-miettes de commencer à faire son travail. Si cette méthode est appelée trop souvent, le programme sera ralenti.

## Utilisez des tableaux à une dimension

Les tableaux à plusieurs dimensions sont traduits en mémoire par un arbre de tableau à une dimension. Par exemple, le tableau `int[3][4][5]` sera traduit en mémoire en :

- un tableau de tableau à trois éléments,
- trois tableaux de tableaux à quatre éléments,
- et douze tableaux d'entier à cinq éléments.

Cela donne au total seize tableaux pour soixante éléments. Dans l'hypothèse simpliste où la taille d'une référence est égale à la taille d'un entier et qu'il n'est pas nécessaire d'avoir des attributs supplémentaires pour la classe `Object`, cela donne une consommation mémoire de  $3+(3*4)+((3*4)*5)=75$ , soit une augmentation de 25% au minimum.

Lorsque les tableaux à dimensions multiples ont une forme rectangulaire ou cubique, il est préférable de ne construire qu'un tableau à une dimension et d'effectuer les calculs d'index nécessaire.

```
public class Tableau
{ private int[][][] array_=new int[3][4][5];

  public int get(int x,int y, int z)
  { return array_[x][y][z];
  }
}
```

devient

```
public class Tableau
{ private int[] array_=new int[3*4*5];

  public int get(int x,int y, int z)
  { return array_[x+y*4+z*(4*5)];
  }
}
```

Cela améliore les performances et réduit la consommation mémoire. Les tableaux à dimensions multiples doivent être utilisés, si et seulement si, chaque ligne peut avoir une taille différente.

Il est également préférable d'utiliser des tableaux à la place de `Vector` car cela économise les appels de méthodes.

## Conteneur

Les différents conteneurs de Java permettent d'indiquer une taille initiale. La choisir judicieusement peut faire économiser beaucoup de temps. Cela est particulièrement vrai pour la classe `Hashtable`. En effet, pour augmenter la taille du conteneur, l'algorithme doit recalculer toutes les valeurs de hash de ses éléments.

D'autre part, il faut adapter les algorithmes de calcul de la valeur de hash aux situations particulières à traiter. Par exemple, l'algorithme proposé par la classe `String` n'est pas adapté pour les URL car le format est toujours le même. Les valeurs de hash résultantes ne seront pas également réparties dans l'ensemble des valeurs possibles. Il est préférable d'utiliser un algorithme ad hoc pour bénéficier des dictionnaires.

Utilisez les `Enumeration` pour consulter un conteneur à la place des index.

```
Dictionary v=...
for (int i=0;i<v.size();+i)
{
    System.out.println(v.elementAt(i));
}
```

Doit être modifié en

```
Dictionary v=...
for (Enumeration i=v.elements();i.hasMoreElements();)
{
    System.out.println(i.nextElement());
}
```

Pour les vecteurs, le code utilise généralement un tableau. Dans ce cas, l'utilisation d'un index peut être envisagé.

## Exception

Les instructions `try-catch` n'entraînent pratiquement aucune perte de performance si aucune exception arrive. Le coût est significatif si une exception est généré.

Pour tirer avantage de cette implémentation, vous pouvez améliorer votre code en capturant les exceptions dans les situations rares.

```
if ((i>=0) && (i<array.length))
    x=array[i];
else
    // Error
```

Peut devenir

```
try
{ x=array[i];
} catch (ArrayOutOfBoundsException x)
{
    // Error
}
```

Cela permet d'économiser un test pour la majorité des situations.

## Graphisme

Utilisez les informations de clipping pour réduire le travail de la méthode `paint()`. Il n'est pas nécessaire de repeindre toute une fenêtre si seulement une petite partie est visible. Une erreur classique consiste à utiliser `repaint()` sans paramètre. Cela entraîne le rafraîchissement de toute la fenêtre alors qu'une seule partie est généralement nécessaire.

Une autre erreur consiste à repeindre une fenêtre plusieurs fois alors qu'une seule suffit.

Au contraire, il peut être intéressant de regrouper les zone de clipping pour limiter le nombre d'invocation à la méthode `paint()`. Par exemple, dans une ergonomie de type maître-détail, une partie de l'écran propose une liste d'instances. En sélectionnant une instance, un panneau de détail doit être rafraîchi. Il est préférable de demander le rafraîchissement de tous le panneau plutôt que le rafraîchissement de chaque champ dans le panneau.

## Entré/Sortie

Utilisez `BufferedInputStream` et `BufferedOutputStream` pour optimiser les accès aux I/O. Cette modification est très facile car les flux s'enchaînent les uns aux autres.

Si vous utilisez les sockets java, essayez d'appeler `java.net.Socket.setTcpNoDelay(true)`. Cela permet de supprimer l'algorithme décrit dans le RFC 896 qui conserve la bande passante en réduisant le nombre de segments émit. Lorsque l'algorithme est éteint, les données sont envoyées plus rapidement, mais utilisent plus de bandes passantes. Les tests effectués par IBM indique une amélioration de 10 à 50 fois plus rapide.

Ecrire une information sur la console prend beaucoup de temps et de ressources et n'est généralement pas nécessaire.

## Serializable

Il n'est pas nécessaire de sérialiser les informations de cache. Les caches sont, par définition, recalculables à l'aide des autres attributs. Il faut déclarer les caches `transient`.

Le traitement par défaut de la sérialisation utilise l'introspection pour analyser les attributs à sauver dans le flux. Cela prend du temps. Si vous rédigez les méthodes `readExternal()` et `writeExternal()` vous-même, vous pouvez optimiser le code.

```
public class MaClass implements Externalizable
{
    private int i;
    private String s;
    private long[] array;
    private Point p;
```

```

public void writeExternal(ObjectOutput out) throws IOException
{
    out.writeInt(i);
    out.writeObject(s);
    out.writeObject(array);
    out.writeObject(p);
}
public void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException
{
    i=out.readInt();
    s=(String)out.readObject();
    array=(long[])out.readObject();
    p=(Point)out.readObject();
}
}

```

Vous pouvez également sauver le contenu de certains objets à la place des objets eux-mêmes. Cela permet d'optimiser le code et de réduire la taille du flux de la sérialisation.

```

public void writeExternal(ObjectOutput out) throws IOException
{
    ...
    out.writeInt(p.x);
    out.writeInt(p.y);
}
public void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException
{
    ...
    p=new Point(out.readInt(),out.readInt());
}

```

## JNI

Groupez les appels aux méthodes natives pour limiter le travail de la machine virtuelle. Le passage de la machine virtuelle vers la DLL JNI est coûteux en performance.

## 2. OPTIMISER LA MEMOIRE

Pour réduire la place mémoire nécessaire à l'application, il faut réduire le nombre d'instance nécessaire. Pour qu'une application fonctionne correctement, il faut un espace mémoire nécessaire à l'ensemble des objets, augmenté d'un tampon. Une taille raisonnable et d'avoir en réserve de 30 à 40% après le passage du ramasse-miettes. Avoir une taille mémoire trop importante n'est pas efficace. Cela consomme trop de mémoire pour les autres applications, et demande au ramasse-miettes d'effectuer plus de travail.

### Réduisez la taille des instances

- Utilisez si possible des variables statiques afin d'éviter les redondances dans les instances.
- N'utilisez pas de cache.
- Partagez les informations communes à plusieurs instances dans une instance de corps. Vous pouvez dupliquer l'instance corps lors d'une modification.

```

public class MaClass
{
    static class Body
    { boolean shared_=false;
      //...
      Body()
      {
      }
      Body(Body x)
      { // ...
      }
      boolean getShared()
      { return shared_;
      }
      synchronized void setShared(boolean x)
      { shared_=true;
      }
      synchronized void setAttr(int x)
      { //...
      }
      public synchronized Object clone()
      { return new Body(this);
      }
    }
}

```

```

    }
}

private Body body_;

public MaClass()
{ body_=new Body();
}
public MaClass(MaClass x)
{
    body_=x.body_;
    body_.setShared(true);
}
public Object clone()
{
    return new MaClass(this);
}
public void setAttr(int x)
{
    synchronized(body_)
    {
        if (body_.getShared())
        { body_=(Body)body_.clone();
        }
    }
    body_.setAttr(x);
}
}
}

```

C'est l'approche utilisée par la classe `StringBuffer` pour partager son tampon avec l'instance `String`.

## Chargement paresseux

Certains objets ne sont pas toujours nécessaires. La méthode `get()` peut alors les construire lors du premier appel. Par exemple, un Beans doit pouvoir enregistrer des observateurs. Pour cela, il utilise une instance `java.beans.PropertyChangeSupport`. Si le beans est utilisé sans que personne ne soit intéressé par ses événements, il n'est pas nécessaire de construire l'instance.

```

import java.beans.*;

public class MonBeans
{ private transient
    PropertyChangeSupport propertyChange_;

    private
    java.beans.PropertyChangeSupport getPropertyChange()
    { if (propertyChange_ == null)
        {
            propertyChange_ = new PropertyChangeSupport(this);
        }
        return propertyChange_;
    }

    protected void firePropertyChange(String propertyName,
                                      Object oldValue,
                                      Object newValue)
    { if (propertyChange_!=null)
        propertyChange_.firePropertyChange(propertyName,
            oldValue,
            newValue);
    }

    public synchronized void addPropertyChangeListener(
        PropertyChangeListener listener)
    { getPropertyChange().
        addPropertyChangeListener(listener);
    }

    public synchronized void removePropertyChangeListener(
        PropertyChangeListener listener)
    { getPropertyChange().
        removePropertyChangeListener(listener);
    }
}

```

Si aucune instance ne s'est enregistrée, il n'est pas nécessaire de propager les événements. Attention, la construction paresseuse peut être dangereuse en utilisation multitâche.

## Limitez la durée de vie des instances

La durée de vie d'une instance locale dépend de la durée du traitement. Il peut être judicieux de forcer les références à `null` avant d'effectuer une étape longue.

```
public void doCompile()
{
    // 1. Parse
    Lexicale lex=parse();
    ...
    lex=null;

    // 2. Optimise
    optimize();

    // 3. Compile
    compile();
}
```

Cette situation peut arriver dans des cas plus subtils. Par exemple, lors de l'utilisation d'une variable locale déclarée dans un bloc.

```
public void run()
{
    for (;;)
    {
        Command cmd;
        cmd=waitNextCmd();
        cmd.go();
    }
}
```

La méthode `waitNextCmd()` attend la présence d'une commande avant de retourner à l'appelant. Lorsque le premier cycle est effectué, la variable `cmd` obtient un objet. Lors du deuxième cycle, la valeur de `cmd` ne change pas tant que `waitNextCmd()` ne retourne rien. L'instance `Command` précédente ne peut pas être détruite. Il faut explicitement forcer la valeur à `null` pour permettre au ramasse-miettes de libérer la `Command` lors de l'appel à `waitNextCmd()` qui peut être très long.

```
public void run()
{
    for (;;)
    {
        Command cmd;
        cmd=null;
        cmd=waitNextCmd();
        cmd.go();
    }
}
```

Pour utiliser un code plus clair, la valeur `null` doit être affectée à la fin d'un cycle.

```
public void run()
{
    for (;;)
    {
        Command cmd;
        cmd=waitNextCmd();
        cmd.go();
        cmd=null;
    }
}
```

De même, libérez les instances statiques qui ne sont plus nécessaires.

## Limitez le nombre de tâches

Chaque tâche requiert la création d'une pile native. La taille de la pile est contrôlée par le paramètre `-ss`. Par défaut, elle est de 32KB sur les PC. Pour une application ayant vingt tâches, il faut 20\*32K ou 640KB. Limiter le nombre de tâches nécessaires à une application permet de réduire considérablement la taille mémoire et améliore les performances.

De même, lorsqu'une instance `Thread` est créée mais non démarré (pas d'appel à la méthode `start()`), une référence vers la tâche est gardée par l'instance `ThreadGroup` associé. Cette tâche ne peut pas être récupéré. Elle devient un fantôme qui ne sera pas tué. Le seul moyen pour supprimer l'instance `Thread` est d'appeler la méthode `start()` suivi de la méthode `stop()`. Attention, en JDK 1.2, les méthodes `stop()`, `suspend()` et `resume()` sont `deprecated`. Il faut utiliser `interrupt()` et coder la méthode `run()` en conséquence.

En général, une instance `Thread` doit rapidement être démarrée.

## Recyclez les instances

Voir plus haut

## Inversez l'ordre d'exécution d'une boucle

Voir plus haut

### Utilisez le dictionnaire de chaîne.

La classe `String` propose une méthode obscure : `intern()`. Cette méthode permet de mémoriser une chaîne de caractère dans un dictionnaire global. Elle retourne une référence sur l'instance présente dans le dictionnaire. Si une chaîne est déjà présente dans le dictionnaire lors de l'invocation de la méthode `intern()`, c'est celle-ci qui est retournée. A quoi cela sert-il ? A optimiser la mémoire et certaines comparaisons.

Pour comparer deux chaînes de caractères, il faut normalement utiliser la méthode `equals()`.

```
void f(String a,String b)
{
    if (a.equals(b))
    { ...
    }
}
```

Dans certaines situations particulière, on sait que les deux références, si les chaînes sont identiques, pointent sur la même instance. Il est alors possible de comparer les pointeurs.

```
void f(String a,String b)
{
    if (a==b)
    { ...
    }
}
```

Pour garantir cette situation, il faut demander à une chaîne de caractère d'être installé dans le dictionnaire. Les chaînes ayant une durée de vie équivalente à la durée de vie de l'application peuvent judicieusement être placé dans le dictionnaire.

```
public class MaClass
{
    private static String className_ ;
    ...
    public static void calcul(String basename)
    { className_=( "monpackage."+basename).intern();
    }
}
```

La chaîne de caractère `className_` est le résultat d'un calcul simple, l'ajout d'un préfixe. La chaîne résultante possède une durée de vie importante. Pour économiser les instances, et optimiser la comparaison, la chaîne est installé dans le dictionnaire de la classe `String`.

Ce dictionnaire est utilisé par java pour identifier les noms de classes, de méthodes, etc. Il y a de grande chance que la chaîne `className_` soit déjà présente.

### Utilisez `static final` pour les constantes

Voir plus haut

### Choisissez le bon type

Java propose différents types primitifs plus ou moins précis. Ils se différencient par la place prise en mémoire et les valeurs minimums et maximales autorisées. Vous pouvez utiliser les type `char` ou `byte` dans des instances étant présentes en de nombreux exemplaires dans la mémoire. Le code nécessaire à la manipulation de ces types est plus important (le fichier `.class` est plus grand), mais cela est compensé par les économies réalisées sur chaque instance. De même, utilisez `float` à la place de `double` si la précision nécessaire le permet.

### Utilisez des tableaux à une dimension

Voir plus haut

### Libérez les ressources

La méthode `finalize()` peut s'occuper de libérer les ressources nécessaires à un objet. Cette méthode est appelée par le ramasse-miettes lorsque l'instance n'est plus nécessaire. Cela peut intervenir bien après la perte de l'objet. Il est préférable de gérer explicitement les ressources par un appel à `close()`, par exemple lorsqu'un flux n'est plus nécessaire.

### Graphisme

La classe `Image` construit un tampon en mémoire adapté à l'écran utilisé. Ce tampon est copié par le système directement dans la mémoire vidéo. Il est supprimé de la mémoire lorsque la méthode `flush()` est appelé. Il sera reconstruit lorsque l'image sera de nouveau nécessaire. Une image à besoin de deux fois sa taille en mémoire : une fois pour l'objet et une fois pour le système. Une image de 1 Mo utilise réellement 2 Mo.

Si vous construisez beaucoup d'images, mais que vous ne détruisez pas les tampons, le mémoire utilisé peut être très importante. Par exemple, un algorithme de double-buffering permet de dessiner une fenêtre complexe dans la mémoire avant de l'afficher. Cela permet de cacher

les étapes de la construction. Le tampon peut prendre jusqu'à deux fois la taille de l'écran ! Il est nécessaire de libérer la ressource immédiatement après l'avoir utilisée.

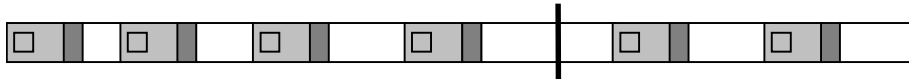
Swing propose automatiquement un double-buffering. Il faut faire très attention à la conséquence mémoire que cela entraîne.

### 3. OPTIMISER LE CHARGEMENT

Suivant l'architecture de déploiement, le chargement des applets est plus ou moins efficace. Nous allons étudier les différentes démarches possibles et les moyens pour les améliorer.

#### 3.1 Le chargement d'une applet

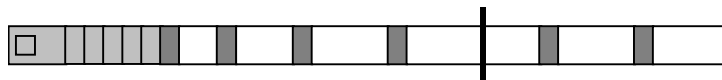
Lors du chargement d'une applet ou d'une application, il y a une étape d'initialisation pendant laquelle l'utilisateur doit patienter. Pour exécuter une classe dans la machine virtuelle, il faut procéder à trois étapes : charger le fichier `.class`, installer la classe et exécuter le code. Si l'applet n'est pas présente dans une archive, le chargement peut être symbolisé comme ceci.



La flèche du temps s'écoule de la gauche vers la droite. La barre verticale indique la fin de l'initialisation de l'applet. La couleur gris clair représente le temps de chargement du fichier `.class` ; le petit carré indique le temps de connexion au serveur HTTP ; le gris foncé indique l'installation de la classe ; la couleur blanche indique l'exécution du programme.

Java utilise une installation paresseuse. Une classe est chargée puis installée lorsque c'est nécessaire. Cela évite d'avoir un grand exécutable en mémoire alors qu'une partie seulement est utilisée. Au fur et à mesure de l'exécution du programme, de plus en plus de classes sont chargées. Le temps consacré à l'exécution réelle augmente. Le démarrage d'une portion gris claire demande un temps plus long car il faut se reconnecter au serveur HTTP.

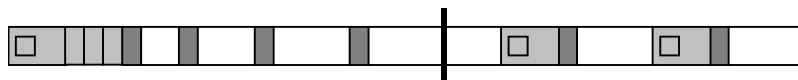
Si l'applet est présente dans une archive, les chargements des fichiers `.class` sont regroupés avant l'installation de la première classe.



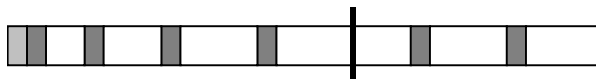
L'installation des classes continue à être répartie tout au long de l'exécution du programme. On économise avec cette architecture les temps de reconnexion au serveur HTTP et l'on réduit le temps de chargement des fichiers `.class` car ceux-ci sont compressés, mais on doit attendre le chargement de toute l'archive avant de pouvoir utiliser l'applet.

Les classes n'ont pas toutes le même statut. Certaines devront être chargées, d'autres non. Les classes indispensables à l'initialisation de l'applet seront, quoi qu'il arrive, installées dans la machine virtuelle. D'autres classes ont le même statut car l'utilisateur devra passer par certains chemins de l'application. Par contre, les classes s'occupant de l'impression ne sont nécessaires que si l'utilisateur le demande. De même, les classes des exceptions ne sont nécessaires que si l'une d'entre elles se produit.

La règle des 20/80 peut s'appliquer au chargement des applets importantes : 20% des classes sont obligatoires, 80% des classes peuvent être chargées à la demande. Vous pouvez améliorer le déploiement de votre applet en ne plaçant dans l'archive que les classes strictement nécessaires et les classes devant être signées. Les autres restent dans des fichiers `.class` séparés. Les fichiers `.class` supplémentaires seront compressés à la volée par les modems lors du chargement.



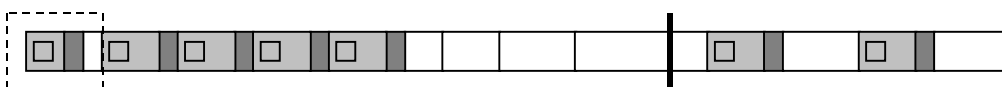
Les applets java sont généralement chargés sur le poste client à la demande. Le navigateur peut garder en cache l'archive ayant permis de lancer l'applet, mais, dans certaines conditions, il peut la supprimer. Si l'archive est trop importante pour le cache (>64k), et devrait forcer à supprimer trop de fichiers déjà présents dans le cache, elle n'est pas mémorisée. Cela peut entraîner un rechargement de l'archive à chaque consultation de la page. Dans ce cas, il peut être préférable d'installer l'archive sur le poste client. Netscape et Microsoft proposent des solutions pour cela. Le JDK 1.3 également. Lorsque l'applet est installée sur le poste client, le temps d'ouverture de l'archive est négligeable.



#### 3.2 Améliorer le chargement

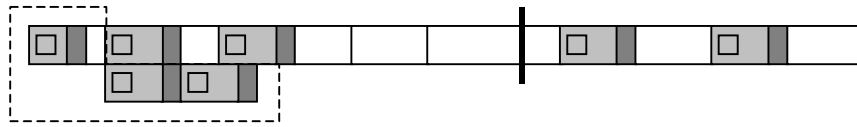
Pour améliorer l'expérience de l'utilisateur, il faut regrouper le temps d'installation des classes en affichant une barre de progression.

Si on n'utilise pas d'archive, cela donne :



La section encadrée de pointillés représente l'additif à l'applet pour permettre le pré-chargement des classes.

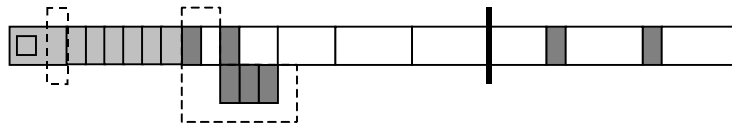
Une partie importante du chargement est passive. Il faut attendre les paquets du réseau. Pour utiliser les temps morts lors du chargement d'un fichier `.class`, il faut lancer une tâche s'occupant de charger les autres classes.



On bénéficie ainsi de deux connexions au serveur HTTP simultanément. Les navigateurs utilisent généralement quatre connexions avec le serveur pour charger une page HTML et ses images. Le `ClassLoader` de java ne peut pas procéder ainsi car la résolution des classes doit être synchrone. Utiliser une autre tâche pour le chargement des classes permet d'améliorer sensiblement le chargement de l'applet.

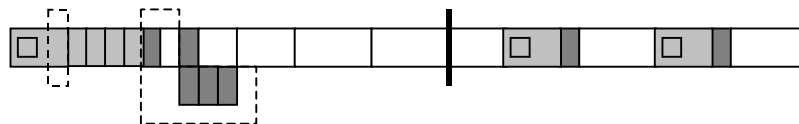
Lorsque l'applet s'affiche, toutes les conditions sont réunies pour une exécution rapide. Un programme de démarrage va se charger d'afficher la barre de progression, de charger et d'installer les classes nécessaires à l'application. Il doit être très petit et ne nécessiter que peu de classes (<15ko). Cela permet, lors d'une connexion avec un modem, d'avoir rapidement l'affichage de la barre de progression.

Avec l'applet dans une archive, nous avons :



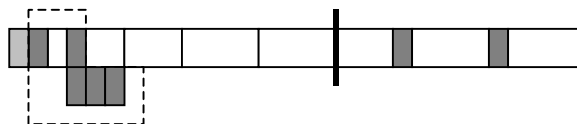
Les fichiers `.class` sont compressés. L'utilisateur attend le chargement de toutes les archives avant de voir la barre de progression. Celle-ci s'exécute rapidement. L'expérience de l'utilisateur est améliorée, mais ce n'est pas la situation idéale.

Si l'archive ne possède que les classes strictement indispensables au démarrage de l'applet, le schéma devient :



Le temps de chargement de l'archive est réduit. L'utilisateur obtient un chargement efficace.

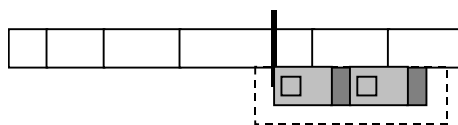
Si l'application est installée sur le poste utilisateur, le temps de chargement est négligeable. Le schéma devient :



La barre de progression avance rapidement. Cela est cohérent avec l'initialisation de l'applet.

### 3.3 Anticiper l'utilisation

Une fois l'application démarrée, on peut lancer une tâche s'occupant de charger les classes supplémentaires qui seront nécessaires par la suite.



Cela permet de bénéficier des temps morts de l'utilisateur pour continuer l'installation de l'applet. Cette approche est possible car la machine virtuelle décharge les classes lorsque le `ClassLoader` correspondant est déchargé.

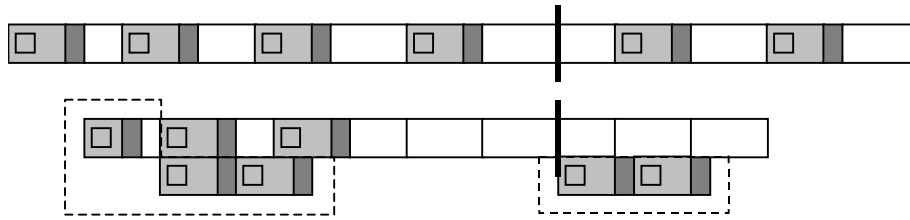
Les classes ont trois statuts différents : indispensable, préférable et optionnel.

- Les classes indispensables seront présentes dans l'archive de l'applet ;
- les classes préférables seront chargées en tâche de fond après l'initialisation de l'applet
- les classes optionnelles seront chargées à la demande par le navigateur.

### 3.4 Architecture globale

Nous sommes partis d'un chargement morcelé et long pour aboutir à un chargement structuré et optimisé. Plusieurs architectures peuvent bénéficier de toutes ces technologies.

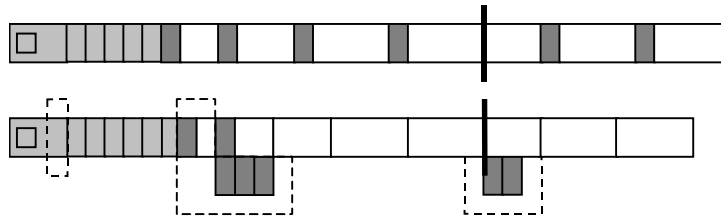
Dans le cas d'un chargement des `.class` non présent dans une archive, nous améliorons le chargement comme ceci :



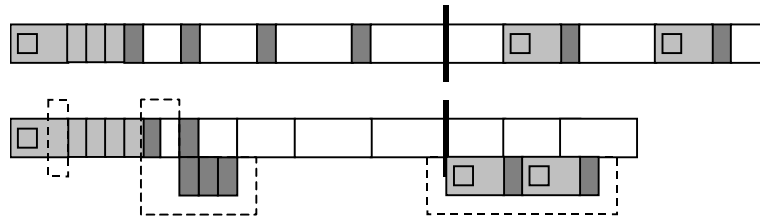
Le schéma du dessus représente le chargement classique de l'applet. Le schéma du dessous représente la version optimisée.

Si toute l'applet est dans une archive, on peut utiliser cette architecture, mais l'expérience de l'utilisateur n'est pas complètement satisfaisante. Tout dépend de la proportion de classes strictement nécessaire par rapport au nombre de classe totale.

Si l'archive possède essentiellement des classes indispensables au démarrage de l'applet, on peut utiliser la barre de progression lors du chargement. Par exemple, si l'applet utilise plus de 80% des classes de l'archive pour s'initialiser.

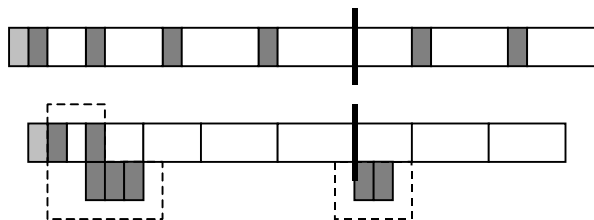


Sinon, l'archive ne doit posséder que les classes nécessaires à l'initialisation.



L'archive est de taille réduite et le chargement des classes supplémentaires s'effectue en tâche de fond.

Dans le cas d'une applet installé sur le poste de l'utilisateur, le chargement est amélioré comme ceci :



La barre de progression avance rapidement et une tâche de fond s'occupe d'installer les classes supplémentaires.

Pour implanter ces architectures, il faut connaître l'ensemble des classes nécessaire à l'initialisation d'une applet. Cette information est difficile à obtenir. Avec l'aide d'un `ClassLoader`, des utilitaires peuvent construire cette liste.

### 3.5 Les autres techniques

D'autres techniques permettent de réduire le temps de chargement.

#### Choix du format

Java propose d'utiliser une archive (extension `.jar`) pour regrouper un ensemble de classe. Cela permet d'effectuer une seule requête HTTP pour exécuter l'applet. Chaque fichier dans l'archive est compressé individuellement. Par exemple, s'il y a dix fichiers identiques de deux caractères dans l'archive, l'algorithme de compression ne pourra pas intervenir. Au contraire, le format `.cab`, proposé par Microsoft effectue une compression sur l'ensemble de l'archive. Ainsi, les dix fichiers seront compressés. Le format `.cab` est plus efficace que le format `.jar`.

#### Utiliser l'optimisation du compilateur

Pour optimiser le temps de chargement d'une application java, il faut réduire la taille du code généré. Utilisez `javac -O`. Si votre programme utilise souvent des méthodes en ligne, la taille du code généré peut être réduit en évitant les différents appels. Il faut comparer le résultat avec et sans l'option.

## Réduire la hiérarchie

Une classe possédant beaucoup de super classe est plus lent à charger. Il faut en effet obtenir tous les fichiers `.class` des supers classes. Vous pouvez éventuellement éviter une sous-classe à l'aide d'une variable d'état.

## Réduire le nombre de classe nécessaire

Une classe non utilisée peut être chargé par le navigateur afin de vérifier la validité d'une autre classe. Lorsqu'une classe est installée dans la machine virtuelle, celle-ci utilise un algorithme de vérification. Le format du fichier `.class` est vérifié et tous les chemins de toutes les méthodes sont analysés.

```
class BaseClass
{ public void baseMethode()
  {
  }
}

class AutreClass extends BaseClass
{
}

class MaClass
{
  public void f(AutreClass cl)
  {
    cl.baseMethode();
  }
}
```

La méthode `f()` utilise la méthode `BaseClass.methode()` à partir d'une instance `AutreClass`. Même si `f()` n'est jamais appelé, la classe `AutreClass` sera chargé par le navigateur afin de vérifier la validité de l'appel. La machine virtuelle doit vérifier si `AutreClass` hérite de `BaseClass`. Vous pouvez vérifier cela en sélectionnant le niveau quatre de déverminage dans la console de Netscape. Cela trace les chargements de chaque classe.

Pour éviter le chargement de la classe `AutreClass` tant que l'on n'utilise pas la méthode `f()`, il faut ajouter une classe intermédiaire pour toutes les méthodes appelant une méthode héritée d'une instance ou utilisant une conversion implicite d'une sous-classe vers une super classe.

```
class MaClass
{
  public void f(AutreClass cl)
  {
    class Bridge extends MaClass
    {
      public void f(AutreClass cl)
      {
        cl.baseMethode();
      }
    };
    new Bridge().f(cl);
  }
}
```

La classe `MaClass` ne propose plus d'utilisation directe d'une instance `AutreClass`. On peut regrouper tous ces appels dans une classe interne.

```
class MaClass
{
  class Bridge extends MaClass
  {
    public void f(AutreClass cl)
    {
      cl.baseMethode();
    }
    public void g(...)
    { ...
    }
  };
  public void f(AutreClass cl)
  {
    new Bridge().f(cl);
  }
  public void g(...)
  {
    new Bridge().g(...)
  }
}
```

## Ne pas réinventer les API

Java propose un certain nombre d'API pré-installé sur le poste du client. Pour réduire la taille du code, il est préférable d'utiliser ces APIs plutôt que d'en rédiger une nouvelle version.

## Utilisez l'héritage

Utilisez l'héritage pour factoriser un maximum de code. Cela évite les redondances et réduit le risque d'erreurs. Par exemple, séparez le code non portable dans des sous-classes.

## Séparer le code commun

Placez le code utilisé dans différent endroit dans une méthode éventuellement `private`. C'est une approche inverse des méthodes `inline`. Si le code redondant est important, cela peut réduire la taille du fichier `.class`.

## N'initialisez pas les gros tableaux

Si vous initialisez un gros tableau dans une seule expression, cela génère beaucoup de code. Il peut être préférable d'écrire une `String` qui sera analysée pour construire le tableau.

## Utiliser correctement la concaténation des `String`

La description de la méthode `toString()` explique comment utiliser correctement l'addition de deux chaînes de caractères.

## `synchronized`

Le code généré pour une méthode `synchronized` est plus court que pour l'appel de `synchronized(this)`

```
public MaClass
{ synchronized void f()
  {
    ...
  }
}
```

est plus court que

```
public MaClass
{ void f()
  {
    synchronized(this)
    {
      ...
    }
  }
}
```

## Utilisez des noms de classe et de méthode courts

Les noms des classes et des méthodes nécessaires à une instance sont indiqués dans chaque fichier `.class`. En réduisant la taille des noms, vous réduisez la taille du dictionnaire de la classe. Des outils permettent d'effectuer ces optimisations automatiquement.

## Supprimer les méthodes inutiles

De nombreuses méthodes ne sont présentes que pour le déverminage. Par exemple, la plupart des classes possèdent une méthode `main()` afin de pouvoir effectuer les tests unitaires. Ces méthodes ne sont pas nécessaires à l'application. Vous pouvez séparer les méthodes de tests dans une autre classe, suffixé par exemple de `UnitTest`. Ainsi, il est facile de supprimer tous les fichiers `*UnitTest*.class` de l'archive avant de la publier sur le réseau.

## Optimisez les fichiers `.class`

Des utilitaires permettent de transformer le nom des classes et des méthodes pour n'avoir qu'un ou deux caractères, de supprimer les codes morts ou les méthodes non utilisées. Cela permet de réduire notablement la taille des fichiers `.class` (Cf. <http://www.alphaworks.ibm.com/formula/jax> ou <http://www.codework.com/dashO/product.html> )

## 4. CONCLUSION

- Ne pas optimiser si on ne sait pas si c'est nécessaire
- Si vous voulez optimiser, commencez par modifier l'algorithme.
- Avant d'effectuer une optimisation de bas niveau, mesurez le code pour voir où cela est nécessaire. Analysez le par la suite pour voir les impacts des modifications.