Bibliographie Java

La documentation de référence sont les livres de la "Java series" publiés dans diverses éditions (Addison Wesley, ...) :

- The Java Application Programming Interface, Vol 1 (core packages), Vol 2 (Window Toolkit and applets) : J. Gosling, F.Yellin, traduits en français
- The Java Programming Language : K. Arnold, J. Gosling ; ed Addison Wesley traduit en français.
- The Java Tutorial : M. Campione, K. Walrath
- The Java Virtual Machine Specification : T. Lindholm, F. Yellin.

Java in a nutshell : David Flanaghan ; ed O'Reilly traduit en français

Sur Java 1.1 : Java 1.1 developer's Handbook : Philip Heller et Simon Roberts ; ed sybex

Teach yourself Java in 21 days : Laura Lemay, Charles L.Perkins ; ed Sams.net traduit en français ed S&SM "Le programmeur Java"

`http://java.sun.com/`

groupe de news `comp.lang.java.*`, `fr.comp.lang.java`

la documentation des classes Java 1.0 à partir de :
http://java.sun.com/JDK-1.0/api/packages.html

Une excellente revue sur Java
http://www.javaworld.com/

La FAQ (toute petite) se trouve a
http://java.sun.com/products/jdk/faq.html
Une autre meilleure
http://www.digitalfocus.com/faq/

Un grand site Java est :
http://www.gamelan.com/

Le tutorial Java :
http://java.sun.com/books/Series/Tutorial/index.html

Un cours Java
http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/JAVAF/index.html

Un autre cours Java avec exercices corrigés :
http://cedric.cnam.fr/~farinone/Java2810

Les concepts

Présentation

Java : langage orienté objet inspiré de :
- Smalltalk, Objective C (utilisation des références d'objet, polymorphisme dynamique, bibliothèques de classes, garbage collector, ...)
- Ada (paquetage, exception...)
dans sa sémantique
- C++ dans sa syntaxe.

Avec environnement (classes de base + debugger)

permet d'écrire des programmes orientés objets puissants.

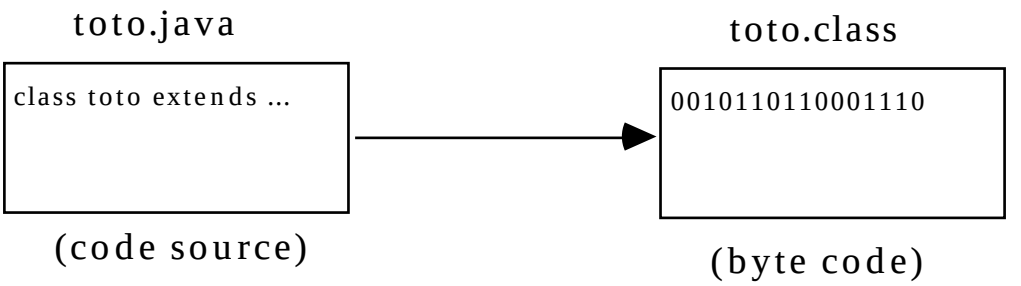
Fournit des API pour les interfaces graphiques, le son, le multithreading, la programmation réseau, ...

permet de construire du code chargeable par Internet et interprétable sur de multiples architectures => une partie d'Internet est banque de programmes

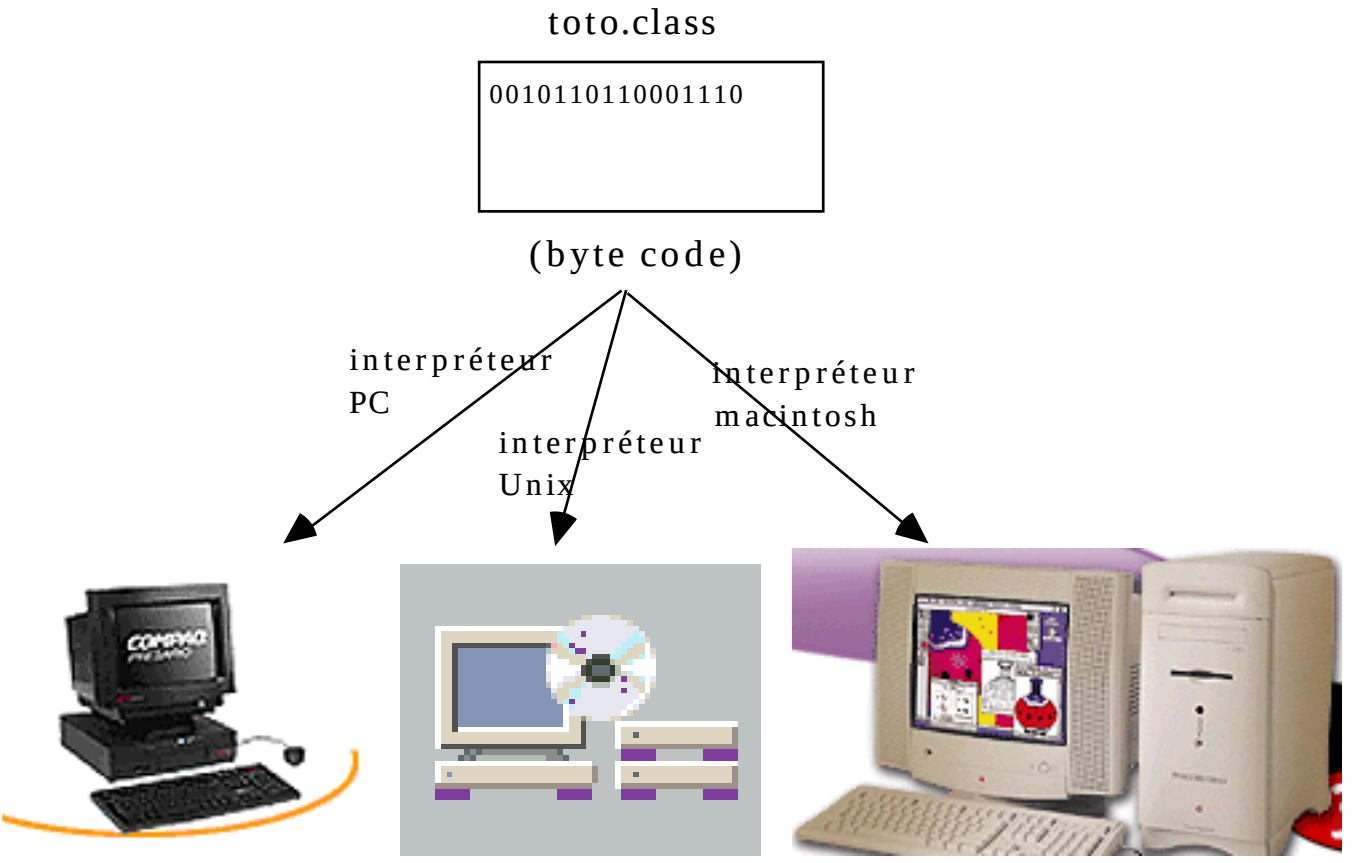
La documentation est en ligne à
`http://java.sun.com` ou
`http://www.javasoft.com/`

Présentation (suite)

d'abord compilé (on obtient un byte-code)



puis interprété



le JDK

L'environnement minimal pour faire des programmes Java est gratuit est disponible en ligne : c'est le Java Development Kit (JDK). Il contient :
- un compilateur
- un interprêteur
- un environnement de développement (outils, utilitaires pour la programmation, ...)

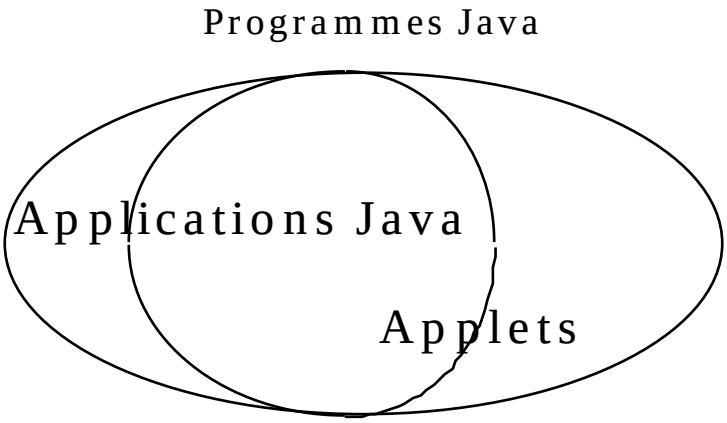
Le JDK 1.1 existe sur :
- Solaris 2.4, 2.5 SPARC et 2.5 x86
- Windows 95, NT

voir à
http://www.javasoft.com/products/jdk/1.1/index.html
mais aussi sur :
les Unix : Linux, FreeBSD, AIX 4.1.3, OSF1, SunOS, ...
OS/400, Windows 3.1, Amiga 0S, BeOS, OS/2
voir
http://www.javasoft.com/cgi-bin/java-ports.cgi
donc Java est indépendant des architectures.

Programme Java = applications indépendantes et/ou applets

Un programme Java est ou bien :
- un programme à interpréter sans le Web : application indépendante (standalone)
- un programme chargeable par Internet à l'aide d'un client Web et exécuté en local à l'aide de l'interpréteur intégré dans le client Web : les applets.

Il peut être les 2.



Java : la compilation

Sous Windows et Unix

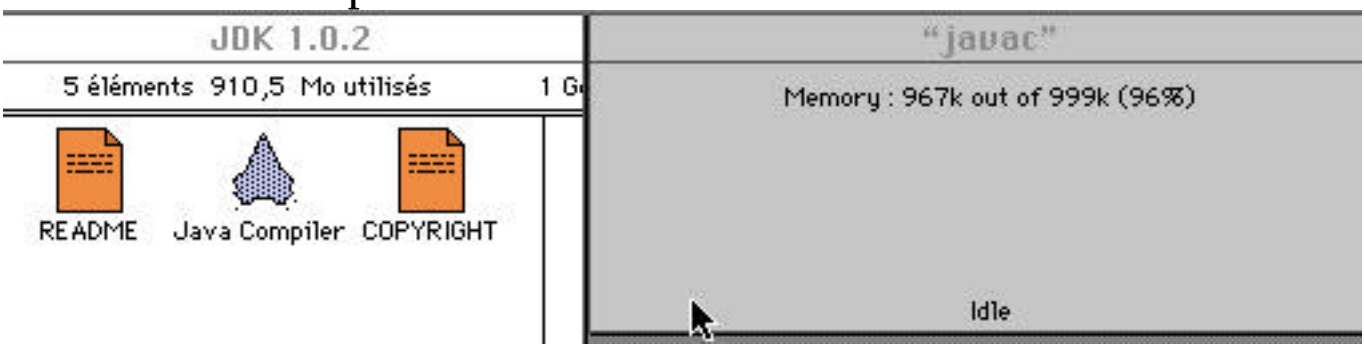
C'est le programme javac.

construction

Sur macintosh

double cliquer sur l'icône

Puis ouvrir File | Compile File et cherche le fichier a compiler.



L'exécution d'une application Java

on lance l'interpréteur sur le fichier byte-code contenant la méthode `main()`.

exemple

fichier `HelloWorld.java`

```
class HelloWorld {
    public static void main (String args[]) {
        System.out.println("Bonne journée Java");
    }
}
```

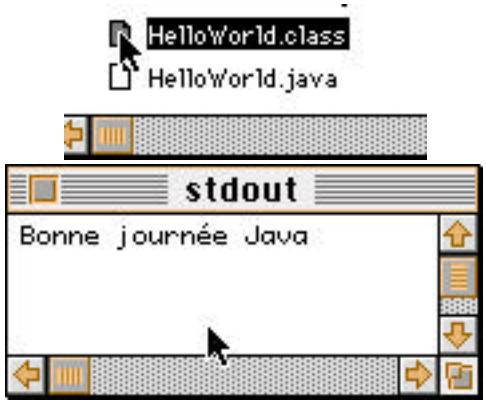
Sous Windows et Unix

Exécuter :

```
% java HelloWorld
Bonne journée Java
%
```

Sur macintosh

Exécuter :
en double cliquant sur le fichier .class créé contenant la méthode main.



exemples d'application Java

Les produits industriels

voir Java in the real world à :
http://www.javasoft.com/nav/used/index.html
http://www.javasoft.com/nav/used/prevstories.html
Kodak (Photo CD on the Web), accès aux bases de données, application météorologique (Université du Michigan), ...

Les services postaux américains
Le centre médical UCLA (accès aux informations et imagerie médicales)
Le robot NASA sur Mars en Juillet 1997
...

L'environnement Java

Les outils autour de Java sont des programmes Java. Par exemple :
jdb (le débogueur Java), javac, hotjava, javadoc, •••

Java : langage ...

orienté objet et manipule les notions objets : classes, encapsulation donc masquage de données, objets, héritage, polymorphisme.

Java 1.0 fournit un ensemble de classes permettant de construire et de manipuler :
- les applets
- les interfaces graphiques.
- les manipulations d'images
- les entrées sorties
- la programmation réseau TCP et UDP.
- les structures de données fondamentales (table de hachage, générateur de nombre aléatoire, pile, liste chaînée à accès direct et hétérogène, mini analyseur lexical)

ceci pour la version Java 1.0.

Il y en a d'autres supplémentaires en Java 1.1.

garbage collector (=> ce n'est plus au programmeur de gérer la désallocation mémoire)

bibliothèque de classes graphiques : classes d'objets conteneur, classe d'objets "contrôle" (boutons, labels, ...)

bibliothèque de classes audio et image.

L'environnement de base (suite)

On trouve la documentation en ligne à :
pour Java 1.0
http://www.javasoft.com/products/jdk/1.0.2/api/
pour Java 1.1 à
http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html

Au CNAM (merci Gersan) à :
http://deptinfo.cnam.fr/Ressources/Java/api.jdk-1.0.2/index.html pour Java 1.0
http://deptinfo.cnam.fr/Ressources/Java/jdk-1.1/docs/api/packages.html pour Java 1.1.
Attention : Ces deux URL ne sont accessibles qu'à partir des machines du CNAM.

Java : notions de base

Les structures de données fondamentales (tableaux, chaîne de caractères, ...) sont des objets avec leurs méthodes associées voire leur exception (indice non valide, etc.)

```
String chaine1 = "bonjour";
String chaine2;

chaine2 = chaine1 + " a tous" // surdéfinition de +



int a[ ] = new int [ 10 ];

a [5] = 1;
a [1] = a[0] + a [2];

a[-1] = 4; // lève une exception
           // ArrayIndexOutOfBoundsException
System.out.println (a.lenght);  // donnée associée aux tableaux
```

Les types primitifs

Ces types prédéfinis (dits "primitifs") sont les suivants et ont une taille fixe quelle que soit l'architecture de la machine. Ce sont :

Type	taille (en bits)	format
byte	8	complément à 2
short	16	complément à 2
int	32	complément à 2
long	64	complément à 2
float	32	IEEE 754
double	64	IEEE 754
char	16	Unicode
boolean	1	

Les entiers (byte, short, int, long) ainsi que les flottants (float, double) sont signés.

Le type boolean N'est PAS un synonyme de int. Les valeurs possibles sont true et false.

Les variables de ces types ont une valeur nulle (0 ou 0.0) par défaut et false pour les boolean.

Constantes et macros

Toute variable déclarée à l'aide du modificateur `final` est une constante (i.e. sa valeur ne peut pas changée). En général une telle constante est déclarée `static final`.

Il n'y a pas de macros en Java : on estime que ceci est traité par les compilateurs modernes.

Passage de paramètres aux méthodes

Il y a 2 sortes de types de données en Java : les types de données "primitifs" et les types de données "référence".
Les types de données primitifs sont `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` et `double`.
Les autres types sont des types "référence" et repèrent des objets de classe.
Le passage des paramêtres aux méthodes est toujours par valeur. Il n'y a pas de recopie d'objet lors de l'appel d'une méthode.
Comme un objet est manipulé par une référence, on peut à l'intérieur d'une méthode manipuler l'objet qui a été passé grâce à l'argument référence et on dit parfois que "les objets passent par référence".
Exemple :

```
MaClasse a, b; // a et b ne sont pas des objets
        // mais des références sur de futurs objets.
a = new MaClasse("A"); // a réfère un objet créé
        // dynamiquement et appelé A
b = new MaClasse("B"); // b réfère un objet créé
        // dynamiquement etappelé B
a.meth(b); // l'objet référencé par b n'a pas été recopié et
        // dans l'exécution de meth(), on peut modifier l'objet
        // de nom B.
```

Lancement des méthodes

Les méthodes (non `static`) sont toujours lancées sur des objets. Même si la syntaxe est trompeuse, le code :

```
class maClasse {
        void foo() { bar();}
}
```

indique que la méthode `foo()` devra être lancé sur un objet de la classe `maClasse`.
Par exemple le code :

```
maClasse ref;
ref = new maClasse().
ref.foo();
```

qui lance la méthode `foo()`, appelle alors la méthode `bar()` sur le même objet référencé par `ref`.
On précise parfois que le lancement de `bar()` est fait sur le même objet que celui qui à lancer `foo()` (i.e. l'objet courant) par

```
this.bar();
```

dans le corps de la méthode `foo()`.

Les classes String et StringBuffer

Les objets de la classe String modélise les chaînes de caractères de longueur constante et leur contenu ne peut pas changer. On peut utiliser une chaîne notée entre guillemets partout où on peut mettre un objet String. Par exemple :

```
"chaine de longueur".length();
```

est correcte.

```
String st = "Bonjour a tous";
```

est sémantiquement équivalent à :

```
String st = new String("Bonjour a tous");
```

bien que l'objet soit créé dans le tas dans la dans la seconde instruction.

Les objets de StringBuffer modélisent les chaînes de caractères de longueur variable. Dans un objet de cette classe, on peut insérer (méthodes insert()) ajouter à la fin (méthode append()) des objets convertissables en "chaînes de caractères". On peut aussi modifier un caractère dans une StringBuffer (méthode setCharAt()).

Pour ces 2 classes les opérateurs + (concaténation) et += sont définis.

Les paquetages

On peut regrouper un ensemble de classes et d'interfaces dans un paquetage définit par l'instruction `package`. Si cette instruction n'existe pas le paquetage est le paquetage "sans nom".
Cette notion permet de restreindre l'accessibilité du code d'un paquetage vis à vis d'un autre paquetage.

Un paquetage est défini par l'instruction

```
package monPaquetage;
```

L'instruction `package` si elle est existe, doit être unique dans un fichier et être la première instruction (après les éventuels commentaires).

Un nom de paquetage peut être de la forme : `nom1.nom2.•••.nomN` auquel cas les classes de ce paquetage doivent être dans un répertoire de la forme : *debut*`/nom1/nom2/•••/nomN` (et dans des fichiers de nom `nomDeLaClasse.class`).

Le mot import

Il ne signifie pas que quelque chose est disponible ou a été chargé.
Il évite de taper du code : c'est un raccourci lexical.

Les instructions `import` doivent apparaître au début du fichier, après l'instruction `package` facultative et avant la définition de toute classe ou interface. Il y a 3 sortes d'instructions `import` :

```
import monpaquetage; // 1

import monpaquetage.MaClasse; // 2

import monpaquetage.*; // 3
```

La forme 1 permet de connaître le paquetage par son dernier nom uniquement. Par exemple

```
import java.awt.image;
```

permet d'écrire `image.ImageFilter` à la place de `java.awt.image.ImageFilter`.

La forme 2 permet d'utiliser le nom d'une classe du paquetage sans mettre le nom du paquetage. Par exemple

```
import java.util.Hashtable;
```

permet d'écrire `Hashtable` à la place de `java.util.Hashtable`.

Le mot `import` (fin)

La forme 3 permet d'utiliser toutes les classes du paquetage en n'indiquant que le nom de la classe. Par exemple

```
import java.util.*;
```

permet d'écrire

```
Date now = new Date();
```

au lieu de

```
java.util.Date now = new java.util.Date();
```

Remarque

```
import java.lang.*;
```

est implicite dans tout programme Java.

Java n'a pas de directive comme `#include` du langage C. A la place mettre ce qui est commun dans un paquetage et l'utiliser à l'aide de `import`.

Les modifieurs

Ce sont les mots suivants : `public`, `protected`, par défaut (i.e. pas de modifieur), `private protected`, `private`, ainsi que `abstract`, `final`, `native`, `static`, `synchronized`.

Ils peuvent agir sur les classes, les méthodes, les variables (i.e. les champs de données d'une instance) ou les interfaces. Ils n'agissent pas tous dans ces 4 domaines. Ils apparaissent au début de déclarations. On a :

Modifieur	utilisé dans :	signification
`public`	classe ou interface méthode ou variable	est visible partout visible partout où sa classe est visible
`protected`	méthode ou variable	visible à l'intérieur du paquetage de sa classe et dans toutes ses sous classes
par défaut (i.e. pas de modifieur)	classe ou interface ou méthode ou variable	visible seulement à l'intérieur de son paquetage
`private`	méthode ou variable	visible qu'à l'intérieur de sa classe

Les modifieurs (suite)

compléments

Dans un fichier `.java`, il ne peut y avoir qu'une seule classe `public`. S'il existe une telle classe, elle donne le nom au fichier. Par exemple :

fichier `MaClasse.java`

```
...
public class MaClasse {
      ...
}
```

Les champs de modifieurs non `private` sont dits parfois "friendly" (i.e. visible dans tout le paquetage).

Une sous classe dans un paquetage différent de sa super-classe peut accéder aux champs `protected` hérités par ses instances mais pas à ce même champ sur des instances de sa super classe.

Modifieur	utilisé dans :	signification
abstract	classe	la classe ne peut pas être instanciée.
	interface	tous les interfaces sont abstract et ce mot est optionnel pour les interfaces.
	méthode	le corps de la méthode n'est pas fourni et sera fourni dans une sous classe
final	classe	on ne peut hérité d'une telle classe.
	méthode	la méthode ne peut pas être réimplantée dans une sous classe.
	variable	ne peut plus changer de valeur
native	méthode	cette méthode est implantée dans un autre langage
static	méthode	méthode de classe. elle est implicitement final.
	variable	variable de classe
synchronized	méthode non static	verrouille l'instance pendant son exécution
	méthode static	verrouille les champs static pendant son exécution

Une classe contenant une méthode abstract doit être abstract.

Une constante de classe est une variable static final.

Les modifieurs : conclusion

En résumé

Un interface peut être : public, "par défaut", abstract.

Une classe peut être : public, "par défaut", abstract, final.

Une méthode peut être : public, protected, "par défaut", private, abstract, final, native, static, synchronized.

Une variable (i.e. une donnée membre) peut être : public, protected, "par défaut", private, final, static.

Le mot clé réservé synchronized

utilisé comme modifieur de méthode de classe ou d'instance (cf. ci dessus) :

```
public synchronized void methAtomique() { ••• }
```

ou

```
public static synchronized void methClasseAtomique() { ••• }
```

synchronized peut aussi être utilisé pour un bloc d'instructions :

```
synchronized (expression) { ••• }
```

où expression repère un objet.

synchronized : modifieur de méthode d'instance

Lorsqu'une thread veut lancer une méthode d'instance synchronized, le système Java pose un verrou sur l'instance. Par la suite, une autre thread invoquant une méthode synchronized sur cet objet sera bloquée jusqu'à ce que le verrou soit levé.

On implante ainsi l'exclusion mutuelle entre méthode synchronized. Attention les méthodes non synchronized ne sont pas en exclusion mutuelle.

Les applications autonomes Java

Définition

application autonome Java = programme "standalone" qui nécessite pas de browser Web pour être lancée. On dit parfois application Java au lieu de application autonome.

Une application autonome Java est donc similaire à un programme dans les langages traditionnels.

Syntaxe

Une application autonome Java contient la méthode membre `main()` dans une de ces classes. On doit avoir :

```
class LaClassePrincipale {
        public static void main(String args[ ]) {
                •••
        }
}
```

C'est par ce code que commencera l'exécution de l'application java.

Remarque :
Si on passe des arguments au programme par
`% java LaClassePrincipale arg1 arg2`
on a :
`args[0]` est l'objet `String` `"arg1"`
`args[1]` est l'objet `String` `"arg2"`

Trame d'une application Java

Très souvent il faut un interface graphique (i.e. une fenêtre "habillée") pour lancer un tel programme. Une tel programme est une instance d'une classe dérivée de la classe `Frame`. On écrit donc :

```
import java.awt.*;

public class trameApp extends Frame {
        // Donnees

  public trameApp ( ... ) {
        // les init nécessaires
        // ...
        pack();
        setVisible(true);
        // ...
  }

  public static void main(String args[]) {
        // ...
        Frame fr = new trameApp ( ... );
        // ...
        }
}
```