

Le langage Java

Rémi Forax
forax@univ-mlv.fr

Java Avancé



La technologie Java

En quelques mots :

- Orienté Objet
- Simple, Robuste, Dynamique et Sécurisé
- Indépendant de la Plateforme (VM)
- Semi Compilé/Semi Interprété
- Bibliothèque Importante (JDK API)

Java Aujourd'hui

3 environnements d'exécutions différents

- Java ME (Micro Edition) pour PDA, téléphone
- Java SE (Standard Edition) pour desktop
- Java EE (Entreprise Edition) pour serveur
Servlet/JSP/JSTL
Portlet/JSF
JTA/JTS, JDO/EJB
JavaMail, etc.

Java Aujourd'hui (2)

API du JDK 1.6 (~7000 classes) :

- `java.lang`, `java.lang.reflect`, `java.lang.annotation`
- `java.util`, `java.util.prefs`, `java.util.concurrent`
- `java.awt`, `java.applet`, `javax.swing`
- `java.io`, `java.nio`, `java.net`
- `java.beans`, `java.sql`, `javax.sql`

etc...

Java Standard Edition

- JDK 1.0 (1995)
- JDK 1.1 (1997)
- JDK 1.2 aka Java 2 (1999)
- JDK 1.3 (2001)
- JDK 1.4 (2002)
- JDK 1.5 aka Java 5 (2004)
- JDK 1.6 aka Java 6 (2006)

Compatibilité ascendante

Java/OpenSource

- Java est OpenSource depuis novembre 2006 (2008 complètement)
- Toutes les sources sont disponibles :
<http://www.openjdk.org>
- N'importe qui peut contribuer, trouver des bugs, etc.
- Install pour linux (ubuntu, fedora)
<http://www.openjdk.org/install/>

Papa et Maman de Java

SmallTalk :

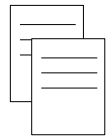
- Tout est objet (même **if**)
- Machine Virtuelle
- Pas de déclaration de type

C/C++ :

- Ecriture du code {, /*, //
- Tout est structure (même la pile :)

Architecture en C

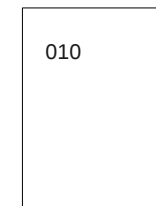
Code en Ascii



Compilateur



Editeur de lien

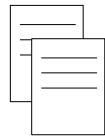


Plateforme + OS

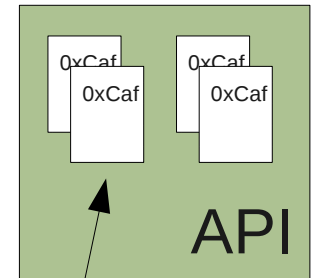
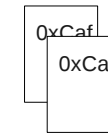
- Le code est compilé sous forme objet relogeable
- L'éditeur de liens lie les différentes bibliothèques entre elles pour créer l'exécutable

Architecture en Java

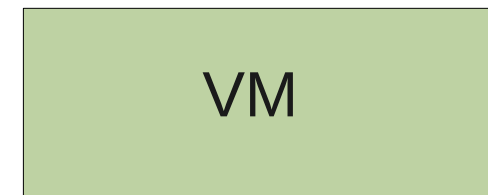
Code en Unicode



Compilateur



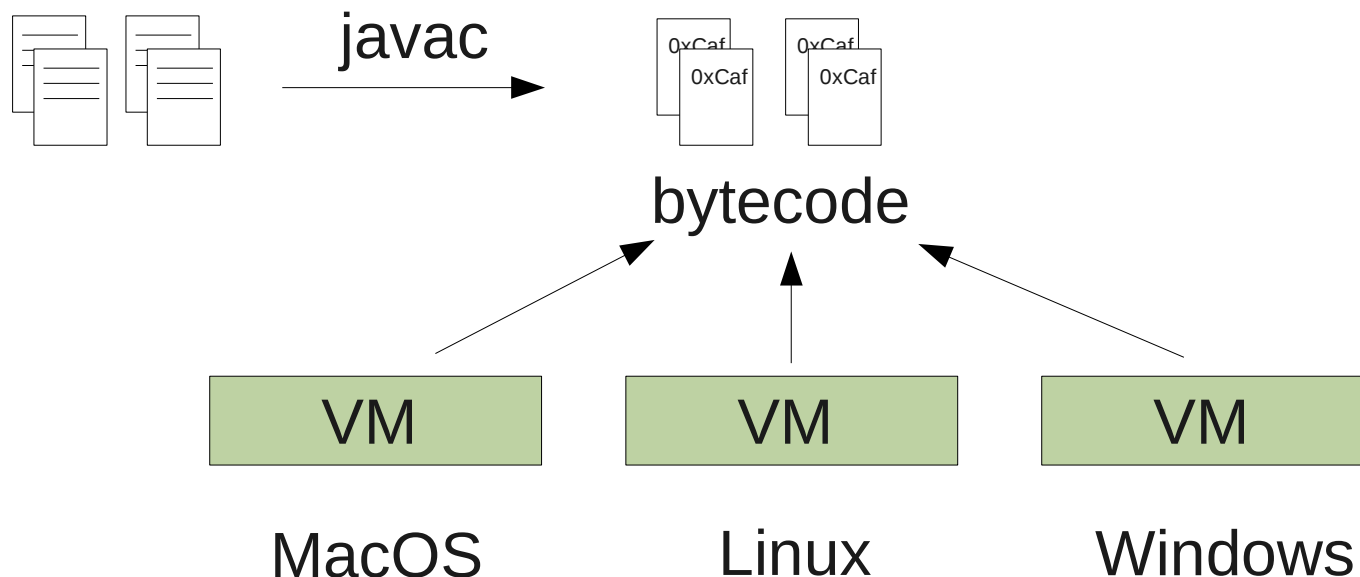
- Le code est compilé dans un code intermédiaire (*byte-code*)
- La Machine Virtuelle charge les classes et interprète le code



Plateforme + OS

Le byte-code

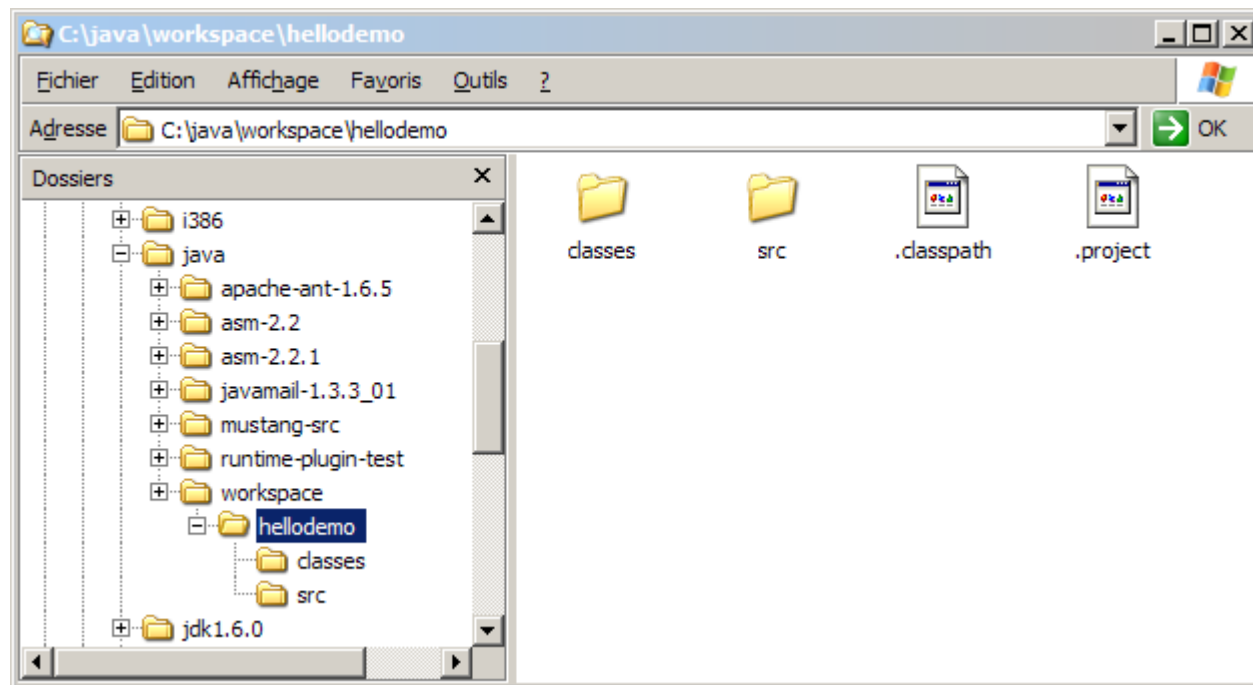
- Il assure la portabilité entre différents environnements (machine/OS)



Et Solaris, HP-UX, BSD etc...

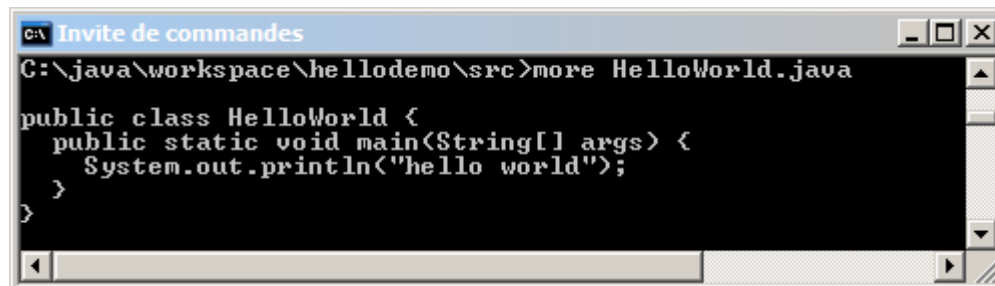
Fichiers sources et byte-code

- Habituellement, on sépare les fichiers sources (dans **src**) des fichiers binaire (dans **classes**)



Compilation simple

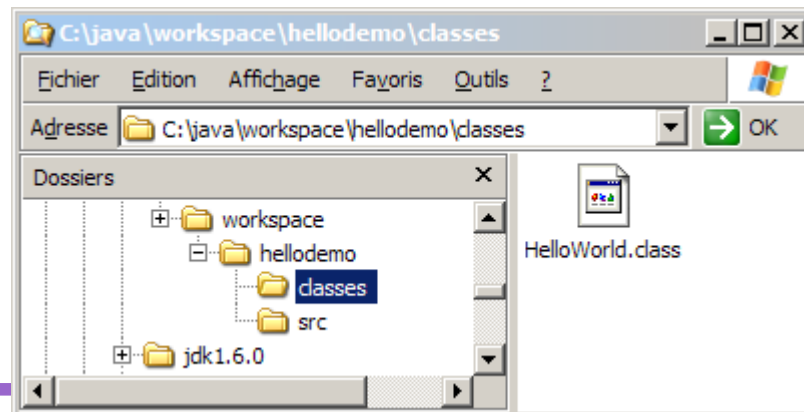
- Dans **src**



```
C:\java\workspace\helldemo\src>more HelloWorld.java

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

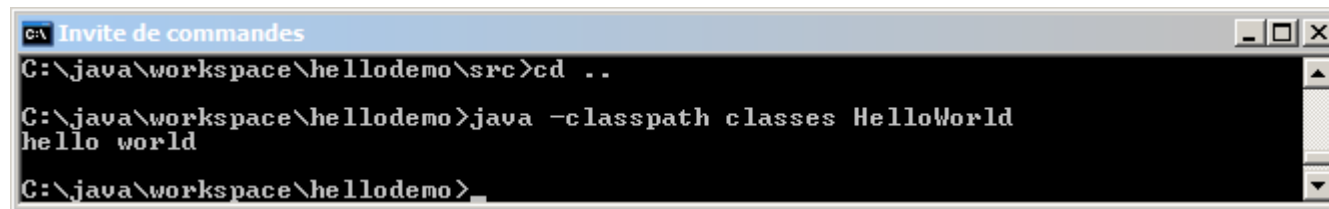
- Compiler avec la ligne suivante :
javac -d ../classes HelloWorld.java



crée le fichier
HelloWorld.class
dans **classes**

Exécution simple

- En remontant d'un répertoire



```
C:\java\workspace\helldemo\src>cd ..
C:\java\workspace\helldemo>java -classpath classes HelloWorld
hello world
C:\java\workspace\helldemo>
```

- Exécuter avec la ligne suivante :
java -classpath classes HelloWorld
- On indique où se trouve les fichiers binaires (avec **-classpath**) ainsi que le nom de la classe qui contient le **main**.

Compilation des sources

- Les sources sont compilés avec javac :
 - **-d** *destination* répertoire des .class générés (le compilateur crée les sous répertoires)
 - **-classpath/-cp** fichiers .class et .jar dont dépendent les sources
 - **-sourcepath** les fichiers sources
- Exemple :
javac -d classes -cp ../other/classes:lib/truc.jar -sourcepath src src/fr/umlv/projet/Toto.java

Variable d'environnement

Java et certains programmes annexes utilisent les variables d'environnement :

- **CLASSPATH** qui correspond aux répertoires des classes
(valeur par défaut de `-cp/-classpath` si l'option n'est pas présente)
- **JAVA_HOME** qui correspond au répertoire racine du JDK (*ant* en a besoin !!)

Version du langage

- Gestion version de Java :
 - source** version [1.2, 1.3, 1.4, 1.5, 1.6]
 - target** version [1.2, 1.3, 1.4, 1.5, 1.6]
- Cela permet de compiler contre un JDK moins récent
- Toutes les combinaisons ne sont pas valides
Ex: -source 1.5 implique -target 1.5

Gestion des warnings

- Par défaut, le compilateur indique qu'il y a des warnings sans préciser lesquelles
- Il faut ajouter :
 - **-Xlint:deprecation** affiche le code déprécié (fonction existante à ne plus utiliser)
 - **-Xlint:unchecked** affiche les cast non vérifiable à l'exécution (cf *generics*)
 - **-Xlint:all** demande tout les warnings

Autres Compilateurs

Trois compilateurs (Java -> bytecode)

- Javac (livré avec Java SDK)
- Jikes (écrit en C++, IBM)
- Eclipse (ex IBM)

Compilateur de Java vers code machine

- GCJ (GCC + frontend/backend Java)
- Excelsior JET

Machine Virtuelle

- Machine virtuelle :
 - Garantie le même environnement d'exécution pour différentes machines
(Write Once Run Anywhere)
 - Surveille la où les applications lancées dans le but de les optimiser en fonction de la machine
(comme un OS)
- Pas propre à Java, Perl, Python, Ruby, PHP possèdent des machines virtuelles

Exécuter une classe

- On lance la VM avec la commande java
 - **-cp/-classpath** locations des archives et .class
 - **-ea** active les assertions
 - **-Dname=value** change/ajoute une propriété au système
 - **-server** demande des optimisation plus agressive
- Exemple :
java -cp classes:lib/truc.jar fr.umlv.projet.Toto

Paramètre de la VM

- Autres options de la VM :
 - **-Xint** mode interpréter uniquement
 - **-Xbootclasspath**
spécifie le chemin des classes de l'API
 - **-Xms** taille initiale de la mémoire
 - **-Xmx** taille maximale de la mémoire
- <http://blogs.sun.com/roller/resources/watt/jvm-options-list.html>

Autres Machine Virtuelle et JIT

Les machine virtuelles :

- Hotspot (SUN) pour Windows, Linux, Solaris et Hotspot (Apple)
- JVM (IBM)
- JRockit (BEA)
(AOT: ahead of Time compiler)
- Kaffe, SableVM, Jikes RVM
- gij (avec GCJ)

Machine Virtuelle et Interprétation

- La machine virtuelle interprète le byte-code
- Un interpréteur :
 - Tant qu'il y a des instructions
 - On lit une instruction de byte-code
Ex: iadd
 - On effectue l'opération correspondante
- Problème : facteur 1 à 1000 par rapport à du code compilé en assembleur.

Le JIT (Just In Time)

- Pour améliorer les performances, lors de l'exécution, on transforme le byte-code en code assembleur pour la machine courante
 - Avantage :
 - Exécution, pas interprétation (très rapide)
 - On adapter l'assembleur au processeur courant (P4 si c'est un P4, Turion si c'est un Turion, etc.)
 - Inconvénient :
 - La transformation prend du temps (allocation des registres, inlining, déroulement des boucles etc)
 - Impossible de produire l'assembleur de tout le programme sans exploser la mémoire

Un exemple

Mesure le temps passé dans la fonction de Fibonacci (mal codée) :

On déclare les types le plus près de là où on les utilise

```
public class JITExample {
    private static int fibo(int n) {
        if (n==0 || n==1)
            return 1;
        return fibo(n-1)+fibo(n-2);
    }
    private static long time(int n) {
        long time=System.nanoTime();
        for(int i=0;i<20;i++)
            fibo(n);
        long time2=System.nanoTime();
        return time2-time;
    }
    public static void main(String[] args) {
        for(int i=0;i<10;i++)
            System.out.println(time(5));
    }
}
```

Un exemple (suite)

- Compilons avec javac

```
C:\eclipse\workspace\java-avancé\src>dir
Répertoire de C:\eclipse\workspace\java-avancé\src

19/07/2004  16:08                509 JITExample.java
                1 fichier(s)                509 octets

C:\eclipse\workspace\java-avancé\src>javac JITExample.java

C:\eclipse\workspace\java-avancé\src>dir
Répertoire de C:\eclipse\workspace\java-avancé\src

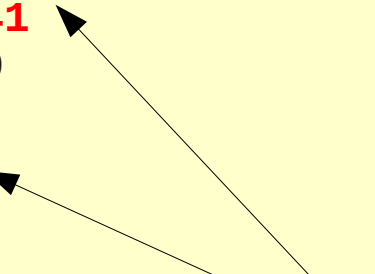
19/07/2004  16:12                650 JITExample.class
19/07/2004  16:08                509 JITExample.java
                2 fichier(s)                1 159 octets

C:\eclipse\workspace\java-avancé\src>
```

On exécute avec la VM

On utilise la commande java

```
C:\eclipse\workspace\java-avancé\src>java JITExample
34082
32966
31568
31568
809041
61739
5867
5587
5587
5587
```



Expliquez les fluctuations de vitesse ?

Explication

- Le code est interprété un certain nombre de fois puis transformé en code machine à l'aide du JIT
- Cette transformation à lieu en tâche de fond ce qui ralentit l'exécution (au moins sur une machine mono-processeur)
- Une fois généré, le code assembleur remplace le code interprété ce qui accélère l'exécution

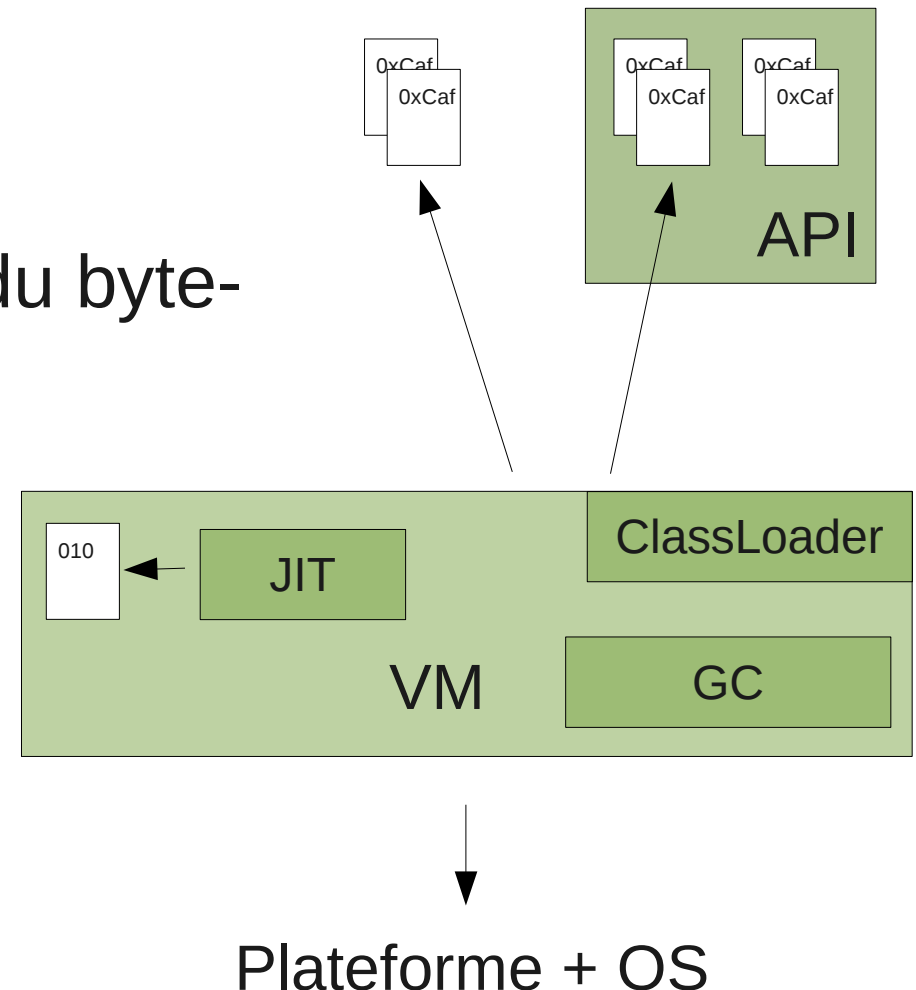
Explication (2)

- L'option `+PrintCompilation` affiche la transformation en code machine

```
C:\eclipse\workspace\java-avancé\src15>java -XX:+PrintCompilation
JITExample
  1    b    java.lang.String::charAt (33 bytes)
32686
29054
28495
28774
  2    b    JITExample::fibo (25 bytes)
994260
61181
5587
...
```

Architecture en Java (revisit )

- Un chargeur de classe (classloader)
- Un JIT (transformation   la vol e du byte-code)
- Le Garbage Collector (r cup re les objets non utilis s)



Java & Performance

- Même ordre de magnitude que le C ou le C++ (dépend de l'application)
- Théoriquement plus rapide car :
 - Compilation à l'exécution donc :
 - Optimisé en fonction de l'utilisation
 - Optimisé en fonction du processeur réel
 - Inline inter-bibliothèque
 - GC est plus efficace que malloc/free !!

<http://www.idiom.com/~zilla/Computer/javaCbenchmark.html>

Archive java

- La commande jar permet de créer une archive (au format ZIP) contenant les fichiers .class et les ressources
- Une archive contient des méta-données stockées dans un fichier manifest
- Créer une archive avec manifest :
cd classes
jar cvfm ../manifest ../archive.jar

Archive java

- La commande jar permet de créer une archive (au format ZIP) contenant les fichiers .class et les ressources
- Une archive contient des méta-données stockées dans un fichier manifest
- Créer une archive avec manifest :
cd classes
jar cvfm ../manifest ../archive.jar

Désassembleur de *bytecode*

- Javap permet d'afficher les informations contenues dans les fichiers de classes
 - **-classpath** localisation des fichiers
 - **-public/-protected/-package/-private** visibilité minimum des membres affichés
 - **-c** affiche le code en plus de la signature des méthodes
 - **-s** affiche les types des membres suivants le format interne de la VM

Exemple de javap

```
-bash-2.05b$ javap -classpath classes -c JITExample
public class JITExample extends java.lang.Object{
public JITExample();
    0:    aload_0
    1:    invokespecial    #8; //Method java/lang/Object."<init>":()V
    4:    return

public static void main(java.lang.String[]);
    0:    iconst_0
    1:    istore_1
    2:    goto    18
    5:    getstatic        #33; //Field
java/lang/System.out:Ljava/io/PrintStream;
    8:    iconst_5
    9:    invokestatic     #37; //Method time:(I)J
   12:    invokevirtual    #39; //Method java/io/PrintStream.println:
(J)V
   15:    iinc      1, 1
   18:    iload_1
   19:    bipush   10
   21:    if_icmplt      5
   24:    return
}
```

debugger/profileur

- Le debugger :
 - jdb remplace la commande java, même ligne de commande que java
 - permet de lancer le debugger sur une machine différente de la machine de production
- Le profiler :
 - Utilise le système d'agent
 - **java -agentlib:hprof=[help][<option>=<value>, ...]**

Autre outils autour de Java

- **javadoc**
génération de la doc automatique
- **jstat, jconsole, jmap, jps, jinfo, jhat**
monitoring de la machine virtuelle
- **javah**
génération header en C (interop avec C)
- **keytool, policytool**
gestion signature et sécurité
- **rmiregistry, ordb, derby.jar** (JavaDB)
Registry RMI, Orb CORBA, BD embarquée

Organisation d'un projet Java

- A la racine, un **README** et un **build.xml**
- Les sources (**.java**) sont dans le répertoire **src**
- Les fichiers de byte-code (**.class**) sont dans le répertoire **classes**
- Les bibliothèques nécessaires (**jar** et **so/dll** en cas de **JNI**) sont dans le répertoire **lib**
- La documentation dans **docs**, la javadoc dans **docs/api**

Script ant

- Un script ant est l'équivalent d'un makefile pour les projets java
- Ant utilise un fichier XML appelé build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="mon projet" default="compile"
    basedir="."
    <description>
        Description du projet
    </description>

    <!-- déclaration des tâches -->
</project>
```

Script ant

- Les tâches ont la forme suivante
- Ant utilise un fichier XML appelé build.xml

```
<target name="run" depends="compile">  
  <!-- action a effectuer -->  
</target>
```

- Chaque action est une balise XML dont les attributs/éléments sont spécifiques

```
<javac srcdir="src" destdir="classes"  
  debug="true"/>
```

Script ant

- Les tâches ont la forme suivante
- Ant utilise un fichier XML appelé build.xml

```
<target name="run" depends="compile.jar">  
  <!-- action a effectuer -->  
</target>
```

- Chaque action est une balise XML dont les attributs/éléments sont spécifiques

```
<javac srcdir="src" destdir="classes"  
  debug="true"/>
```

Tâche ant : compilation

- La tâche javac à la forme suivante :

```
<javac srcdir="src" destdir="classes"  
    debug="true">  
    <compilerarg value="-Xlint:all"/>  
</javac>
```

- En utilisant une liste de fichier :

```
<javac srcdir="src" destdir="classes"  
    debug="true">  
    <compilerarg value="-Xlint:all"/>  
    <fileset dir="${src}" includes="fr/umlv/monprojet/**/*.java"/>  
</javac>
```

Tâche ant : exécution

- Pour exécuter des classes :

```
<java classpath="classes"  
      classname="fr.uml.v.projet.Main"/>
```

- Exécuter un jar :

```
<java jarfile="projet.jar"/>
```

- Pour exécuter dans une autre VM :

```
<java fork="true" jarfile="projet.jar"/>
```

Script ant : Paramétrage

- Les propriétés permettent de paramétrer facilement un fichier ant

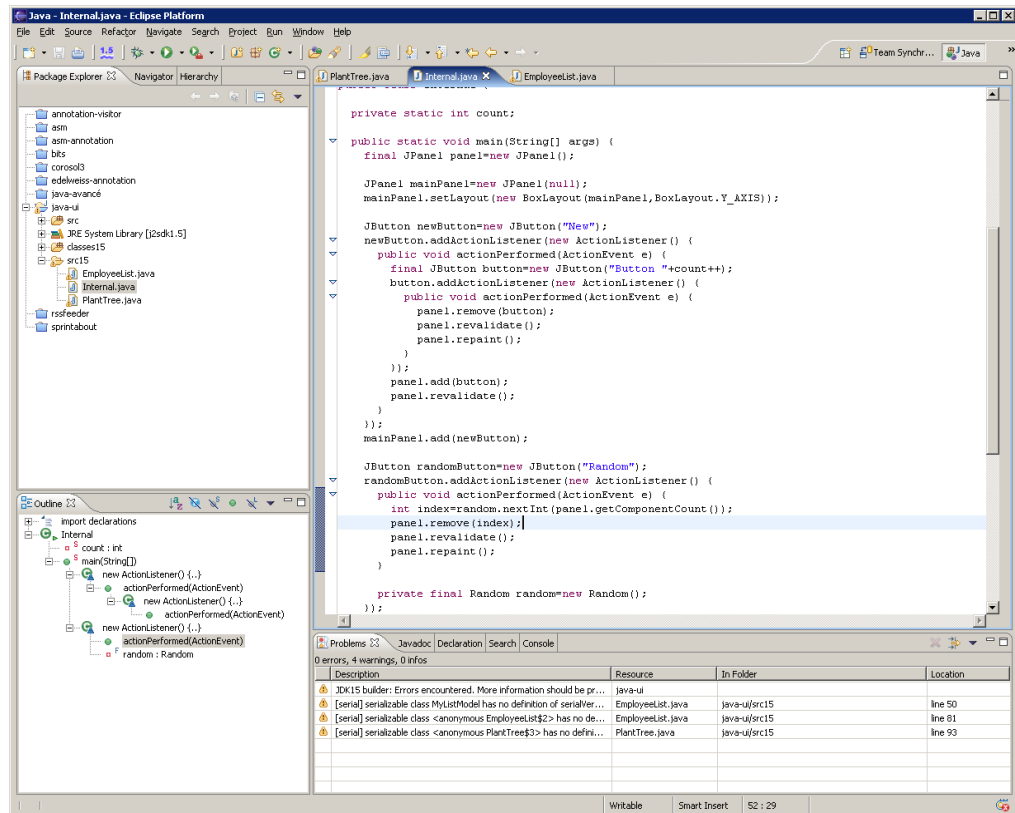
```
<property name="src.dir" location="src"/>
<property name="classes.dir" location="classes"/>
<property name="main" value="fr.umlv.projet.Main"/>

<target name="compile">
  <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
</target>
<target name="run" depends="" compile>
  <java classpath="${classes.dir}"
    mainclass="${main}"/>
</target>
```

IDE pour programmer en Java

Outils de développement visuels (IDE) :

- Eclipse (ex IBM)
- NetBeans (ex SUN)
- IDEA (IntelliJ)



Plateforme Java - En résumé

- Code portable (VM)
- Syntaxe du C mais objet
- Accès à la mémoire fortement contrôlé
(pas de pointeurs mais des références)
Libération automatique de la mémoire (GC)
- Transformation en code machine à la volée
- Exécution de morceaux de code en concurrence
- Introspection