Cours JAVA : Gestion de projets Java.

Version 1.01

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe RÉGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Licence professionnelle DANT - 2015/2016

Grandes lignes du cours

Outline

Gérer un gros projet

Dans le cycle de vie d'un projet :
- le temps consacré au débugage et à la maintenance est très largement supérieur au temps de développement ;
- les développeurs sont nombreux et changent souvent.

Les sources du projet doivent donc :
1. suivre les mêmes règles de syntaxe et de présentation ;
   $\Longrightarrow$ **de nombreuses règles sont données dans ce cours**
2. être correctement structurées ;
   $\Longrightarrow$ **il faut utiliser des packages hiérarchiques**
3. être suffisamment documentées ;
   $\Longrightarrow$ **les sources doivent être annotées pour la javadoc**
4. se compiler et se distribuer simplement.
   $\Longrightarrow$ **l'utilisation de l'outil ant simplifie les choses**

Outline

Outline

Les packages : définition

Définition

*Un **package** est une bibliothèque d'objets, c'est-à-dire une collection de classes rangées ensemble (dans un même répertoire).*

Il y a trois raisons pour faire des packages :
- **structuration des projets :** chaque package est relié à un répertoire.
- **conflits de nom :** les packages définissent des espaces de nommage.
- **affiner la visibilité :** la notion de package est associée à un niveau de visibilité intermédiaire.

Déclaration d'un package

Pour créer un package :
- il suffit de créer un répertoire ;
- le nom du répertoire, en minuscule, est le nom du package.

Pour rattacher une classe à un package :
- on écrit sur la première ligne du fichier texte contenant la définition de la classe le mot-clé **package** suivi du nom du package ;
- les classes d'un même package sont toutes rangées dans un même répertoire.

```
package pim ;

public class Poum {
   ...
}
```

Organisation hiérarchique des packages

Les packages peuvent être organisé hiérarchiquement :

- Par exemple la classe `Poum` : `./pim/pam/Poum.java`

```
package pim.pam;

public class Poum {
   ...
}
```

Pour éviter les conflits de nom lors la distribution des classes :

- on utilise comme racine une URL inversée
- suivi des répertoires l'arborescence

```
package fr.lip6.perso.jsopena.dant.pim.pam;
```

```
package org.xml.sax.helpers;
```


Le package par défaut

Définition

*Un **package par défaut**est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Toutes ces classes appartiennent donc au même package.*

Attention

Toutes les classes d'un projet doivent être affectées à un package. En effet, le mécanisme du **package par défaut** ne permet pas d'aider le compilateur et la machine virtuelle à trouver les classes utilisées (voir exemple plus loin).

Outline

Recherche des classes à la compilation

Lorsque l'on utilise un membre (de classe ou d'instance) d'une classe `A` dans une méthode d'une classe `B`, le compilateur va chercher la classe `A` :

1. dans la classe B (*notion de classe interne*) ;
2. parmi toutes les classes appartenant au même package que `B` ;
3. parmi toutes les classes du package `java.lang`.

Si la classe `A` n'est pas définie dans l'un de ces trois ensembles, vous devez aidez le compilateur à la trouver en indiquant :

- soit à chaque utilisation de la classe `A` ;
- soit une fois pour toute au début de la classe `B`.

Accès à une classe par chemin complet

Pour utiliser la classe `Poum` du package hiérarchique `pim.pam`, on doit aider le compilateur en lui indiquant à chaque utilisation :

le chemin complet d'accès `pim.pam.Poum`

```
public class Canon {
  pim.pam.Poum poum ;

  public Canon () {
    this.poum = new pim.pam.Poum();
  }
  public pim.pam.Poum getPoum() {
    return this.poum ;
  }
}
```


Import d'une classe pour en simplifier l'accès

Pour alléger le code, on peut utiliser un **import** au début du fichier. On indique a alors une fois pour toute où trouver la classe `Poum`.

```
import pim.pam.Poum ;

public class Canon {
  Poum poum ;

  public Canon () {
    this.poum = new Poum();
  }
  public Poum getPoum() {
    return this.poum ;
  }
}
```


Import d'une classe pour en simplifier l'accès

Pour utiliser les fonctionnalités d'un package, il est souvent nécessaire d'importer de nombreuses classes.

C'est, par exemple, le cas avec les entrées/sorties du `java.io`

```
import java.io.InputStream;
import java.io.DataInputStream;
import java.io.BufferedInputStream;

public class Test {
  public static void main (String[] args) {
    InputStream is = new InputStream();
    DataInputStream dis = new DataInputStream(is);
    BufferedInputStream bdis = new BufferedInputStream(dis);
    ...
  }
}
```


Import d'une classe pour en simplifier l'accès

Lorsque l'on utilise de nombreuses classes d'un package, on peut simplifier l'importation en important simultanément toutes les classes du package grâce à une astérisque :

import `nomPackage.*`

```
import java.io.*;

public class Test {
  public static void main (String[] args) {
    InputStream is = new InputStream();
    DataInputStream dis = new DataInputStream(is);
    BufferedInputStream bdis = new BufferedInputStream(dis);
    ...
  }
}
```

Import d'une classe pour en simplifier l'accès

Attention

L'utilisation de l'astérisque dans un **import** n'importe pas récursivement les sous-packages : seules les classes se trouvant directement dans le package sont importées.

```
import java.awt.*; // Importe java.awt.Dialog
import java.awt.event.*; // Importe java.awt.event.WindowEvent
public class MonListener {
  Dialog dialogue;
  public void windowClosing(WindowEvent e) {
    dialogue.dispose();
  }
}
```


Boite à outils : import de membres de classe

De nombreuses classes java sont des "boites à outils" :

- elles offrent un ensemble de constantes et de méthodes de classe.
- elles ne sont pas destinées à être instanciées.

Comment doit-on utiliser ces boites à outils, par exemple la classe `Math` ?

- Accès aux méthodes par une instance : `Math m = new Math() ;`
  $\Longrightarrow$ **C'est horrible**
- Importer la classe `Math`
  $\Longrightarrow$ **Inutile, elle est importée par défaut avec `java.lang`**
- Accès aux méthodes par la classe : `Math.cos(90) ;`
  $\Longrightarrow$ **C'est lourd s'il y a beaucoup de calculs**
- Quelle est alors la bonne pratique ?
  $\Longrightarrow$ **La bonne solution c'est l'import static**

Boite à outils : import de membres static

Exemple utilisant des méthodes de classe sans **import static**

```
public class Test {
  double testTrigo (int i) {
    Thread.sleep(3);
    return Math.pow(Math.sin(i),2)+Math.pow(Math.cos(i),2);
  }
}
```

Le même exemple en utilisant des **import static**

```
import static java.lang.Thread.sleep;
import static java.lang.Math.*;
public class Test {
  double testTrigo (int i) {
    sleep(3);
    return pow(sin(i),2)+pow(cos(i),2);
  }
}
```


Les packages standards

Les classes du langage Java sont organisées en package :

- `java.lang` : Les classes de base et la gestion des processus (*importer par défaut*)
- `java.util` : Des structures de données bien pratiques
- `java.io` : La gestion des flux comme l'accès aux fichiers
- `java.math` : Calcul sur des nombres infinis
- `java.awt` : Composants graphiques : boutons, fenêtres ...
- `javax.swing` : Une Alternative aux composants graphiques AWT
- `java.applet` : Tout pour faire des applets
- `java.net` : Permet des connections réseau IP
- `java.sql` : Utilisation de bases de données relationnelles en SQL
- `java.security` : Mécanismes assurant des transactions sécurisées
- `javax.rmi` : Objets distribués RMI propriétaire Java
- `org.omg.corba` : Objets distribués selon la norme CORBA

Outline

Les conflits sont fréquents

Spécifier le chemin complet

Préciser l'import

Outline

Visibilité

Le modificateur qui précède chaque méthode (et attribut) détermine sa visibilité :

Modificateur	Visibilité
`private`	uniquement au sein la classe
`pas de modificateur`	uniquement au sein du même package
`protected`	*lié à l'héritage*
`public`	au sein de toute classe

Outline

Outline

Les commentaires

Définition

*Les **commentaires** sont des portions du code source ignorées par le compilateur ou l'interpréteur, car ils ne sont pas nécessaires à l'exécution du programme. Ils sont souvent utilisés pour expliquer : la structure code, le fonctionnement des algorithmes, le sens des variables, les optimisations, ...*

Un bon commentaire :

- ne paraphrase pas le code ;
- n'explique pas le "quoi" ou le "comment" ;
- décrit succinctement le "pourquoi" du code.

Avant d'ajouter un commentaire :

1. se demander si le code n'est pas déjà assez clair ;
2. clarifier le code plutôt que de le commenter

La documentation

Définition

*Une **documentation** est un texte écrit qui accompagne un logiciel ou une bibliothèque informatique afin d'en expliquer le fonctionnement, et/ou l'utilisation.*

Une documentation, qui n'est pas à jour, est inutile. Il peut être couteux de maintenir la synchronisation code/documentation. Une solution est la fusion code/documentation en un seul fichier.

Deux approches sont envisageables :

1. **programmation littéraire** : développement du code dans un texte descriptif (préconisée par Donald Knuth).
2. **documentation générée** : à partir de texte descriptif en commentaires dans du code (javadoc, doxygen, docstrings, ...) .

Les commentaires *javadoc*

Définition

***JavaDoc** est un outil, fournit avec le SDK, permettant de créer une documentation au format HTML depuis les commentaires présents dans un code source Java.*

Ces commentaires suivent des règles précises :

- ils commencent par `/**`, finissent par `*/` et contiennent toujours au minimum une phrase. Chaque ligne supplémentaire commence par des espaces (ignorés) puis `*` ;
- ils peuvent contenir des tags *HTML* pour formatter le texte ;
- ils utilisent des tags prédéfinis par *javadoc*, commençant tous par le caractère arobase @, pour spécifier des éléments particuliers : auteur, paramètres, valeur de retour,...

Outline

Les tags définis par *javadoc*

Il existe deux types de tags :

@tag : les tags standards ;

{@tag} : les tags qui seront remplacés par une valeur.

Ces tags peuvent être utilisés pour documenter :

- globalement le programme (fichier overview.html) ;
- des **packages** (fichier package.html ou package-info.java) ;
- des **classes** et **interfaces** ;
- des **constructeurs** et **méthodes** ;
- des **champs**.

Les tags des packages, classes et interfaces.

Pour documenter les packages, on peut utiliser :

- **@author texte :** précise le ou les auteur(s) ;
- **@version texte :** numéro de version du packages.

Pour documenter les classes et interfaces, on peut utiliser :

- **@author texte :** précise le ou les auteur(s) ;
- **@version texte :** numéro de version de l'élément ;
- **@deprecated texte :** permet de préciser qu'un élément est déprécié.

Les tags des méthodes et constructeurs.

Pour documenter les méthodes et constructeurs :

- **@version texte :** numéro de version de l'élément
- **@deprecated texte :** permet de préciser que la méthode est dépréciée ;
- **@param nom_paramètre description du paramètre :** défini un paramètre d'une méthode
- **@exception nom_exception description :** exception pouvant être levée par une méthode
- **@throws nom_exception description :** identique à @exception
- **@return texte :** description de la valeur de retour de la méthode
- **{@inheritDoc} :** recopie de la documentation de l'entité de la classe mère

Les tags génériques.

Les textes correspondant aux différents tags sont libres, mais les caractères liés au *HTML* comme **<** ou **>** sont interprétés spécialement par javadoc. Pour écrire **<**, on doit donc utiliser **&lt ;**.

En plus des tags spécifiques aux différents éléments, on peut utiliser :

- **@docRoot :** chemin relatif de la racine de la documentation ;
- **{@link} :** insère un lien vers un élément de la documentation ;
- **{@linkplain} :** identique à link dans une police normale ;
- **@see entité :** référence à un autre élément ;
- **@since texte :** version où l'élément a été ajouté.
- **{@literal texte} :** permet d'insérer plus facilement les caractères spéciaux sans les interpréter.

Outline

Génération de la documentation

Pour générer la documentation, on utilise la commande **javadoc** suivie d'une liste de fichiers `<fichier.java>`.

Les options les plus couramment utilisées :

-d <dir> : permet de choisir le répertoire dans lequel sera générée la documentation ;

-author : permet de prendre en compte les tag @author ;

-version : permet de prendre en compte les tag @version ;

-public : limite la documentation aux classes publiques ;

-protected : limite la documentation aux classes et aux membres publics et protégés (par défaut) ;

-package : limite la documentation aux classes et aux classes et aux membres publics, protégés et ceux accessibles aux classes d'un même package ;

-private : génère la documentation pour toutes les classes et tous les membres.

Problème des accents avec javadoc

Attention, pour utiliser correctement les accents avec la javadoc, il faut préciser les encodages avec les options suivantes :

-encoding spécifie l'encodage utilisé dans les sources java.

-docencoding spécifie l'encodage qui sera utilisé pour générer les fichiers de documentation HTML.

-charset permet d'ajouter dans le code source HTML l'header `Content-Type` avec l'encodage spécifié, ce qui permet au navigateur d'utiliser directement le bon encodage.

Pour obtenir une doc HTML en *utf8* avec des sources en *latin1* :

```
javadoc -encoding ISO-8859-1 -docencoding utf8 -charset utf8 *.java
```


Formatage à l'aide de doclet

Pour formater la documentation, *javadoc* utilise une **doclet** qui permet de préciser le format de la documentation générée.

Par défaut, *javadoc* utilise une doclet qui génère une documentation au format HTML, mais il est possible de définir ses propres doclet pour changer le contenu ou le format de la documentation.

De multiples doclets sont disponibles sur www.doclet.com. Parmi les doclets utiles, on peut citer :

DocCheck : rapport sur la cohérence du code

PDFDoclet : génération d'une javadoc dans un document PDF

Exemple : Documentation d'un package

Depuis *Java 5.0* on peut utiliser directement la syntaxe `javadoc` pour documenter les packages.

Pour cela, il faut ajouter dans le répertoire correspondant au package un fichier **package-info.java** qui contient une seule ligne java **package <nomPackage> ;**.

```java
/**
 * Package regroupant des animaux en danger.
 * @author Julien Sopena
 * @version 2.0
 */
package wwf ;
```


Exemple : Documentation d'une classe

```java
package wwf ;

/**
 * Classe générique d'animaux
 * @author Julien Sopena
 * @version 1.00
 */
public class Animal {
  int pos=0;
  /**
   * Méthode simulant la course de l'animal
   * @param dist distance parcourue par l'animal
   * @return retourne la position actuelle
   */
  public int courir (int dist) {
    return pos += dist ;
  }
}
```



```java
package wwf ;

/**
 * Classe d'un chien qui traverse la route, et ...
 * @see <a href="http://fr.wikipedia.org/wiki/Paf_le_chien">wikipedia</a>
 * @author Julien Sopena
 * @version 1.01
 */
public class Chien extends Animal {
  private boolean state = true ;

  /**
   * Teste l'état du chien
   * @return true si l'état du chien est correct.
   */
  public boolean isCorrect () {
    return state ;
  }

  /** {@inheritDoc} et enregistrant la collision */
  @Override
  public int courir (int dist) {
    this.paf(false);
    return super.courir(dist);
  }

  /**
   * et paf le chien
   * @param state nouvel état du chien
   */
  private void paf (boolean state) {
    this.state = state;
  }

}
```

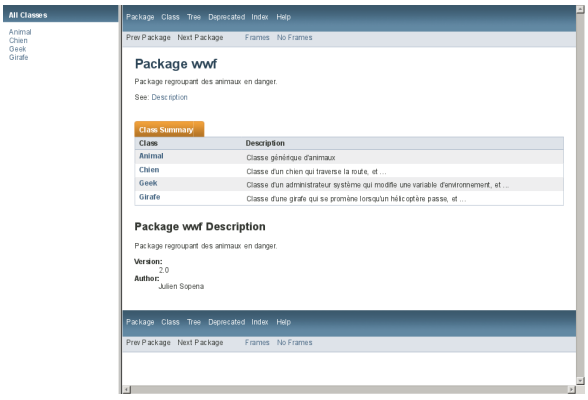

Exemple : on génère la documentation

Pour générer la documentation il suffit de lancer la commande *javadoc* sur l'ensemble des sources java.

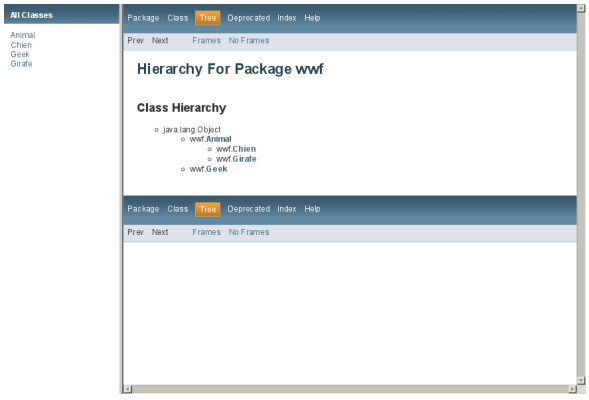
La documentation contient un fichier *HTML* par classe et trois par package. On trouve aussi à la racine plusieurs fichiers dont un *css* et un *index.html*.

```
ls wwf
 Animal.java Chien.java package-info.java

javadoc -private -author -version -encoding utf8 wwf/*.java

ls wwf
 Animal.html Chien.html package-frame.html package-summary.html
 Animal.java Chien.java package-info.java package-tree.html
```

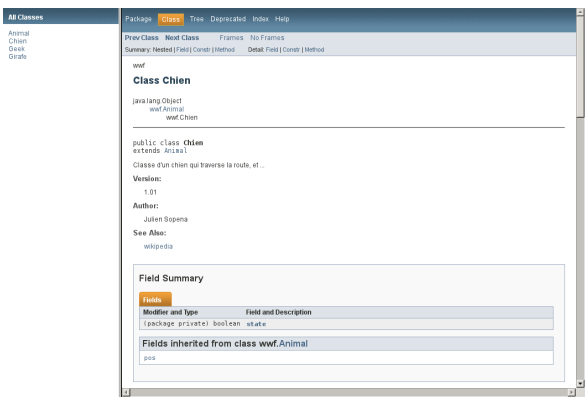

Exemple : la javadoc du package wwf (1)



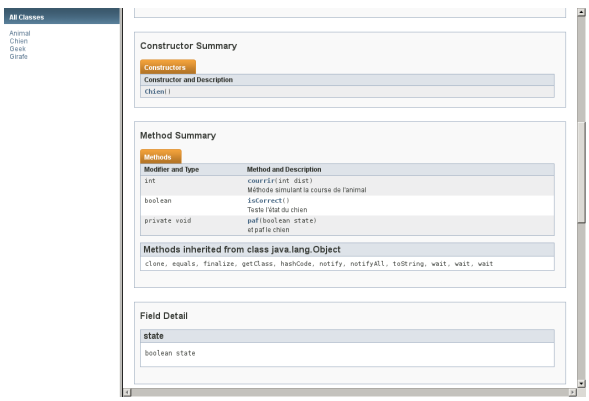
Exemple : la javadoc du package wwf (2)



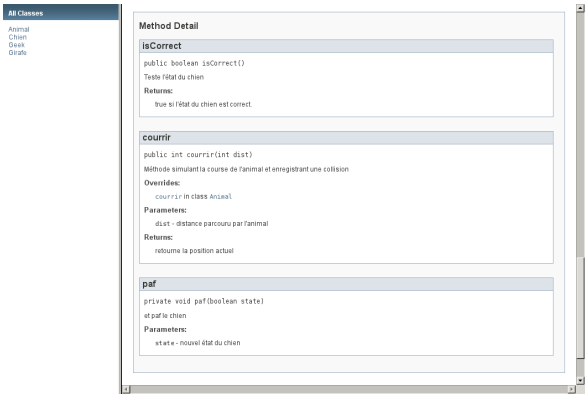
Exemple : la javadoc de la classe Chien (1)



Exemple : la javadoc de la classe Chien (2)



Exemple : la javadoc de la classe Chien (3)



Outline

Outline

Tout est mélangé par défaut

Par défaut, le compilateur **javac** et l'outil **javadoc** produisent respectivement le bytecode (`xxx.class`) et la documentation (`xxx.html`) dans le répertoire du fichier source (`xxx.java`).

Ce mélange complique :

- le **parcours** des sources lors du développement ;
- la **distribution** qui ne contient pas toujours les sources ;
- l'**installation** si bytecode et documentation s'installent séparément.

La solution standard est de créer à la racine du projet 3 répertoires :

1. **./src** : qui contiendra l'ensemble des packages et des classes
2. **./bin** : qui contiendra l'ensemble des bytecodes
3. **./doc** : qui contiendra l'ensemble de la documentation

Tout est mélangé par défaut

Pour séparer **bytecodes** et **sources**, on se place à la racine du projet, on utilise deux options du compilateur **javac** :

-d <dir> : génère les bytecodes dans le répertoire `<dir>` (qui doit déjà exister) en reprenant l'arborescence des sources.

-classpath <dir> : permet d'ajouter `<dir>` à la liste des répertoires où le compilateur cherche les classes déjà compilées.

```
javac -d ./bin -classpath ./bin src/*.java src/wwf/*.java
```

Pour séparer **documentation** et **sources**, on se place à la racine du projet et on utilise deux options de l'utilitaire **javadoc** :

-d <dir> : génère les documentations dans le répertoire `<dir>` (qui doit déjà exister) en reprenant l'arborescence des sources.

```
javadoc -d ./doc src/*.java src/wwf/*.java
```


Compiler les dépendances

L'outil de compilation intègre un mécanisme récursif de compilation des dépendances. Comme dans un *Makefile*, les dépendances sont :

- **compilées :** si le bytecode n'existe pas ou n'est pas accessible depuis le *classpath*.
- **re-compilées :** si le bytecodes existant est plus ancien que le fichier source.

Si l'on ne lance pas la compilation depuis la racine des sources, on doit aider le compilateur à trouver les sources avec l'option :

-sourcepath <dir> : ajoute le répertoire `dir` comme racine de recherche des dépendances.

```
javac -d ./bin -classpath ./bin -sourcepath ./src src/GrossesTetes.java
```


Outline

Jar le format de distribution Java

La machine virtuelle chargeant les classes dynamiquement, un programme n'est pas constitué d'un unique binaire. Distribuer son application revient donc à distribuer les bytecodes de chaque classe utilisée. Pour simplifier cette tache, Java définit un format de distribution : le **jar**.

Définition

*Un fichier **jar** (**Java ARchive**) est un fichier **zip** regroupant un ensemble de classes Java ainsi que des métadonnées*

Définition

*La commande **jar** est une commande permettant de simplifier la création d'archive **jar**.*

Définition

*La commande **fastjar** est une réimplémentation de **Jar** en C, plus rapide que la version originale implémentée en Java.*

Utilisation de la commande *jar*

La commande `jar` reprend les opérations (**c**, **t** et **x**) de la commande tar ainsi que ses options **v** pour les détails et **f** pour indiquer le nom du fichier :

- Créer une archive en incluant récursivement toutes les classes (l'option **-C** permet de spécifier la racine) :

```
jar cvf GrossesTetes.jar -C bin .
```

- Afficher le contenu d'une archive :

```
jar tf GrossesTetes.jar
```

- Extraire un fichier d'une archive :

```
jar xf GrossesTetes.jar
```

- Extraire le contenu d'une archive :

```
jar xf GrossesTetes.jar wwf/Animal.class
```


MANIFEST métadonnées

Définition

*Le **MANIFEST.MF** contient l'ensemble des métadonnées d'une archive jar sous forme d'un **unique** fichier texte stocké dans le répertoire **META-INF**.*

Il peut contenir entre autre :

Manifest-Version : numéro de version ;

Created-By : nom de l'auteur ;

Class-Path : nom d'autre archive contenant des dépendances

Main-Class : nom de la classe contenant la main à exécuter.

```
Manifest-Version: 1.0
Created-By: Julien Sopena
Main-Class: GrossesTetes
```


Attention à la syntaxe des MANIFEST

Attention

Le fichier **MANIFEST** doit obligatoirement :

- être encodé en *UTF8* ;
- se terminer par un retour à la ligne (dernière ligne vide) ;
- n'avoir aucun espace à la fin des différentes lignes.

Le non respect de ces règles peut entraîner des erreurs à la création du JAR ou lors de son utilisation par la JVM.

Ajout du `MANIFEST.IF`

L'option **m** permet de fournir un `MANIFEST.IF` à la création du *jar* :

```
jar cmf src/MANIFEST.MF MonAppli.jar -C bin .
```

Par défaut la commande `jar` ajoute un `MANIFEST.IF` minimal :

```
jar cf MonAppli.jar -C bin .
```

```
Manifest-Version: 1.0
Created-By: 1.7.0_07 (Oracle Corporation)
```

L'option **e** permet d'ajouter une `Main-Class` au fichier par défaut :

```
jar cef MonAppli.jar GrossesTetes -C bin .
```

```
Manifest-Version: 1.0
Created-By: 1.7.0_07 (Oracle Corporation)
Main-Class: GrossesTetes
```