**Компиляция и запуск java программ в CMD**

Например Java файл B.java берется из папки src/classes

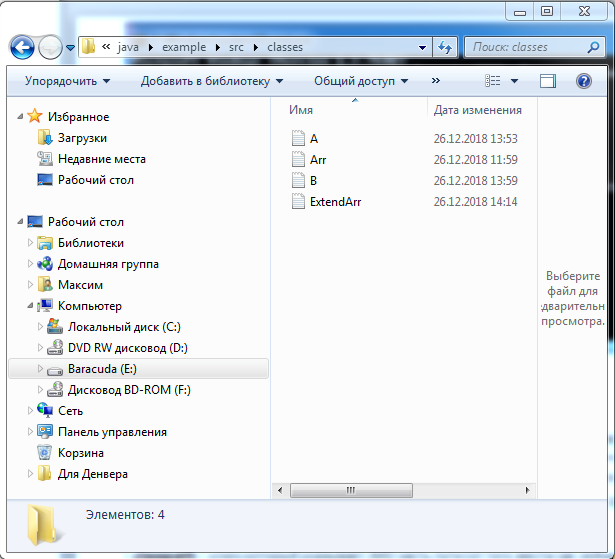


Рис.1

И компилируется в папку bin/classes

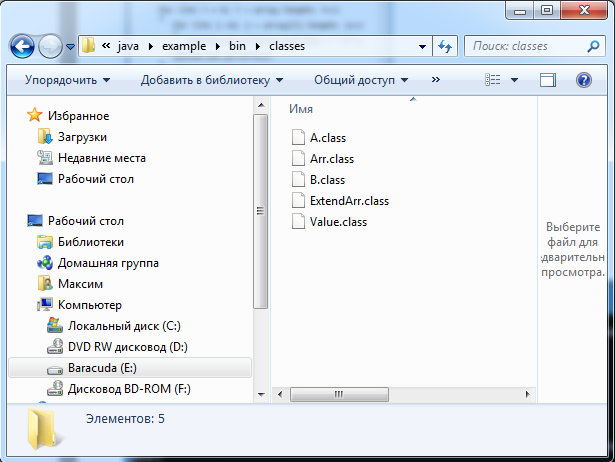


Рис.2

Подпапки **classes** после папок **src** и **bin** должны совпадать (для правильного построения логики ветвления).

Причем при компилировании подпапку classes компилятор берет из пространства имен написанного в коде файла после **package**.

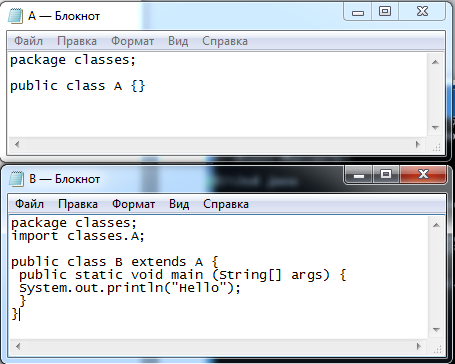


Рис.3

Код, который при этом пишется в **cmd**:

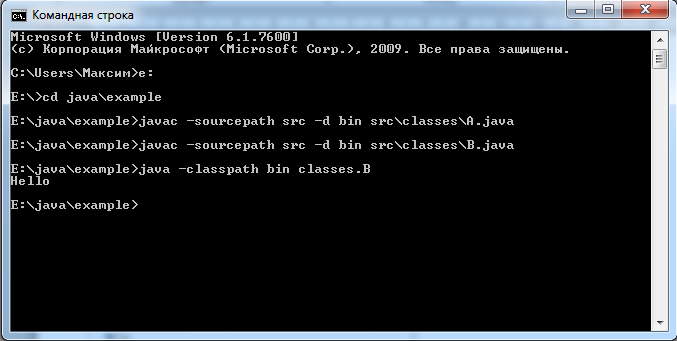


Рис.4

**Где:**

**- sourcepath** – ключ, который указывает директорию, которая содержит иерархию исходных файлов в их собственных директориях.

**–d –** ключ,после которого, прописывается часть пути (от того места где сейчас находится **cmd.** На Рис.4 это **E:\java\example>**) для сохранения скомпилированого файла B.class . + еще глубже этот файл уведет путь (подпапка/ки) прописанный в пространстве имен после оператора *package* (Рис.2/3).

Дальше в этой строке через пробел, указывается путь к файлу (от того места где сейчас находится **cmd.** На Рис.4 это **E:\java\example >**) + сам файл с расширением (в примере это B.java).

**- classpath** – ключ который указывает JVM часть пути (от того места где сейчас находится **cmd.** На Рис.4 это **E:\java\example >**) откуда нужно запустить в работу скомпилированный файл B.class.

Дальнейший путь (подпапку/ки), который указан в пространстве имен после оператора *package* (Рис.3) + само название файла/класса (без расширения), необходимо указать дальше через пробел в этой же строке.

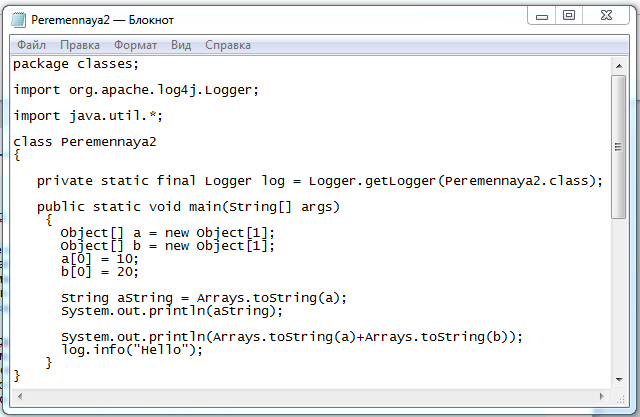
А если скомпилированный файл A.class, на который мы ссылаемся в коде, будет лежать в дополнительной подпапке, например:

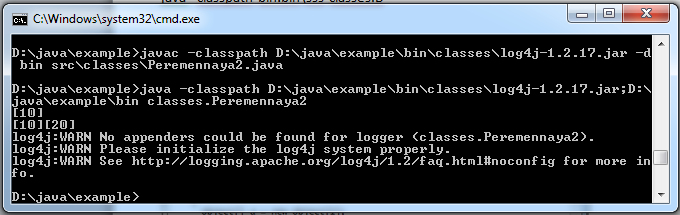
E:>java\example\bin\sss\classes\A.class

То тогда в строку запуска программы, через “:”, нужно дописать путь к “A.class” :

java –classpath bin:bin\sss classes.B

**Запуск файла с Логгированием:**





, тут  - это библиотека, которую мы используем в своей программе (страшно себе представить, как бы выглядела строка запуска моего сайтика, с тем количеством библиотек, которые там используются!!!!!)

**В classpath** при перечислении папок\файлов через “;” **писать без пробелов после “;”.**

**Это была практика, теперь глобальная теория**

**Ликбез**

* [Основы](http://www.skipy.ru/technics/likbez.html#basics)
* [Поиск классов](http://www.skipy.ru/technics/likbez.html#lookup)
* [Пишем HelloWorld](http://www.skipy.ru/technics/likbez.html#hello)

Этой статьи быть вообще не должно было. Предполагается, что читающий человек знает буквы, а музыкант – ноты. Точно так же я предполагал, что любой java-разработчик знает всё, что будет написано в этой статье. Однако события последнего времени и некоторые дискуссии в форуме меня убедили в обратном. Ладно, я понимаю, что существуют разработчики, которые не знают элементарных вещей в силу того, что заниматься Java они начали недавно. Всё когда-то узнается в первый раз, и эти элементарные вещи – не исключение.

Однако существуют програ... прог... черт, язык не поворачивается их так назвать. Они себя почему-то называют программистами. В общем, существует некий класс граждан, которые не только не знают этих вещей, но и не хотят их знать, более того, они своим незнанием гордятся. Когда я говорю, что это должен знать каждый, мне с гордостью отвечают приблизительно следующее: "Категорически не согласен! Я пишу на Java уже полгода, написал то-то и то-то, но до сих пор не знаю, как делать эту элементарную вещь!"

К счастью, таких воинствующих немного. Однако число разработчиков, не знакомых с элементарными вещами, растет угрожающими темпами. Именно поэтому и появилась эта статья.

\* \* \*

**Основы**

Начнем вот с чего. Что такое класс, надеюсь, знают все. Если нет – прошу сюда: [Наследование как явление](http://www.skipy.ru/philosophy/inheritance.html). Я там достаточно подробно останавливался на том, что такое класс с точки зрения языка. Тут мы обсудим, что такое класс с точки зрения виртуальной машины.

Кто-то может задать вопрос – что такое виртуальная машина? Отвечаю. Java – язык интерпретируемый. Что означает, что компиляция производится не в машинный код, а в некий промежуточный код, который может понимать интерпретатор. Этим достигается переносимость – без перекомпиляции код работает на любой платформе, где есть соответствующий интерпретатор. Этот интерпретатор называют еще виртуальной машиной, хотя понятие "виртуальная машина" более емкое. Я бы предпочел считать, что виртуальная машина – это система, обеспечивающая исполнение Java-кода.

Пойдем дальше. Класс описывается в своем файле, который имеет расширение *.java*. Это – ИСХОДНЫЙ КОД. Для исполнения его необходимо скомпилировать. (Это я говорю специально для тех, кто пытается запустить .java-файл). После компиляции (успешной!) появляется файл с расширением *.class* и тем же именем, что и файл *.java*. (Внутренние классы и прочие исключения мы пока во внимание не принимаем.) Где этот класс будет лежать – мы разберемся позже.

Теперь задайте себе вопрос. А сколько классов можно держать рядом и не запутаться? Десять? Сто? Тысячу? Тысяча исходников в одной директории заставит поседеть любого. Это нереально. А между тем, в моем текущем проекте около тысячи файлов с исходным кодом. А в прошлом было около трех с половиной тысяч. И в этом коде надо разбираться, причем хорошо. Что делать?

Ответ напрашивается. Структурировать их. Разнести по куче директорий. Создать дерево директорий, в котором легко ориентироваться. И именно это и сделано в Java. Мы подходим к такому понятию как *пакет*.

Пакеты иерархичны. Иерархия описывается через '.', потому точка не может быть в имени пакета. Пакет, к которому принадлежит класс, описывается в самом начале исходного файла. Если это описание есть, оно должно быть первым (раньше него могут быть только комментарии):

**package** ru.skipy.web;

Директива, приведенная выше, означает, что класс принадлежит к пакету *ru.skipy.web*. Для обеспечения уникальности имен пакетов принято в начало добавлять реверсированный домен второго уровня разработчика пакета. Т.е., если у меня есть домен *skipy.ru*, то имена всех моих пакетов должны начинаться с *ru.skipy*. Но это не требование, а, скорее, джентельментское соглашение.

Иерархия пакетов может не совпадать с физической иерархией исходников. Но она ОБЯЗАНА совпадать с физической иерархией скомпилированных классов. Об этом заботится компилятор. Он создает древовидную структуру, в точности повторяющую структуру пакетов, и раскладывает скомпилированные исходники в этой структуре.

Имя пакета является частью полного имени класса. И потому появляется возможность создавать классы с одним именем, но в разных пакетах. Классический пример – *java.awt.List* и *java.util.List*. Первый из них – компонента пользовательского интерфейса (список элементов), второй – интерфейс для работы с упорядоченными списками.

Однако, компилятору нужно указать, откуда брать класс. Он не обладает телепатическими способностями, следовательно, при упоминании *List* он зайдет в тупик. Выходов тут два. Первый, более предпочтительный – *импортировать* класс:

**import** java.util.List;

В этом случае компилятор будет знать, что все ссылки на *List* в коде – это, на самом деле ссылки на *java.util.List*.

Можно импортировать и весь пакет *java.util* целиком:

**import** java.util.\*;

Это полезно, когда из пакета берется много класов и, соответственно, нужно много директив импорта. Многие среды разработки настраиваются так, что при импорте более чем n (как правило, 5) классов импорты отдельных классов заменяются на импорт пакета. Это, возможно, влияет на скорость компиляции (класс надо искать во всех импортированных целиком пакетах), но настолько незначительно, что во внимание принимать это не следует. Кстати, если импортировать пакет или класс, но не использовать его, на байткод (скомпилированный код) это не повлияет!

Минусов при импортировании пакетов два. Во-первых, нарушается наглядность. Видно, что используется класс из пакета, но не видно, какой. И во-вторых, если одновременно импортировать два пакета, в которых будут классы с одинаковым именем, то при использовании таких классов (обратите внимание – ТОЛЬКО при использовании; простой импорт не вызовет проблем!) компилятор опять зайдет в тупик, будучи не в состоянии определить, какой из классов нужен – из пакета *java.awt* или из *java.util*.

Вот тут спасает факт, о котором я упомянул – пакет является частью имени класса. Это означает, что тот самый *java.util.List* можно так и использовать:

java.util.List list = **new** ArrayList();

И в этом случае у компилятора проблем не возникнет. Разумеется, *java.awt.List*, в случае необходимости его использования, тоже придется идентифицировать по полному имени.

Более того, при использовании полного имени можно вообще не импортировать классы. Однако, я не рекомендую этого делать по соображением читаемости. Представьте себе наличие в тексте кучи префиксов типа *com.sun.org.apache.xalan.internal.xsltc.trax* – этот текст будет невозможно читать. Между тем, пакет абсолютно реальный, из jdk5.0. И, кстати, не самый длинный.

Как я уже говорил, иерархия исходников может не совпадать с иерархией пакетов. Однако я очень рекомендую поддерживать их в соответствии. Собственно, все современные среды разработки делают это сами. Но знать об этом надо.

А зачем вообще нужно использовать пакеты? И нельзя ли обойтись без них? Теоретически – да, можно. Практически – сложно. Ибо какой-либо ваш класс по имени может совпасть с классом из импортируемого пакета. В этом случае ВАШ класс использовать просто не удастся, т.к. компилятор не сможет отличить его по имени от импортированного. А по полному имени, вместе с пакетом, – не сможет ввиду наличия отсутствия вышеозначенного.

Итак, с пакетами покончили. Перейдем дальше.

Представьте себе законченный набор классов, реализующий некую функциональность. Например, журналирование (logging). У этого набора есть ядро и куча интерфейсов, которые дают возможность его использовать. И называется это набор... правильно, библиотека. Как правило, все классы собраны в один файл с расширением jar (Java ARchive). Вопрос. Как именно их использовать?

Казалось бы, можно просто импортировать нужные классы или пакеты и использовать их. И это действительно так. Тогда второй вопрос. Как компилятор узнает, откуда именно брать классы? В момент компиляции ему нужны классы, чтобы проверить правильность вызовов, поставить ссылки и т.п. Да и в процессе выполнения интерпретатору тоже нужно знать, где брать классы. Как ему это сказать?

Кстати, точно тот же вопрос касается и кода, написанного Вами. Интерпретатор точно так же должен знать, где его брать.

Существует несколько способов. Однако, практика показывает, что подавляющее большинство разработчиков, особенно начинающих, имеют об этих способах весьма смутное представление. Более того, зачастую этим незнанием гордятся! Этого я решительно не понимаю. И принимая во внимание, сколько проблем вызывает незнание этой темы, я расскажу обо всех известных мне способах указания интерпретатору, откуда брать классы, а так же коснусь их плюсов и минусов.

Итак,

**Указание интерпретатору, где искать классы**

Интерпретатору нужно знать все места, где он может найти классы. Что является таким местом? Директория и/или библиотека – jar-файл. Прошу обратить особое внимание – именно **файл**. Иначе говоря, если у вас файл javaee.jar находится в директории E:\java\lib\ – интерпретатору нужен **полный** путь к файлу, а именно – E:\java\lib\javaee.jar. Простого указания директории НЕДОСТАТОЧНО!

Как я уже говорил, существует несколько способов указания интерпретатору на источники классов. Они рассмотренны ниже. (Поиска классов в сервлет-контейнерах и серверах приложений я не касаюсь, это выходит за рамки статьи для начинающих.)

**Способ 1. Переменная окружения CLASSPATH**

Способ прост. Заводится переменная окружения CLASSPATH, и в ней через разделитель (';' для Win, ':' для \*NIX) прописываются все полные пути к файлам библиотек и к директориям, в которых лежат деревья классов. Обратите внимание, что директория – это точка, в которой лежит КОРЕНЬ всех скомпилированных классов. Т.е., если классы из пакета mypackage.test компилировались в директорию c:\myproject\classes, то в CLASSPATH должна фигурировать именно c:\myproject\classes, а не c:\myproject\classes\mypackage и не c:\myproject\classes\mypackage\test.

Способ простой, однако имеет несколько недостатков, за которые я его очень не люблю. Первое. Хорошо, библиотеки можно подключить так. Да и то с оговорками, см. ниже. А как быть с текущими классами? С теми, которые я как раз и пишу? Их тоже надо подключать. Включить все? Нереально, у меня больше 20 проектов, за деревьями не будет видно леса, если для каждого случая в CLASSPATH указать полный путь. Указывать относительный? Он тоже разный. Честно сказать, приемлемого решения я не нашел.

Второе. Допустим, я хочу подключить библиотеки (jar-файлы) через CLASSPATH. Вопрос. А сколько их у меня? Поверхностный поиск по диску дает... более 1550! Ладно, львиная доля из них – от приложений. Но тех, которые использую я сам, прямо или опосредованно, более 100 совершенно точно. Что будет, если в CLASSPATH прописать их все? Хватит ли размера памяти, выделяемого под переменную окружения операционкой? А если все приложения сделают то же самое?

Третье, самое неприятное. У меня стоят Log4J версий 1.2.8 и 1.3.0, Velocity 1.2, 1.3 и 1.4, Hibernate 2.1 и 3.0, Servlet 2.2, 2.3, 2.4 и JSP 1.1, 1.2 и 2.0, JDBC-MySQL трех различных версий, несчетное множество XML-парсеров и т.д. и т.п. В одних проектах я использую одни, в других – другие. Если я включу их все в CLASSPATH – компилятор будет ВСЕГДА использовать первый найденный класс в порядке перечисления библиотек. Какой версии? Не знаю. Но в любом случае – одной и той же. А мне нужны разные. И если первые два момента можно было бы как-то пережить, то этот – нет.

Потому – я никогда не устанавливаю у себя переменную окружения CLASSPATH. Тем более, что существует...

**Способ 2. Указание ключа -classpath интерпретатору (компилятору)**

Этот способ, на мой взгляд, наилучший. Я могу указать интерпретатору ровно те библиотеки, которые мне нужны. И ровно те источники кода, из которых он этот код должен брать. Не больше и не меньше.

Пример. Допустим, у меня в проектной директории есть поддиректория classes, в которой находятся скомпилированные классы, и поддиректория lib, в которой лежат библиотеки velocity-1.4.jar и log4j-1.2.8.jar. Тогда командная строка для запуска моего класса ru.skipy.myproject.Main будет выглядеть так:

java -classpath ./classes;./lib/velocity-1.4.jar;./lib/log4j-1.2.8.jar ru.skipy.myproject.Main

По пунктам: я указываю интерпретатору, что классы надо искать в директории ./classes ('.' – текущая директория), в библиотеке ./lib/velocity-1.4.jar и в библиотеке ./lib/log4j-1.2.8.jar, в порядке их перечисления.

Если же мне не нужно использовать внешние библиотеки, то командная строка превращается в следующую:

java -classpath ./classes ru.skipy.myproject.Main

Обратите внимание на -classpath ./classes. Я не знаю, откуда в некоторых книгах взялась конструкция java ru.skipy.myproject.Main, без указания classpath. Я не поленился, поставил себе jdk1.1.8, но и там classpath надо указывать. Именно отсутствие classpath в параметрах интерпретатора является источником большого количества проблем. Впрочем, об этом ниже.

Справедливости ради нужно упомянуть и третий способ подключения библиотек:

**Способ 3. Директория <JRE\_INSTALLATION\_DIRECTORY>/lib/ext**

Способ, пожалуй, самый простой. Но, должен признать, он пригоден только для подключения jar-файлов. Файл кладется в указанную директорию. Всё. При старте виртуальной машины библиотека подключается.

Недостаток очевиден. У меня в системе, к примеру, стоит 6 виртуальных машин, не считая тех, что шли с приложениями вместе: 1.1.8 (для которой этот способ вообще не работает), 1.3.1, 1.4.2, 1.5.0, JRockIt 1.4.2, JRockIt 1.5.0. Библиотеку придется класть КАЖДОЙ. Вопрос. Зачем?

Естественно, у этого метода есть и достоинства. Он хорош, когда нужно что-то поставить на клиентскую машину, где есть всего одна JRE (и то если есть). Есть также небольшие отличия с точки зрения безопасности (в разрешенных действиях) между кодом, подгруженным через classpath и кодом, загруженным из <JRE\_INSTALLATION\_DIRECTORY>/lib/ext. Но это уже тонкости, выходящие за рамки этой статьи.

А что будет, если не указать таки classpath, ни в качестве переменной окружения, ни в качестве параметра интерпретатору? А будет всем известная ошибка. Вопрос о том, что это такое, появляется не реже раза в неделю. Что в числе прочего и побудило меня написать эту статью. Ошибка эта называется ...

**Exception in thread "main" java.lang.NoClassDefFoundError: ...**

Вместо '...' пишется имя класса, который виртуальная машина не смогла найти. Ошибка эта означает, что определение класса не найдено. Под определением класса виртуальная машина понимает его байткод. Т.е. она просто не смогла найти нужный файл.

Иногда встречается более изощренная разновидность этой ошибки. Допустим, класс, который нужно запустить, называется test.Test. Одна из классических разновидностей действия: скомпилировать класс, потом дойти до директории, где лежит непосредственно Test.class и запустить его оттуда:

javac Test.java

@rem file ./test/Test.class created

cd test

java -classpath . Test

Ошибка при этом будет такой:

Exception in thread "main" java.lang.NoClassDefFoundError: Test (wrong name: test/Test)

at java.lang.ClassLoader.defineClass0(Native Method)

at java.lang.ClassLoader.defineClass(ClassLoader.java:539)

at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)

at java.net.URLClassLoader.defineClass(URLClassLoader.java:251)

at java.net.URLClassLoader.access$100(URLClassLoader.java:55)

at java.net.URLClassLoader$1.run(URLClassLoader.java:194)

at java.security.AccessController.doPrivileged(Native Method)

at java.net.URLClassLoader.findClass(URLClassLoader.java:187)

at java.lang.ClassLoader.loadClass(ClassLoader.java:289)

at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:274)

at java.lang.ClassLoader.loadClass(ClassLoader.java:235)

at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:302)

Что это означает? А означает это вот что. Виртуальная машина нашла класс, который я пытаюсь запустить – Test. Нашла она его исключительно потому, что я сказал – искать в этой директории класс по имени Test. Прошу обратить внимание – по ПОЛНОМУ имени Test. Машина нашла его. Но класс-то на самом деле – test.Test. Именно об этом она и сообщает – wrong name: test/Test

Еще одна ошибка, вызывающая появление сообщения Exception in thread "main" java.lang.NoClassDefFoundError: имя класса при запуске пишут вместе с расширением файла .class. Естественным образом виртуальная машина не находит этого класса, т.к. расширение считает частью имени, приписывает к имени .class и начинает искать файл с именем, заканчивающимся на class.class

Для продвинутых разработчиков сущестует еще один способ получить java.lang.NoClassDefFoundError. Заключается он в следующем: приложение, использующее стороннюю библиотеку, [собирается в jar-файл](http://www.skipy.ru/technics/likbez.html#step5) и запускается командой java -jar <имя jar-файла>, после чего и появляется данная ошибка. Происходит это по следующей причине: при запуске приложения именно таким образом, с ключом интерпретатора -jar, в classpath включается один единственный файл – тот самый, который указывается в командной строке. Все остальные библиотеки – описаные в переменной CLASSPATH, указаные через ключ -classpath, – все они ИГНОРИРУЮТСЯ. Единственный способ этого избежать (и найден он – вот ведь поразительный факт! – опять-таки в [документации](http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#JAR%20Manifest)) – указать в файле manifest.mf атрибут Class-Path – список относительных путей (обращаю особое внимание – *относительных*, относительно этой библиотки!) к необходимым библиотекам. Разделяются эти пути пробелами. Естественно, при этом работает и обычный вариант – указывать в явном виде classpath, включая туда все библиотеки, и имя исполняемого класса.

Еще один тип ошибок – при компиляции выдается сообщение *"Can't resolve symbol ..."* с указанием на точку в коде, где объявлена переменная, вернее, на тип этой переменной. Причина та же – компилятор не может найти класс, указанный как тип этой переменной по причине отсутствия этого класса в classpath.

Возможно, есть еще какие-нибудь характерные ошибки, связанные с classpath. Если вспомню – допишу.

Ну и напоследок. Простейшее упражнение на закрепление пройденного материала.

**Пишем программу Hello, World!**

**Шаг 1. Создание структуры директорий**

Единственное допущение – мы находимся в директории-корне проекта HelloWorld.

mkdir src

mkdir classes

cd src

mkdir mypackage

cd ..

Что мы сделали. Создали директорию для исходных файлов – src. Создали директорию для скомпилированных файлов – classes. Создали внутри src директорию, соответствующую имени пакета – mypackage.

**Шаг 2. Создание исходного файла**

В директории src/mypackage создаем исходный файл – HelloWorld.java:

**package** mypackage;

**public class** HelloWorld{

**public static void** main(String args[]){

System.out.println("Hello, World!");

}

}

**Шаг 3. Компиляция**

Обратите внимание, в командной строке мы находимся в корне проекта! Команда компиляции выгдядит так:

javac -classpath ./classes -d ./classes src/mypackage/HelloWorld.java

Обратите внимание на ключ -d. Этот ключ указывает компилятору, где находится корень для скомпилированных классов. Если не указать его, корнем будет считаться текущая директория. При компиляции нескольких классов в пакете одновременно можно вместо имени конкретного файла указать \*.java.

В результате компиляции в директории ./classes создается поддиректория mypackage, в которую помещается файл HelloWorld.class

**Шаг 4. Выполнение программы**

java -classpath ./classes mypackage.HelloWorld

Еще раз обращаю особое внимание. Имя класса указывается вместе с пакетом! И указывается именно имя класса, а не файла! Т.е. .class в конце добавлять не надо.

Специально распишу логику поиска файла с байткодом:

1. Все точки в полном имени класса заменяются на разделители – '/'
2. К полученной строке приписывается .class
3. Файл ищется в библиотеках и указанных директориях со скомпилированными классами в порядке их указания. Файл ищется по полученному на прошлом шаге относительному пути.

Т.е. в нашем случае имя класса mypackage.HelloWorld преобразуется в относительный путь к файлу mypackage/HelloWorld.class. Виртуальная машина находит файл по данному относительному пути в директории ./classes.

**Шаг 5. Создание архива (jar-файла)**

Собираем архив, который можно запускать с помощью команды java -jar <имя архива>. Для начала создаем файл manifest.mf, в котором будет указан главный исполняемый класс:

Manifest-Version: 1.0

Created-By: 1.6.0\_19 (Sun Microsystems Inc.)

**Main-Class: mypackage.HelloWorld**

**Важный момент!** После последней строки – в данном случае это указание Main-Class – **должен быть перевод строки**. Если его не будет – последняя строка не будет прочитана. Скорее всего, это ошибка в реализации jar. Как бы то ни было – если в данном примере вы не поставите в конце перевод строки, основной класс найден не будет и при запуске вы получите ошибку :

Failed to load Main-Class manifest attribute from helloWorld.jar

Спасибо разработчику из Санкт-Петербурга Алексею Яблокову за указание на этот момент!

Итак, manifest.mf создали, перевод строки не забыли. Идем дальше. Выполняем команду по сборке архива:

jar cvmf manifest.mf helloWorld.jar -C ./classes mypackage

Специально подчеркиваю – порядок ключей *важен*. В каком порядке стоят m и f – в таком же порядке должны стоять имена manifest- и jar- файлов соответственно. manifest.mf – это имя файла, в котором мы указали исполняемый класс. В принципе, этот файл может называться как угодно, как правило, имя такое (так этот файл называется внутри jar-архива, и это имя уже фиксировано). helloWorld.jar – имя создаваемого архива. Конструкция "-C ./classes mypackage" означает "сменить директорию на classes и взять там директорию mypackage" (напоминаю еще раз – все команды выполняются из корневой директории проекта).

В результате должен быть создан файл helloWorld.jar со следующей структурой:

META-INF/MANIFEST.MF

mypackage/HelloWorld.class

**Шаг 6. Выполнение программы запуском jar-файла**

Выполняем следующую команду:

java -jar helloWorld.jar

Эту команду необязательно выполнять из той директории, где находится корень проекта. Можно указать полный путь, например, так:

java -jar c:\myprojects\HelloWorld\helloWorld.jar

Разница только в том, что текущей директорией приложения при запуске будет считаться та, из которой вы запустили приложение на выполнение. При использовании полного пути она может быть любой, это иногда бывает полезно.

В результате выполнения этой команды, как и на [шаге 4](http://www.skipy.ru/technics/likbez.html#step4), в консоль будет выведена строка "*Hello, World!*"

Упражнение окончено.

**Еще один важный момент!**. При запуске приложения через jar-файл, командой java -jar ..., в classpath попадает **только этот jar-файл**. Никакие попытки указать ключ -classpath ни к чему не приведут. Однако не всё так страшно, необходимые библиотеки указать можно. Для этого необходимо в манифесте (manifest.mf) указать относительные пути к библиотекам. Т.е., если нам надо подключить, например, библиотеки velocity-1.4.jar и log4j-1.2.8.jar, лежащие в директории lib рядом с нашим jar-файлом, то в манифест надо добавить строчку:

Manifest-Version: 1.0

Created-By: 1.6.0\_19 (Sun Microsystems Inc.)

Main-Class: mypackage.HelloWorld

**Class-Path: lib/velocity-1.4.jar lib/log4j-1.2.8.jar**

Указываются пути относительно jar-файла, разделитель – пробел. И в этом случае в classpath будет включен не только наш jar-файл, но и обе указанные библиотеки